סיכום קריפטו

24 ביולי 2022

תוכן עניינים

חלק I

קריפטוגרפיה:

0.1 הרצאה 1 - הגדרת מערכת, עקרונות בטיחות ומערכת סודיות מושלמת:

- **התנהגות עויינת:** התנהגות שבונה המערכת לא הגדיר כהנהגות המערכת.
- **מערכת:** אובייקט עבורו הגדרנו אופן פעולה מסויים, או מערכת קלט פלט.

0.1.1 מערכת הצפנה סימטרית:

- מערכת המאפשרת לשני אנשים לתקשר בצורה סודית אף אם יש אדם שלישי המאזין לתקשורת.
- **הנחה - מפתח הצפנה:** שני אנשים שרוצים לתקשר מסכימים על מפתח הצפנה משותף, שומרים עותק שלו ופועלים על פיו.
- **מערכת הצפנה סימטרית פועלת ע"פ שלשה אלגוריתמים:**

1 $keyGen$**:** אלגוריתם הסתברותי ליצירת מפתחות. מחזיר מפתח הנדגם ע"י התפלגות כלשהי. נסמן את המפתח - k, ואת אוסף כל המפתחות ב $\mathcal{K}$.

2 Enc**- אלגוריתם ההצפנה:** יתכן דט' ויתכן הסתברותי. מקבל מפתח והודעה ומחזיר כפלט הודעה מוצפנת. נסמן ב m את ההודעה. $\mathcal{M}$- אוסף כל ההודעות בהן המערכת תומכת. c - ההודעה המוצפנת. $\mathcal{C}$ - אוסף כל ההודעות המוצפנות בהן המערכת תומכת.

3 Dec **- אלגוריתם פענוח:** דט' מקבל מפתח k והודעה מוצפנת c, ומחזיר הוגעה m.

- **סימונים:**

אלגוריתם דט' נסמן ב " $=$ ".

אלגוריתם הסתברותי נסמן ב " $\Rightarrow$ ".

- **הגדרה - נכונות:** לכל מפתח k והודעה m אם נצפין את m בעזרת k ואח"כ נפענח את ההדעה בעזרת k, נקבל חזרה את m.

$$Dec_k\left(Enc_k(m)\right) = m$$

- **בטיחות:**

1: מספר המפתחות אמור להיות גדול ממש כדי שהתוקף לא יוכל לעבור על המפתחות אחד אחרי השני בזמן סביר.

2: נעדיף שלא יהיה מיפוי קבוע לכל אות.

0.1.2 מערכות הצפנה:

- **צופן הסטה:** נזיז כל אות ב k אותיות מימין אליה (מספר המפתחות - 26).
- **צופן החלפה:** נחליף כל אות באות אליה היא ממופה לפי המפתח (מספר המפתחות - $26!$).
- **צופן וגנר:** נדגום וקטור של מספרים באורך t, כאשר כל אחד מ t המספרים נבחר באופן אחיד 0-25 . מספר המפתחות האפשרים הוא 26^t עבור t כלשהו שבחרנו.
  בהצפנה - מסיטים כל אות במיקום ה i של ההודעה ב $k_{i \ mod(t)}$ קדימה.

0.1.3 עקרונות בטיחות:

- **1 - נגדיר הגדרות בטיחות -** מה אנו רוצים להשיג. מפני אילו התקפות אנו רוצים להגן, איך התוקפים יתקפו ואיה כח חישוב עומד לרשותם.
  נגדיר מה נחשב שבירת בטיחות - לדוג' שחזור המפתח, או תוקף שהשיג מידע על הודעה שהוצפנה.
- **2 -** תיאור ההנחות עליהם המערכת מתבססת, לדוג' שימוש במספרים ראשוניים.
  נפרסם את כל ההנחות בכדי שחוקרים אחרים יחשפו אליהן, וגם בכדי שנוכל לייעל את המערכת ולהשוות בין מערכות.
- **3 -** ניתן לעשות שימוש בשני העקרונות הראשונים בכדי להוכיח באופן מתמטי כי המערכת בטוחה.

0.1.4 סודיות מושלמת:

- **הגדרת בטיחות - סודיות מושלמת:**
  ההודעה המוצפנת c לא מגלה שום מידע על ההודעה m. (מלבד הידע הקודם של התוקף על ההתפלגות של ההודעה m).
- **הגדרה - נאמר כי מערכת הצפנה Π מספקת סודיות מושלמת אם:** לכל התפלגות המיוצגת ע"י מ"מ M מעל מרחב ההודעות - $\mathcal{M}$, עבור כל $m \in \mathcal{M}$ ולכל $c \in \mathcal{C}$ עם הסתברות חיובית $Pr[C = c] > 0$ מתקיים כי:

$$Pr[M = m \mid C = c] = Pr[M = m]$$

  כלומר - ההסתברות כי מ"מ M שווה להודעה ספציפית m, **בהינתן** ש מ"מ c שווה להודעה מוצפנת ספציפית C, שווה להסתברות כי המ"מ m שווה להודעה ספציפית M.
  כלומר - העובדה שהתנינו על כך שההצפנה היא ספציפית לא שינה לנו דבר על הידע שלנו על ההתפלגות של ההודעה המוצפנת.
- **במילים אחרות:** הגדרת הסודיות המושלמת מבקשת **אי תלות** בין ההתפלגויות M, C. אם אכן יש אי תלות כזו אזי ההודעה המוצפנת לא מגלה ידע חדש על ההצפנה.
- **מערכת סודיות מושלמת לדוגמה -** $OTP = one - time\ pad$**:**
  נצפין בעזרת xor של ההודעה m ושל המפתח k.

חסרונות:

1: אורך המפתח צריך להיות גדול שווה לאורך ההודעה.

2: ההצפנה תקפה להודעה אחת בלבד והצפנת שתי הודעות בעזרת אותו המפתח תחשוף מיהו המפתח k.

3: אם התוקף יודע מה תוכן ההודעה m, אזי הוא יכול לשחזר את המפתח ולפענח בעזרתו כל הודעה.

- **משפט:** תהי Π מערכת הצפנה סימטרית, אם Π מקיימת את הגדרת הסודיות המושלמת אזי $|\mathcal{K}| \geq |\mathcal{M}|$. כלומר - מרחב המפתחות גדול שווה ממרחב ההודעות.

0.2 הרצאה 2 - בטיחות חישובית:

- **בטיחות חישובית** $Computational\ security$**:** בהינתן הצפנה של הודעה m לא ניתן להסיק כל מידע **מועיל** על ההודעה בעזרת חישוב.

- **נחליש את הדרישות באופן הבא:**

1: נדרוש בטיחות רק כנגד יריבים חסומים חישובית.

2: נאשר ליריבים לשבור את המערכות שלנו בהסתברות זניחה אך שונה מ 0.

- **כיצד נעשה זאת:**

1 - גישת הבטיחות הקונקרטית: עבור t, ϵ נגיד שסכמה קריפטוגרפית היא t, ϵ בטוחה, אם כל יריב שרץ בזמן לכל היותר t, מצליח לשבור את הסכמה בהסתברות לכל היותר ε.

נקודות חזקה: ניתן להתאים את t, ϵ למערכות שלנו ולכח החישוב המצוי כיום.

חסרונות: כשהמערכות ישתכלל הגחשה תהיה לא רלוונטית. בנוסף היא לא מספקת לנו וודאות על יריב שרץ בזמן $t <$.

2 - גישת הבטיחות האסימפטוטית: ניפטר מרגישות לשינויים קטנים, נאמר שסכמה קריפטוגרפית היא בטוחה אם כל יריב הסתברותי פולינומיאלי יצליח לשבור אותה אך ורק בהסתברות זניחה $(negligible)$.

כיצד נדע אם הקלט של היריב פולינומיאלי: נוסיף פרמטר בטיחות שיסומן ב n, כל האלגוריתמים יקבלו אותו בתור קלט.

- **הגדרה אלגוריתם פולינומיאלי:** נאמר כי אלגוריתם A רץ בזמן פולינומיאלי אם זמן הריצה שלו על קלט x קטן מפילונום של x.

- **הגדרה אלגוריתם הסתברותי:** אלגוריתם שקובע באופן הסתברותי את ההתקדמות.

- **הגדרה - פונקציה זניחה:** המטרה היא לתפוס את הקצב שבו פונקציה חיובית קטנה לכיוון אפס.

פונקציה זניחה היא פונקציה הקטנה לכיוון 0 מהר יותר מההופכי של כל פולינום קבוע.

באופן פורמלי: פונקציה f תיקרא זניחה אם לכל פולינום P קיים N כך שלכל $n > N$ מתקיים $f(n) < \frac{1}{P(n)}$.

- **טענה - זניחות וחיבור:** עבור כל שני פונקציות זניחות f, g גם חיבור שלהן וכפל בפולינום $P \cdot (f + g)$ היא פונקציה זניחה.

- **הצפנות בלתי ניתנות להבחנה** (IND)**:**

1: לא ניתן יהיה להבחין בין כל זוג של הודעות שונות.

2: היריב יצפה רק בהצפנה של הודעה אחת.

באופן פורמלי: נאמר כי מערכת הצפנה היא בלתי ניתנת להבחנה אם לכל יריב הסתברותי פולינומיאלי A יש פונקציה זניחה ν כך ש

$$Pr\left[IND(n)=1\right] \leq \frac{1}{2}+\nu(n)$$

כלומר - הסתברות היריב לנחש נכון איזו הודעה מבין m_0, m_1 היא ההודעה שהוצפנה, היא לכל היותר $\frac{1}{2}+\nu(n)$.

חזרה: הצפנות בלתי ניתנות להבחנה

תהא $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ מערכת הצפנה. לכל אלגוריתם A ולכל $n \in \mathbb{N}$ נגדיר את הניסוי $\mathsf{IND}_{\Pi,A}(n)$ באופן הבא:

1. מגרילים מפתח $k \leftarrow \mathsf{KeyGen}(1^n)$.
2. A מקבל בתור קלט 1^n ופולט זוג הודעות (m_0, m_1).
3. מגרילים ביט $b \leftarrow \{0,1\}$, מחשבים $c^* \leftarrow \mathsf{Enc}_k(m_b)$ ומעבירים את c^* ל־A.
4. A מחזיר ביט $b' \in \{0,1\}$.

תוצאת הניסוי היא 1 (נסמן זאת ע"י $\mathsf{IND}_{\Pi,A}(n)=1$) אם $b'=b$. אחרת, תוצאת הניסוי היא 0 (נסמן זאת ע"י $\mathsf{IND}_{\Pi,A}(n)=0$).
נאמר ש־Π הינה בעלת הצפנות בלתי ניתנות להבחנה (ובקצרה, Π היא $\mathsf{IND-secure}$) אם לכל אלגוריתם הסתברותי A הרץ בזמן פולינומיאלי קיימת פונקציה זניחה $\nu(\cdot)$ כך ש:

$$\Pr\left[\mathsf{IND}_{\Pi,A}(n)=1\right] \leq \frac{1}{2}+\nu(n)$$

לכל $n \in \mathbb{N}$ גדול דיו.

- **טענה:** תהי Π מערכת הצפנה בלתי ניתנת להבחנה, ויהי B אלג הסת פול המקבל את n בתור קלט והצפנה של הודעה m הנדגמת באופן אחיד מאוסף כל המחרוזות באורך l ביטים, (מטרת האלגוריתם היא לנחש את הביט הראשון). קיימת פונקציה זניחה ν כך ש:

$$Pr\left[\mathcal{B}\left(1^n, Enc(m)\right)=LSB(m)\right] \leq \frac{1}{2}+\nu(n)$$

- **יצרן פסאודו אקראי** (PRG)**:** פונקציה שנסמן ב $G:\{0,1\}^* \to \{0,1\}^*$, אנו רוצים לקבל קלט קצר המפולג באופן אחיד הנקרא $seed$. ולהחזיר פלט ארוך יותר ממנו שלא ניתן להבחין בינו לבין ערך המפולג באופן אחיד באמת.

- **הגדרה - יצרן פסאודו אקראי:** מקיים את תכונת ההרחבה ופסאודו אקראיות.

חזרה: יצרן פסאודו־אקראי

תהא $G : \{0,1\}^* \to \{0,1\}^*$ פונקציה החשיבה בזמן פולינומיאלי, ויהי $\ell(\cdot)$ פולינום כך שלכל $n \in \mathbb{N}$ ולכל $s \in \{0,1\}^n$ מתקיים $G(s) \in \{0,1\}^{\ell(n)}$. נאמר ש־$G$ היא יצרן פסאודו־אקראי אם מתקיימים שני התנאים הבאים:

1. **הרחבה:** $\ell(n) > n$ לכל $n \in \mathbb{N}$.
2. **פסאודו־אקראיות:** לכל אלגוריתם הסתברותי D הרץ בזמן פולינומיאלי, קיימת פונקציה זניחה $\nu(\cdot)$ כך ש:

$$\left| \Pr_{s \leftarrow \{0,1\}^n}[D(G(s)) = 1] - \Pr_{r \leftarrow \{0,1\}^{\ell(n)}}[D(r) = 1] \right| \leq \nu(n)$$

לכל $n \in \mathbb{N}$ גדול דיו.

- **עובדה - התפלגות הפלט של יצרן פסאודו אקראי:** כל תכונה של ההתפלגות האחידה שניתנת לזיהוי ע"י אלגוריתם הסתברותי פולינומיאלי , צריכה להתקיים גם עבור התפלגות הפלט של כל יצרן פסאודו אקראי.
- **מערכת הצפנה סימטרית** (PRG)**:** יהי G יצרן פסאודו אקראי, מרחב המפתחות באורך n ביטים, אך מרחב ההודעות ומרחב ההודעות המוצפנות באורך $l(n)$ ביטים (מרחב המפתחות קטן ממש ממרחב ההודעות).
  אלגוריתם ההצפנה יעבוד כך: $(m)\ XOR\ (G(k))$.
  נשים לב: בדומה ל OTP גם כאן כל מפתח רלונטי להודעה אחת בלבד.
- **משפט:** אם G יצרן פסאודו אקראי אזי המערכת PRG מספקת את הגדרת הבטיחות של הצפנות בלתי ניתנות להבחנה.
- **הגדרת הבטיחות הסמנטית:** דורשת שכל מה שניתן לחשב ביעילות בהינתן הצפנה כלשהי של הודעה m הנדגמת בהתפלגות ידועה, אפשר גם לחשב בלי לקבל את ההודעה m כלל.
  באופן פורמלי:

Definition:
Π is **semantically secure** if for every PPT adversary $\mathcal{A}$ there exists a PPT "simulator" $\mathcal{S}$ such that for every efficiently-sampleable plaintext distribution $M = \{M_n\}_{n\in\mathbb{N}}$ and all polynomial-time computable functions f and h, there exists a negligible function $\nu(\cdot)$ such that

$$|\Pr[\mathcal{A}(1^n, \mathrm{Enc}_k(m), h(m)) = f(m)] - \Pr[\mathcal{S}(1^n, h(m)) = f(m)]| \leq \nu(n)$$

where $k \leftarrow \mathrm{KeyGen}(1^n)$ and $m \leftarrow M_n$.

0.3 הרצאה 3 - הצפנות בלתי ניתנות להבחנה חישובית:

- **בלתי ניתנות להבחנה חישובית** $Computational\ indistinguishability$**:** נאמר כי שתי התפלגויות $X = \{X_n\}_{n\in\mathbb{N}}, Y = \{Y_n\}_{n\in\mathbb{N}}$ בלתי ניתנות להבחנה חישובית אם אף אלגוריתם יעיל (הסתברותי פולינומיאלי) D לא יוכל להבחין בניהן.

פורמלית:

$$|Pr\left[\mathcal{D}\left(1^n, x\right) = 1\right] - Pr\left[\mathcal{D}\left(1^n, y\right) = 1\right]| \leq v(n)$$

ונסמן: $X \approx^c Y$

- **משפט:** יהי G יצרן פסואודו אקראי שמרחיב את הפלט פי 4. ונגדיר פונקציה $H\left(s_1, s_2\right) = G\left(s_1\right) \| G\left(s_2\right)$ אזי H יצרן פסואודו אקראי.

- **טכניקת הוכחה היברידית - $Hybrid\ Argument$:** עבור שתי התפלגויות כך שאחת היא תוצאה של הייצרן ואחת נבחרה רנדומלית.

1: האלגוריתם D מכניס התפלגות חדשה שמורכבת מצירוף שתי ההתפלגויות (חצי מכל אחת).

2: אנו מניחים שלאלגוריתם D יש יתרון לפחות ε להחין בין שתי ההתפלגויות. לכן או שיש לו יתרון של לפחות $\frac{\varepsilon}{2}$ להבחין בין ההתפלגות המקורית להתפלגות החדשה, או שיש לו יתרון של לפחות $\frac{\varepsilon}{2}$ להבחין הין ההתפלגות הרנדומלית להתפלגות החדשה.

3: נבנה מבחין A שונה עבור כל אחד מהמקרים, וכך הראנו מבחין A שיסתור את הפסואודו אקראיות של G.

כיצד נבנה את A: המבחין A מקבל קלט z ותפקידו להבחין הם הוא פלט של G או נגדם רנדומלית. A ישרשר את z לחצי הקלט המתאים וינסה להבחין. אם הוא יצליח אזי הוא סותר את הפסואודו אקראיות של G.

- **הגדרה - גישת אוראקל:** אלגוריתם A בעל גישת אוראקל לפונקציה, הוא אלגוריתם שיכול לשלוח לפונקציה קלט x ולקבל את ההצפנה שלו בכל זמן שיבחר.

- $CPA -$ **Chosen-Plaintext Attack :** ניתן ליריב A לקבל בכל נקודת זמן הצפנה של כל הודעה שירצה (גישה זאת נקראת **גישת אוראקל**). מכיוון שהיריב פולי' אזי הוא יכול לקבל מספר פולינומי של הודעות בלבד, בנוסף אם אלגוריתם ההצפנה הסתברותי אזי בכל פעם הוא יחזיר הודעה רנדומלית אחרת. נאפשר ליריב לעשות זאת בתחילת הניסוי - קבלת ההודעות המוצפנות, ובסוף הניסוי - כשהוא מחזיר את הביט שניחש.

נסמן ניסוי זה ב : $IND(n) = \begin{cases} 1, & \text{if } b' = b \\ 0, & \text{otherwise} \end{cases}$

- **מערכת בטוחה נגד CPA:** נאמר כי מערכת הצפנה Π בטוחה נגד CPA אם לכל יריב הסתברותי פולי' A כך שההסתברות של A לנצח בניסוי שווה ל $Pr\left[IND(n) = 1\right] \leq \frac{1}{2} + v(n)$.

הנחה: בשביל שמערכת תעמוד בניסוי היא צריכה להחזיר עבור הודעה m הצפנה שונה בכל פעם (אלגוריתם הצפנה הסתברותי ולא דטרמניסטי).

- **פונקציה פסואודו אקראית $PRFs$:** נסמן

$$\text{Func}_{n \to \ell} = \text{ set of all functions from } \{0,1\}^n \text{ to } \{0,1\}^\ell$$

פונקציה אקראית: היא פונקציה h הנדגמת באופן אחיד מתוך $\text{Func}_{n \to \ell}$.

פונקציה פסואודו אקראית: נבחר מפתח קצר k (מוגרל באופן אחיד מאוסף כל המחרוזות באוסף n ביטים, עבור n פרמטר הבטיחות), ונגריל פונקציה f מתוך $\text{Func}_{n \to \ell}$.

הרעיון: מבחין D לא יכול להבחין בין הפונקציה f שלנו, לפונקציה h שתוגרל באופן אחיד ורנדומלי, **אף על פי** שיש לו גישת אוראקל לפונקציה.

פורמלית: נאמר שפונקציה f היא פסאודו אקראית, אם לכל מבחין הסתברותי פולינומיאלי D:

$$\left|Pr\left[\mathcal{D}^{F_k(\cdot)}\left(1^n\right)=1\right]-Pr\left[\mathcal{D}^{h(\cdot)}\left(1^n\right)=1\right]\right|\leq v(n)$$

where $k \leftarrow \{0,1\}^n$ **and** $h \leftarrow$ **Func** $_{n\to\ell}$ **.**

עבור $F_k(\cdot)$ המסמן - יש ל D גישת אוראקל לפונקציה F בעלת מפתח k.

- **הערה:** לא כל פונקציה פסאודו אקראית היא גם יצרן פסאודו אקראי. אך אם קיימת פונקציה פסאודו אקראית כלשהי, אז קיים גם יצרן פסאודו אקראי.
  בנוסף יש דרך לבנות יצרן פסאודו אקראי מפונקציה פסאודו אקראית.

0.3.1 בניית מערכת הצפנה סימטרית בהינתן פונקציה פסאודו אקראית:

- **אלגוריתם יצירת המפתחות:** דוגם באופן אחיד מפתח באורך n עבור הפונקציה f (עבור n פרמטר הבטיחות).
- **אלגוריתם ההצפנה:** דוגם באופן אחיד ערך r באורך n וההצפנה היא $c=(r,F_k(r)\oplus m)$.
- **אלגוריתם הפענוח:** עבור $c=(r,s)$ האלגוריתם מחשב את $m=F_k(r)\oplus s$.
- **משפט:** אם f פסאודו אקראית אזי Π_f בטוחה כנגד CPA.
- **היוריסטיקת Block ciphers :** מימוש לא מוכח לפונקציה פסאודו אקראי. בעלת מפתח באורך n, לכל מפתח k - $F(k)$ היא פרמוטציה על אוסף כל המחרוזות.

$$F:\{0,1\}^n\times\{0,1\}^\ell\to\{0,1\}^\ell$$

בעזרת המפתח ניתן לחשב את הפרמוטציה ולהפוך אות ביעילות.

השוני המהותי הוא: שאנו מעוניינים בפרמטרים ספציפים של n,l ולא בפרמטר n שהולך וגדל.

נאמר שהמערכת בטוחה: אם לא ניתן לתקוף אותה ע"י ביצוע מס פעולות בזמן הקטן משמעותית מ 2^n.

- **הצפנת הודעות ארוכות עם** $Block\ ciphers$**:** נחלק את ההודעה לכמה בלוקים, נצפין כל בלוק בנפרד עם מפתח אחר ונשרשר אותם יחד.

0.4 הרצאה 5 - מערכת אימות הודעות:

0.4.1 מערכת אימות הודעות:

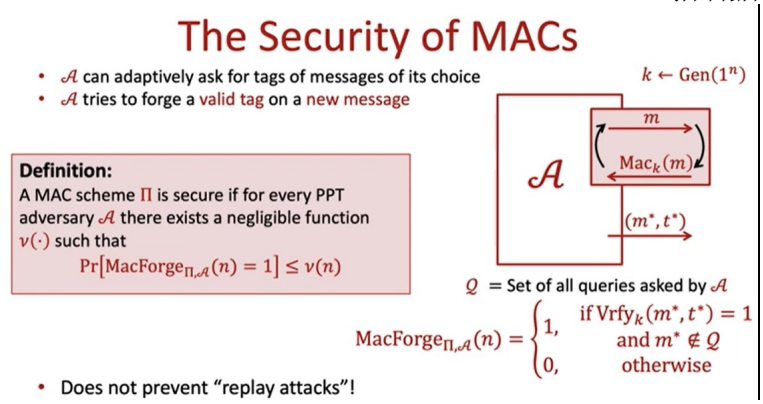
- **אנו רוצים לפתור את הבעיה הבאה:** שליחת הודעות בין שני אנשים, כך שצופה מהצד לא יוכל לשנות את תוכן ההודעה. נשים לב כי המטרה אינה להבטיח את סודיות המידע.

- מערכת אימות הודעות MAC: $Syntax : II = (Gen, Mac, Vrfy)$ כאשר:
  gen - **אלגוריתם יצירת מפתחות:** הסתברותי, מקבל מפתח בטיחות 1^nומחזיר מפתח k.
  mac - **אלגוריתם יצירת דרכי האימות:** הסתברותי, מקבל מפתח k והודעה m ומחזיר ערך אימות t.
  $vrfy$ - **אלגוריתם הווידוא:** דטרמניסטי, מקבל m, m וערך אימות פוטנציאלי t ומחזיר ביט b האם ערך האימות t אכן שייך להודעה m.

- **שימוש:** עם כל הודעה m נשלח t, ואם המפתח t עובר אימות אזי ההודעה בטוחה.

- **הגדרת נכונות:** לכל פרמטר בטיחות n ולכל מפתח k והודעה m אם נפיק בעזרת k ערך אימות t, אזי אלגוריתם הוידוא יקבל אותו בתור ערך חוקי להודעה m עם המפתח k, בהסתברות 1.

- **הגדרת בטיחות:** היריב מריץ את הניסוי $MacForge_{\Pi,\mathcal{A}}(n)$, ואנו רוצים למנוע מכל יריב הסתברותי ופולי' A, שאינו מכיר את המפתח k, ליצור ערכי אימות t שעוברים וידוא להודעה m.
  נרצה למנוע מימנו ליצור t עבור כל הודעה m שהוא יבחר.
  בנוסף הגישה שלו למערכת היא כך שיוכל לצפות בערכי אימות כרצונו (עבור מספר פולי' של הודעות) - גישת אוראקל לmac.
  הגדרה:



- **נשים לב כי בניסוי:** נדרוש שהיריב יצליח לאמת הודעות שלא הייתה לו גישת אוראקל אליהן.

- **הערה:** המערכת לא מונעת מהיריב להפיק t להודעה m בעזרת גישת אוראקל ולשלוח אותה עם t. לכן נשלח עם ההודעה שעה ותאריך.

- **משפט:** אם F פונקציה פסאודו אקראית ויצרנו בעזרתה מערכת אימות אזי מערכת האימות בטוחה. עבור $t = F_k(m)$

- **אימות הודעות מאורך משתנה:** נחלק הודעה m ל d בלוקים, ונקח בחשבון את מספר וסדר הבלוקים.
  נקבל את ערך האימות באופן הבא: נהפוך את האלגוריתם mac להסתברותי עם הערך r, מפתח ההצפנה $t_i = \widehat{Mac_k}(r, d, i, m_i)$ כך שכל ערך אימות t_i הוא עבור הודעה שהיא הששרשור של r עם מספר הבלוקים d, של מספר הבלוק i ושל הבלוק ה m_i.
  החסרון: ערכי האימות מאוד ארוכים - לכל הודעה d ערכי אימות.

- **פתרון שני - $CBC-Mac$:** נרצה אלגוריתם כך שמספר ערכי האימות לא תלוי באורך ההודעה.
נחלק את ההודעה ל d בלוקים בגודל שווה, וניצור ערכי אימות $t_0...t_d$ כך ש $t_0=0^n$ (או כל ערך קבוע אחר), לאחר מכן נחשב עם פונקציה פסאודו אקראית $t_i=F_k\left(t_{i-1}\bigoplus m_i\right)$, ו t_d הוא ערך האימות של ההודעה.
הערה: אם היריב יכול לבקש בניסוי להצפין הודעות כך שאורך הבלוקים אינו קבוע אזי המערכת אינה בטוחה.

0.4.2 פונקציות האש ללא התנגשויות:

- **פתרון שלישי - גישת האש:** נכווץ את ההודעה עם פונקציה h שתחזיר לנו $h(m)$ לאחר מכן נפיק עבורה ערך אימות.
החסרון: צריך לשים לב שהודעות שונות לא ימופו לאותם ערכי אימות.

- **פונקציות האש העמידות בפני מציאת התנגשויות:** פונקציות הנינתנות לחישוב בזמן פולי'. והן מקיימות את התכונות הבאות: 1 - כיווץ הודעות. 2 - זוג של קלטים שונים ימופו לערכים שונים בהסתברות גבוהה.

- **הגדרה:** הפונקציה מוגדרת באופן הבא - $\Phi=(Gen,H)$
Gen - אלגוריתם יצירת מפתחות: מקבל פרמטר בטיחות ומחזיר מפתח.
H - אלגוריתם חישוב פונקצית האש: מקבל מפתח s וקלא x ומחזיר פלט $h_s(x)$ שאורכו $l(n)$.

- **בטיחות:** נדרוש כי לכל יריב הסתברותי פולי' A יש הסתברות זניחה במציאת שני ערכי קלט שונים x,x' הממופים לאותו ערך פלט.
אנו ניתן ליריב את המפתח s בצורה מפורשת.
נאמר כי הפונקציה בטוחה אם היא מקיימת:

$$Pr\left[\text{ HashColl }_{\Phi,\mathcal{A}}(n)=1\right]\leq v(n)$$

- **הניסוי** $HashColl_{\Phi,\mathcal{A}}(n)$**:** ניתן ליריב את **המפתח** s, והוא מנצח אם הוא מוצא שני פלטים x,x' כך ש $h(x)=h(x')$.

- **מתקפת יום ההולדת:** עבור קלט באורך l, נדגום $q=\mathcal{O}\left(2^{\ell/2}\right)$, קלטים באקראי ונחשב את ערך הפונקציה h עבור כ"א מהם. ההסתברות להתנגשות לא טרוויאלית גבוהה מאד.
כיצד נבטיח בטיחות: נחזיר פלטים באורך גדול מ 128 ביטים.

- **מערכת אימות עם האש תוגדר כך:**

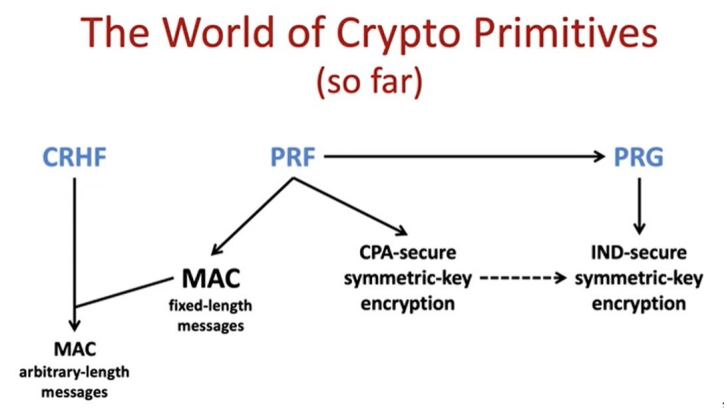
Hash-and-Authenticate

Let $\widehat{\Pi} = (\widehat{\mathrm{Gen}}, \widehat{\mathrm{Mac}}, \widehat{\mathrm{Vrfy}})$ be a fixed-length MAC, and let $\Phi = (\mathrm{Gen}_H, H)$ be a keyed hash function. Consider the following MAC scheme $\Pi = (\mathrm{Gen}, \mathrm{Mac}, \mathrm{Vrfy})$ for arbitrary-length messages:

- **Key generation:** On input 1^n sample $k \leftarrow \widehat{\mathrm{Gen}}(1^n)$ and $s \leftarrow \mathrm{Gen}_H(1^n)$, and then output (k, s).
- **Tag generation:** On input (k, s) and $m \in \{0,1\}^*$ output $t = \widehat{\mathrm{Mac}}_k(H_s(m))$.
- **Verification:** On input (k, s), $m \in \{0,1\}^*$, and $t \in \{0,1\}^*$, output $\widehat{\mathrm{Vrfy}}_k(H_s(m), t)$.

0.4.3 מערכת אימות הודעות והצפנה - $Authenticated\ Encryption$:

- **טענה:** מערכת אימות הודעות לא בהכרח מבטיחה הצפנה, ולהיפך.
- **מערכת אימות הודעות והצפנה -** $Authenticated\ Encryption$: נצפין את ההודעה m ולאחר שנקבל הודעה מוצפנת c נחשב לה מפתח אימות t.
- **סיכום, של החומר עד עכשיו:**



חלק II

אבטחת תכנה:

0.5 הרצאה 6 - אבטחת תכנה:

0.5.1 ניהול הזכרון המערכת ההפעלה:

- אנו נדבר על מערכת לינוקס 32 ביט.
- **גודל הזכרון:** הוא $4G$, 2^{32}.
- **כתובות וירטואליות:** כל תהליך שרץ מקבל הקצאה של זכרון ממערכת ההפעלה. מערכת ההפעלה אחראית על חלוקת הזכרון אך היא מדמה זכרון וירטואלי כך שמבחינת התהליך כל הזכרון שלו בלבד.
- **מה נשמר בזכרון:** בכתובות הראשונות נשמר הקוד שמריץ את התהליך, מעליו נשמור מידע סטטי. נשים לב כי כל המידע הזה ידוע כבר בשלב הקומפילציה.
  בכתובות הזכרון האחרונות יישמר מידע על שורת הפקודה והתהליך. מתחתיהם ישמרו המחסנית והערימה.
- **המחסנית** $Stack$**:** גדלה מכיוון הכתובות הגבוהות לממוכות, בכל פעם שהיא מתעדכנת המצביע לראש המחסנית - SP זז איתה. בסוף ריצת פונקציה המצביע חוזר למקומו.
- **הערימה** $Heap$**:** גדלה מכיוון הכתובות הנמוכות לכתובות הגבוהות, נשמר בה מידע דינמי כגון מערכים שמוקצים עם $malloc()$.
- $func\ frame$**:** לכל פונקציה יש מקום מוגדר במחסנית ונשמר אליו מצביע ebp, כך הפונקציה יודעת כיצד לגשת לכל ארגומנט או משתנה לפי $ebp + size$.
- **מה קורא כשפונקציה נקראת:**
  1: הפונקציה הקוראת מכניסה למחסנית את הארגומנטים של הפונקציה הנקראת **בסדר הפוך**.
  2: הפונקציה הקוראת שומרת את כתובת החזרה - הכתובת של השורה הבאה בפונקציה הקוראת ברגיסטר eip.
  3: לאחר מכן הפונקציה הקוראת תאתחל את הרגיסטר eip להצביע על הכתובת של הפונקציה הנקראת.
  4: אח"כ נשמור את המצביע ebp (מצביע ל $func\ frame$) של הפונקציה הקוראת.
  5: לאחר מכן הפונקציה הנקראת מקצה מקום במחסנית למשתנים הלוקאלים שלה. לאחר מכן הפונקציה מתחילה לרוץ.
- **מה קורה בפקודת** $return$**:** הפונקציה הנקראת מערכנת את SP לחזור למקומו ההתחלתי, ובנוסף היא משחררת את ebp וגורמת לו להצביע על השורה הבאה בפונקציה הקוראת (הרגיסטר eip). לאחר מכן הפונקציה הקוראת משחררת את הארגומנטים של הפונקציה הנקראת.

0.5.2 מתקפות באפר אוברפלואו:

- באג הניתן לניצול שמשפיע על קוד בשפות $low\ level$ המנהלות את הזכרון.
- **הגדרה - באפר:** מערך - איזור רציף בזכרון, הנשמר לתוך משתנה.
- **הגדרה - אוברפלואו:** גישה למערך מחוץ לגבולות הזיכרון שהוקצו לו ע"י המתכנת.
- **דוגמה:** אם נשמור משתנה לוקאלי שגודלו גדול יותר מהגודל שהקצנו לו הוא יכול לדרוס משתנים אחרים ואת ערך ההחזרה של הפונקציה.
- **ביצוע המתקפה:** יכול להיעשות ע"י פונקציות כדוגמת $strcpy()$ שמעתיקות סטרינג מסויים למיקום חדש שהוגדר לו, הפונקציה מעתיקה עד התו $null$ הראשון.

0.5.3 מתקפות $Code\ injection$:

- **הרעיון:** מתבסס על מתקפות באפר אוברפלואו שדורסות את תוכן המחסנית, והכנסת קוד חדש.
- **שלב ראשון:** הכנסת הקוד המזיק לזכרון. נעשה זאת ע"י שינוי רגיסטר ה eip שמצביע על השורה הבאה בקוד, שיצביע על הקוד המזיק
- **הקוד המזיק צריך לקיים את התנאים הבאים:**
  1: יהיה כתוב בשפת מכונה.
  2: יכיל כתובות זכרון מלאות משום שהוא לא עובר קומפילציה.
  3: בנוסף נדאג שלא יופיע באמצע הקוד בית שכולו אפסים, משום שפונקציות כדוגמת $strcpy()$ מעתיקות עד הבית שערכו $null$ ולכן המתקפה תעצר באמצע.
- **איזה קוד נזריק:** נזריק $shell\ code$ שנותן גישת CMD לתהליך ומשם התוקף יוכל להמשיך לבד.
  המימוש בפועל צריך להיות בשפת C ולעבור המרה לפי התנאים לעיל.
- **איך נגרום לקוד לרוץ:** נגרום לרגיסטר eip להצביע על הקוד המוזרק, ע"י שינוי כתובת החזרה של הפונקציה השמורה במחסנית (עם באפר אוברפלואו) לכתובת של הקוד המוזרק.
- **הגדרה - $noOp$:** שורת קוד שלא עושה דבר אלא ממשיכה לשורה שאחריה.
- **כיצד ננחש את כתובת הקוד המוזרק:** אנו יודעים את כתובת הבסיס של המחסנית, בנוסף אנו יודעים כיצד המחסנית אמורה לגדול וכך להקטין את טווח הכתובות. בנוסף אנו לא צריכים לקלוע לכתובת של תחילת הקוד המוזרק, אלא יכולים לרפד אותו בשורות של $noOp$ כך שנגיע לשורת הקוד המתאימה.

0.5.4 מתקפות על הערימה:

- **מתקפת באפר אוברפלואו על הערימה:** אם אנו מקצים בקוד מערך דינמי בגודל MAX, ניתן לבצע מתקפת אוברפלואו על המערך ולגרום לו לגדול מעבר לגודל המקסימלי ע"י $strcpy$. וכך לדרוס אובייקט אחר שהוקצה על הערימה.
- **מתקפה עם פונקציה וירטואלית:** כשאנו מגדירים פונקציה וירטואלית ב $c++$ הקומפיילר מוסיף שדה חבוי הנקרא $vtable$ המצביע למערך של מצביעים לפונקציות. ניתן לנצל אוברפלואו של שדה גלוי בכדי לדרוס את המצביע לשדה $vtable$, ולעשות לו השמה עם מצביע אחר שמביע לפונקציות שהגדרנו מראש.
- **מתקפה על ניהול הזכון:** הערימה מכילה הקצאות זכרון של מידע חבוי העוזר בניהול הקצאת הזכרון, אם נדרוס את המידע הזה אנו נפגע בניהול הזכרון של המערכת.
- **מתקפת אוברפלואו על מספרים:** ניתן לעשות מניפולציה על פונקציה $malloc$ כך שנקצה לה מיקום בגודל 0, במקרה זה הא מחזירה מצביע לכתובת במחסנית אותה ניתן לדרוס.
- **מתקפת** $Read\ Overflow$**:** קריאה מהזכרון ממקום שאין לתוקף גישה אליו גם יכולה לגרום לנזק, לדוגמה קריאת ססמאות או מפתחות הצפנה.
  המתקפה יכולה להתבצע כאשר אנו מבקשים מהמשתמש להכניס אורך של קלט, וקלט כלשהו בכדי להדפיס אותו, והמשתמש מצהיר על אורך גדול יותר מהאורך האמיתי של הקלט כך שהמערכת תחזיר מידע רגיש מהערימה.
- **מתקפת שימוש במצביע ששוחרר:** כאשר אנו מקצים זכרון ומשחררים אותו המידע נשמר אך המצביע משוחרר לשימוש חוזר. אם התוקף יבקש הקצאת זכרון חדשה יכול להיווצר מצב שהוא יקבל את הכתובת של המצביע ששוחרר, לאחר מכן הוא יוכל להשתלט על המצביע החדש (באמצעות באפר אוברפלואו), ולאחסן בו מידע זדוני. במידה וניגש בטעות למצביע ששוחרר יתכן ונריץ את הקוד שהתוקף שתל.

0.5.5 מתקפות עם פורמט הדפסה:

- **הדפסה בשפת** C**:** כשאנו רוצים להשתמש בפונקציה $printf()$ אנו צריכים להצהיר על סוג המשתנה אותו אנו רוצים להדפיס עם מציין פורמט .
- **המתקפה:** אם נרצה להדפיס באפר ולא נצהיר מהו מציין הפורמט, תוקף יוכל להזין לתוך הבאפר מחרוזת המכילה מציני פורמט וכך הפונקציה תחכה לארגומנטים הנוספים אותם צריך להדפיס. כתוצאה מכך הפונקציה תדפיס מידע שנשמר במחסנית מחוץ ל $func\ frame$.
- מתקפה נוספת: הפונקציה $printf()$ מתעלמת מרווחים, לכן אם נבקש להדפיס $1\%\ david$ הפונקציה תתייחס ל $\%d$ כאל מציין פורמט ותדפיס מידע מעבר ל $func\ frame$.

0.6 הרצאה 7 - שיטות הגנה:

0.6.1 בטיחות זכרון - $Memory\ Safety$:

- **הקצאות:** תכנית בטוחה צריכה להקצות זכרון עם $maloc$, ולעשות שימוש מצביעים כדי לגשת למקום שהוקצה ולא לחרוג ממנו. תכנית בטוחה תשתמש בשני מרכיבים - בטיחות מרחבית ובטיחות זמנית.

- **בטיחות מרחבית:** מטרתה לוודא שתכנית לא תעשה שימוש במצביע בכדי לגשת לזכרון שלא הוקצה לה (מתקפת באפר אוברפלואו). נעשה זאת ע"י שדרוג מצביעים למבנים בעלי שלשה שדות:
  p: המצביע.
  b: כתובת הבסיס (הכתובת השמאלית) ממנה יכול המצביע להתחיל לגשת לזכרון.
  e: כתובת הסיום - עד להיכן מותר למצביע לגשת.
  ונעשה שימוש במצביע רק אם $b \leq p \leq e - sizeof(type(p))$.

- **בטיחות זמנית:** משלימה את הבטיחות המרחבית. באה לתת פתרון עבור מקרים בהם איזור מסויים בזכרון חוקי רק בזמן מסויים שהתכנית רצה (לדוגמה - שימוש במצביע שכבר שוחרר, או מצביע שלא אותחל). אזורי זכרון אלו יקראו אזורי זכרון מוגדרים.

- **הערה:** שפות $low\ level$ לא מספקות בטיחות זכרון.

0.6.2 מנגנוני אבטחה אוטומטים על הזכרון:

- **מניעת מתקפות הזרקת קוד:** ניזכר כי ההתקפה מתרחשת ע"י טעינת קוד והרצתו. המטרה היא להקשות על מתקפות אלו.

- **שימוש בערכי קנרית - מניעת הזרקת קוד עויין:** נעשה שימוש במנגנון זה בכדי לזהות מתקפות באפר אוברפלואו. מתקפה זו תוקפת את מסגרת הפונקציה הקוראת ע"י שינוי ערך ההחזרה של הפונקציה. נטפל בה באופן הבא:
  ניצור משתנה $canary$ ונכניס אותו בין מצביע מסגרת המחסנית של הפונקציה הקוראת - , לבין המשתנה המקומי הראשון. באופן זה אם תהיה מתקפת באפר שדורסת את המשתנה הלוקאלי ואת מצביע המחסנית, היא תדרוס גם את ערך ה $canary$. לכן נשמור את הערך הזה בצד ונבדוק לפני החזרה מהפונקציה שהערך לא השתנה, אם כן - נמשיך בריצה. אחרת - נעצור את ריצת הפונקציה.

- **הערות לערך קנרית:** נאתחל אותו בזמן ריצה ולא בזמן קומפלציה בכדי שלא יהיה ידוע לתוקף. בנוסף נבחר אותו באופן אקראי מספיק. וגם נשמור אותו בצורה בטוחה בזכרון בכדי שנוכל להשוות אותו.

- **מניעת ריצת קוד עויין:** אם התוקף הצליח להזריק קוד עויין נרצה למנוע ממנו את האופציה להריץ את הקוד. נעשה זאת באופן הבא:
  קוד התכנית נמצא באיזורים הנמוכים בזכרון - איזור הטקסט, בעוד הזרקת קוד מתרחקת בהיפ או במחסנית. לכן כדי למנוע את ריצת הקוד העויין נוכל להגדיר לרגיסטר eip שלא יצביע לזכרון החורג מגבולות הטקסט. כך שאם תוקף יגדיר לרגיסטר זה להצביע להיפ או למחסנית נחסום את ריצת התכנית.

- **מתקפת** $Return\ to\ libc$**:** המתקפה לא מזריקה קוד למחסנית או להיפ אלא עשה שימוש בקוד שכבר נמצא באיזור הטקסט.
  בכל פעם שנקמפל תכנית הקומפיילר יכניס לאיזור הטקסט פונקציות של $libc$ - אוסף של פונקציות בסיסיות בשפת c, תוקף שמכיר את הספריה ואת סידורה בזכרון יכול בעזרת מתקפת באפר אוברפלואו לדרוס ערך חזרה של כתובת של פונקציה מ $libc$.
  התגוננות ממתקפה זו: $ASLR$ מסיט באופן רנדומלי בזכרון את הפונקציות של $libc$ כך שהתוקף לא יוכל לדעת היכן

הפונקציה נמצאת בזכרון וכיצד לדרוס אותה. (לא יעיל במיוחד עבור מערכת 32 ביט)
מנגנון הגנה נוסף: נמנע מלכלול את כל הספריה $libc$ אלא רק את הפונקציות הרלוונטיות לקוד שלנו.

0.6.3 מנגנון ההגנה $control\ flow\ inte-CFI$:

- **הרעיון הבסיסי:** לצפות בהתנהגות התכנית בזמן ריצה ובדיקה האם היא רצה כנדרש, מבחינת קריאה וחזרה מפונקציות. יתפוס הצבעות של רגיסטר eip מחוץ לאיזור הטקסט, והזרקת קוד עויין בנוסף היא יעילה נגד מתקפת $libc$.

- **נגדיר התנהגות ראויה:** נגדיר גרף $control\ flow\ graph-CFG$ כך שריצה תקינה של התכנית אמורה להיות מסלול חוקי בגרף בלבד.
ייצירת הגרף: הקדקודים ייצגו את הפונקציות בתכנית, והצלעות מתארות את הקריאות של הפונקציות וחזרה מפונקציות, בנוסף נתאים את קצוות הקשתות לשורות הקוד בהן התבצע הקריאות לפונקציה.
תהליך בניית הגרף מתבצע בזמן קומפלציה. אח"כ בזמן הריצה נעקוב אחרי ריצת התכנית ונבדוק התאמה לגרף (למעשה נצטרך לעקוב רק אחרי קריאות לא ישירות - קריאות בעזרת מצביע לפונקציה וחזרות מפונקציה), כך במקרה של מתקפה נזהה זאת.

- **כיצד נזהה בזמן הריצה חריגות מההתנהגות התקינה:** נעשה זאת באמצעות IRM, נבצע טרנפורמציה של הקוד בזמן קומפלציה כך שיעקבו אחרי ריצת התכנית. נוסיף שורות קוד - עבור כל קריאה לא ישירה נכלול תוית (אקראית) בקוד הפונקציה הקוראת והנקראת:
הפונקציה הקוראת: לפני שנקפוץ לפונקציה הנקראת נבדוק התאמה בין התוויות (מותר לה להמשיך רק לפונקציה שמסומנת בתוית שהיא שומרת), רק אם יש התאמה - נמשיך בריצת התכנית.
הפונקציה הנקרא: תבדוק התאמה של התוית לפני שהיא חוזרת לפונקציה.

0.6.4 תכנון בטוח בשפות $low\ level$:

- **כלל ראשון:** כאשר אנו מבקשים קלט מהמשתמש, נוודא כי הקלט תואם את ההנחות שלנו לגביו. לדוגמה - אורך המחרוזת אכן תואם לאורך עליו הצהיר המשתמש, אחרת הוא יוכל להזין אורך גדול יותר ולמשוך מידע חסוי.

- **כלל שני:** נעשה שימוש בפונקציות בטוחות כאשר אנו משתמשים בסטרינגים.
פונקציות בטוחות: $strlcpy, strlcat$, פונקציות אלו מקבלו כקלט גם את אורך הסטרינג וכך נמנעות בבאפר אוברפלואו.

- **כלל שלישי:** נזכור שסטרינגים נגמרים בתו $\backslash 0$ ונשמור לו מקום.

- **כלל רביעי:** נעקוב אחרי המצביעים שלנו ואריתמטיקה של מצביעים, בנוסף נזכור ש $sizeof$ מחזירה לנו את הגודל של המצביע השמור.

- **כלל חמישי:** כשאנו משחררים מצביע הוא ממשיך להצבע לכתובת ששוחררה, לכן תמיש כשנשחרר מצביע נגדיר אותו להיות $null$.

0.7 הרצאה 9 - אבטחה באינטרנט:

- **תקשורת:** כשאנו מתקשרים על גבי האינטרנט התקשורת מתבצעת בעזרת שרתים. הלקוחות מבקשים בקשות מהשרת ע"י כתובת IP וארגומנטים הם ניגשים אליו ומבקשים ממנו לבצע פעולות.

- **דפים סטטים ודינאמים:** דף סטטי הוא דף שנשלח בכל בקשה לשרת ואינו משתנה - תוכן $html$. דף דינמי לעומת זאת הינו דף שיכול להתשנות מגישה לגישה - דף php.

- **הפקודות GET ו $POST$:** הפקודה GET היא פקודת בקשת מידע מהשרת. ואילו הפקודה $POST$ היא פקודת ביצוע פעולה.

- **תגובת השרת לפקודות $HTTP$:** השרת יחזיר את הארגומנטים הבאים
  - קוד סטטוס האם הבקשה הצליחה, אם לא הוא יציין היכן נכשלה - צד שרת או צד לקוח.
  - $headers$ מספקים מידע על סוג השרת ואופן פעולתו.
  - הדפדפן שהשרת יציג.
  - עוגיות.

0.7.1 מתקפת הזרקת SQL:

- **הרעיון:** מתקפה המתבצעת על מסד הנתונים של השרת, היא מתבססת על כך שהשרת מדביק לתוך שאילתה את הקלט שהמשתמש מזין.

- **מתקפת הזרקת קוד שאילתה:** כאשר אנו מזינים שם משתמש וסיסמה לאתר, האתר שולח שאילתת SQL למסד הנתונים שבודקת האם מופיעה שורה בה שם המשתמש והסמה זהים למה שהמשתמש הכניס. תוקף יכול לשלוח את השורה הבאה בתור שם משתמש $'name'\ OR\ 1 = 1);--$ כך המסד יחזיר $true$ עבור כל השורות והתוקף יצליח להיכנס ללא סיסמה מתאימה.

- **מנגנוני הגנה:** המנגנון הראשון הוא בדיקת הקלט ומחיקת התווים הבאים: $;, --, '$ או שינוי שלהם.

- **מנגנון הגנה נוסף:** נבדוק כי הקלט לא חשוד, ואם הוא חשוד לא נקבל אותו.

- **שימוש בהצהרות מוכנות:** נפריד בין הקוד ובין המידע, ע"י כך שנגדיר למסד הנתונים מהו החלק של הקוד ומהו החלק שבו ייכנס מידע מהמשתמש. בנוסף נגדיר מהוו סוג המשתנה שאמור להתקבל מהמשתמש. כך כשננסה להריץ את השאילתה המסד ידע להפריד בין קוד לבין קלט משתמש, ולא יריץ את הקלט כקוד.

0.7.2 מנגנוני שמירת מידע ומתקפות:

- **כיצד המידע של הלקוח נשמר:** בכל שלב בתקשורת בין השרת ללקוח, השרת שולח ללקוח את המידע שהלקוח ביצע, ובפעם הבאה שהלקוח יתקשר עם השרת הוא ישלח את המידע הזה. שמירת המידע מתבצעת באמצעות שדות חבויים ועוגיות.

- **שדות חבויים:** שדה שיופיע בו מידע שימושי לשרת, מידע שמוסתר מהמשתמש. לדוגמה בביצוע קניה באינטרנט השרת ישמור בשדה חבוי את סכום החיוב. שדה זה ישלח ללקוח, ובשלב הבא של התקשורת הוא יזכיר לשרת היכן הם היו ומה סכום החיוב.

- **מתקפה על שדות חבויים:** התוקף יכול לשנות את סכום ההזמנה בשדה החבוי ובהתחברות הבאה לשלוח סכום לחיוב נמוך יותר.

- **מנגנון הגנה - סשן** ID**:** השרת ייצר ID שהלקוח יחזיר לו, והמספר הזה ישלח את השרת לסיכום ההזמנה הקודמת, כך שהתוקף לא יוכל לשנות בעצמו את סכום ההזמנה. בנוסף התוקף לא יוכל לבחור ID אקראי כי השרת בוחר אותם בהתפלגות שקשה לנחש.
החסרון: אם נצא מהאתר ה ID ימחק ולא נוכל לחזור למקום שהפסקנו.

- **עוגיות:** באות לפתור את הבעיה של השדות החבויים. השרת ייוצר מצב וישלח אותו ללקוח, הלקוח יאכסן אותו ושילח אותו לשרת בכל סשן עתידי.
מכיל שני רכיבים: שדה ראשי $referer$ שהשרת קובע עבורו ערך. ואוסף של פרמטרים המתאר את אופן השימוש בעוגיה - תאריך תגובה, שם דומיין או משאב ספציפי בדומיין.
חסרונות: ניתן לעקוב אחר משתמש ולדעת היכן הוא גלש, זאת פגיעה חמורה בפרטיות.

- **רשתות פרסום ועוגיות:** כשאנו גולשים באתר שיש בו שטח פרסום, השרת שולח אותנו אל שרת הפרסום שאמור להציג לנו את המודעה. שרת זה שומר את כל האתרים בהם גלשנו והגענו אליו דרכם, כך הוא יכול לעקוב אחרינו ולתרגט לנו מודעות ספציפיות. כדי להימנע מכך נחסום קבלת עוגיות מצד שלישי.

- **מתקפת סשן** $hijacking$**:** כשאנו מתחברים לשרת מסויים הוא שומר בעוגיה את פרטי ההתחברות שלנו לזמן מסויים כדי שלא נצטרך לבצע כניסה בכל פעם מחדש. מתקפה זו מנצלת את השימוש בעוגיות כדי לזהות משתמש שהתחבר לשרת בעבר, ולהזדהות מול השרת בעזרת העוגיה של המשתמש הלגיטימי.
גניבת סשן של עוגיה: מתרחשת בכמה אופנים - פריצה לשרת. ניחוש של הסשן ID. האזנה לתקשורת בין השרת והמשתמש. מתקפה על מנגנון ה DNS (המתרגמת כתובות URL לכתובות IP) - התוקף מתחזה לשרת, וכך המשתמש שולח לתוקף את העוגיה במקום לשרת.
מניעה: נגדיר תאריך תפוגה ונחדש אותן בתדירות גבוהה.

- **מתקפת** $CSRF$**:** מנצלת את העובדה ששרת אמין סומך על מידע שנשלח אליו מהמשתמש. התוקף מתחזה לאתר המבוקש, כך שהמשתמש מתחבר אל האתר המזוייף. התוקף מתחבר עם העוגיה לאתר המקורי.
כיצד התוקף מפנה לכתובת מזוייפת: התוקף מציג באתר המזוייף תמונה המפנה לאתר המקורי, השרת ינסה להשיג את התמונה ולכן יתחבר לאתר המקורי. מכיוון שהמשתמש הוא משתמש חוקי באתר המקורי, אזי החיבור יתבצע עם העוגיה והפעולות שהתוקף מבצע יאושרו. מכיוון שמבחינת השרת הפגיע החיבור בוצע ע"י משתמש חוקי.

- **מנגנון הגנה ראשון:** נשלח שדה $referer$ כך השרת הפגיע יוכל לבדוק אם המשתמש מתחבר או תוקף שמתחזה אליו.
חסרון: מנגנון הז עוקב אחר הלקוח ולכן לא תמיד נרצה להשתמש בו.

- **מנגנון הגנה שני -** $secretized\ links$**:** המטרה שלו היא לקשור את הכתובת אליה המשתמש ניגש, אל הסשן קוקי שהמשתמש שולח. נעשה זאת ע"י שדה חבוי שמופיע בסשן קוקי בעזרת סטרינג אקראי או פרמטר כלשהו.

0.7.3 ג'אווה סקריפט:

- **הרעיון:** השרת שולח לדפדפן קוד, והדפדפן מריץ אותו בעצמו. כך ניתן לשפר את חווית המשתמש. לסקריפטים אלו יש יכולות לעקוב אחר לחיצות עכבר ומקלדת ועוד. מסיבה זו דפדפנים מאובטחים כדוגמת בנקים צריכים להגביל את יכולות הסקריפטים הרצים על הדפדפן, רק למידע הקשור לדומיין ממנו הסקריפט הגיע - שיטה זו הקראת SOP.

- **מתקפת** XSS**:** מנצלת את העובדה שהמשתמש סומך על מידע שנשלח אליו משרת אמין אך פגיע. המתקפה מבוססת על שליחת סקריפט זדוני למשתמש, כך שהדפדפן יחשוב שהסקריפט הגיע ממקור אחר (לדוגמה שרת הבנק) ולהריץ אותו (למעשה המתקפה מתגברת על SOP). המתקפה מבוצעת כך שהסקריפט אכן יישלח לדפדפן ממקור אחר, ללא כל התחזות. יש שתי שיטות לבצע מתקפה זו:

- **מתקפת** XSS **מאוחסן:** התוקף מאחסן את הסקריפט הזדוני בשרת של המקור האחר. כך שהשרת ישלח בטעות את הסקריפט לדפדפן, הדפדפן יאפשר לו גישה ויריץ אותו.
אחסון הסקריפט: פלטפורמות המאפשרות העלאת תוכן חשופות יותר למתקפה זו, משום שמשתמש יכול להעלות סקריפט זדוני לדף שלו, כך שכל מי שיכנס אליו יריץ את הסקריפט משום שהוא מוגדר כבטוח.

- **מתקפת** XSS **משוקף:** בניגוד למתקפת סקריפט מאוחסן, במתקפה זו התוקף יגרום **למשתמש** לשלוח את הסקריפט לשרת הפגיע, כך שהשרת יחזיר אותו למשתמש. לאחר מכן הסקריפט יוגדר כבטוח משום שהוא נשלח ע"י השרת.
שתילת הסקריפט: נעשית כשהמותקף גולש באתר לא מאובטח, ולוחץ על קישור שמפנה אותו לשרת פגיע.
כיצד לגרום לשרת להחזיר סקריפט שהלקוח שלח אליו: לדוגמה - חיפוש ברשת, התוקף ישלח את המשתמש לשרת חיפוש כשבשורת החיפוש מופיע סקריפט זדוני, השרת לא ימצא את הסקריפט ולכן יחזיר אותו למשתמש.
מניעה: נסנן סקריפטים שלא נשלחו ע"י השרת עצמו. אך פתרון זה מגביל את השרת מלהריץ סקריפטים תקולים שאינם זדוניים.

0.8 הרצאה 10 - אנליזה סטטית של קוד:

- **בדיקה אוטומטית של קוד מבחינת אבטחה**: נשתמש בה לאחר מימוש מערכת, או בדיקה למערכת חדשה. נרצה שהמערכת תכוון אותנו לקטע הקוד הבעייתי.

- **בדיקת נכונות של תכניות:** בדיקה שאומרת לנו האם התכנית עושה מה שהיא אמורה לעשות. ניתן לעשות זאת בעזרת טסטים, אך עבור מערכות קוד מסובכות יהיה קשה לעקוב והרצת התכנית תהיה יקרה וארוכה.
גישה נוספת היא $auditing$ - מומחה עובר על הקוד ובודק נכונות של הקוד.

- **גישה ראשונה - אנליזה סטטית:** מתבצעת ללא הרצת הקוד ודומה יותר ל $auditing$ בבדיקת נכונות.
היתרונות: היא מנתחת בפעם אחת הרבה ריצות של התכנית. ניתן להריץ אותה גם על קוד לא שלם.
חסרונות: יכולה לנתח רק תכונות בטיחות מוגבלות ועשויה להיות יקרה מבחינת זמן ריצה. עשויה לפספס מפגעי אבטחה או להעיר ללא צורך על מפגע.

- **הערה:** אנליזה סטטית מושלמת לא קיימת והיא שקולה לבעיית העצירה. לכן נרצה לקרב ולמצוא אנליזה סטטית מועילה - החסרון הוא שהיא יכולה לא לעצור.

- **עיצוב מערכת מועילה:**
מערכת מאוזנת מבחינת דיוק - לא מפספסת פירצות, ולא מדווחת דיווחי שווא.
זמני ריצה - נרצה זמן ריצה נמוך עבור קטעי קוד גדולים.
מעקב - נוכל לעקוב אחר קטעי הקוד הבעייתיים ולהבין את הבעיתיות.
הערה: המערכת תלויה בקוד, וכיצד הוא כתוב.

0.8.1 אנליזה סטטית - $flow\ analisis$:

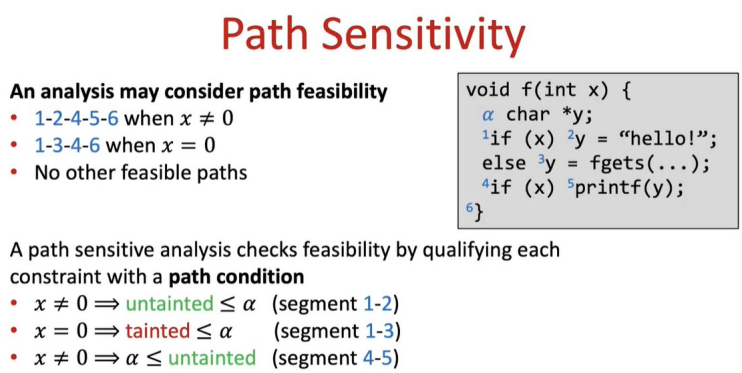
- $flow\ analisis$: נעקוב אחר האופן שבו ערכים מגיעים לזכרון, וכיצד הם יכולים להשפיע על ערכים אחרים.
- **הגדרה - ערכים** $tainted$ **ו** $unteinted$: ערכים שהמשתמש יכול לשלוט עליהם ולהגדיר אותם כרצונו יוגדרו - $teinted$, כלומר חשודים. ואילו ערכים שאין למשתמש שליטה אליהם יוגדרו - $unteinted$ כלומר אמינים.
- **הערה:** מחרוזות קבועות יוגדרו כאמינות.
- **רעיון הערכים:** לכל ערך בתכנית כדוגמת ארגומנט של פונקציה או ערך החזרה של פונקציה, יש סוג חשוד או אמין.
בתחילה נגדיר לכל ארגומנט של פונקציה וערך החזרה בפונקציה האם הוא אמין או חשוד.
כיצד נסווג ערכים: נגדיר ערך כחשוד אם הוא **עשוי** להיות בשליטת תוקף.
ואמין אם הוא **חייב שלא** להיות בשליטת התוקף, לדוגמה - ארגומנט של פונקציה $printf$.
- **המטרה:** לגלות האם עבור כל הקלטים האפשריים לתכנית, לא יתבצע שימוש בערכים חשודים כאשר התכנית מצפה לערך אמין.
- **הגדרה - זרימה: זרימה לא חוקית** - זרימה בה יש זרימה של ערכים חשודים לערכים אמינים, כלומר פונקציה שצריכה לקבל ערך אמין מקבלת ערך חשוד, מתקיים $tainted \leq untainted$ ולכן הזרימה לא חוקית.
זרימה חוקית - כל זרימה שאינה זרימה לא חוקית, נשים לב כי ערך חשוד יכול לקבל כל סוג משתנה (כולל משתנה אמין).
- **ביצוע:** נגדיר לכל ערך בתכנית את הסוג שלו.
- נקצה משתנה עבור כל ערך בתכנית שאנו צריכים לסווג.
- נעבור על כל שורות הקוד ונתרגם כל שורה לאילוץ מסויים על המשנים שהגדרנו, לדוגמה אם y זורם ל x ($x = y$)
- נוסיף את האילוץ $q_y \leq q_x$.
- נבדוק האם קיימת השמה של אמין וחשוד המספקת את כל האילוצים. אם כן - יש בתכנית רק זרימות חוקיות.
אחרת - אם ייתכן וקיימת ריצה של התכנית שיש בה זרימה לא חוקית.
- אם מתקיים התנאי הבא $tainted \leq untainted$ אזי הזרימה לא חוקית.
- **מה נעשה כשיש תנאי בקוד:** אם יש לנו משתנה שהשמתו מותנית בתנאי בקוד, נצטרך לכלול את כל האפשרויות להשמת המשתנה - נתעלם מהתניות, על אף שבכל ריצת תכנית רק השמה אחת תתרחש משום שאנו לא רוצים לפספס פרצות. אך גישה זו תגרום לנו לאזעקות שווא.

כיצד נהפוך $flow\ analisis$ **לרגישה יותר - הימנעות מאזעקות שווא:**

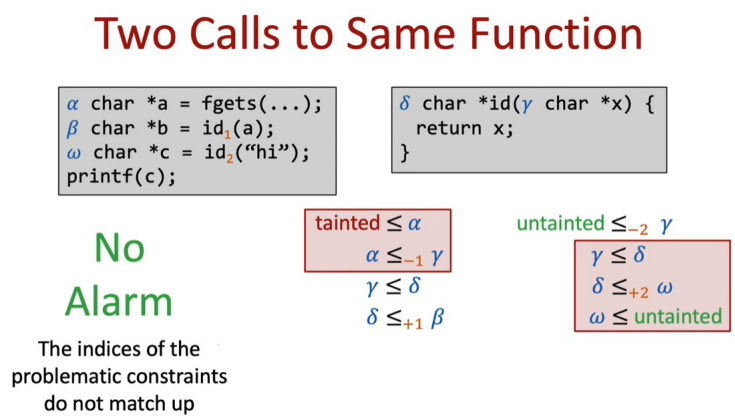
- **מה אנו רוצים:** שלא יהיו לנו יותר מידיי FN - הפחתת אזעקות שווא.

- **רגישות** $flow\ sensitivity$**:** גישה זו באה לטפל במקרים בהם **יש לנו יותר מהשמה אחת לכל משתנה,** לדוגמה - משתנה מקבל השמה כחשוד אך לאחר מכן בהמשך הקוד הוא מקבל השמה כאמין. נרצה שהוא יסוווג כאמין משום שבסוף הקוד כשיעשה בו שימוש הוא אכן יהיה אמין.
הפתרון: נשתמש ב $single\ assignment\ form$, נשייך לכל ערך בתכנית משתנה לכל **השמה** לערך, ולא רק לערך עצמו. כאשר יש שתי השמות לאותו ערך בקוד, נגדיר לכל השמה משתנה, באופן זה נעביר בסוף הקוד את הערך האמין והאילוצים יתקיימו כנדרש.
החסרון: אנו מעלים את מספר האילוצים.

- **רגישות** $path\ sensitivity$**:** גישה זו באה לטפל **במקרה בו הזרימה הלא חוקית מתאימה למסלול בקוד שאין אף ריצה שמריצה אותו.** לדוגמה - אנו עושים השמה חשודה למשתנה בהתאם לתנאי כלשהו, אך אנו לא משתמשים בהשמה החשודה אם מתקיים אותו התנאי.
הפתרון: נפריד את ריצת התכנית למסלולים אפשריים, מסלול אחד עבור כל תנאי בקוד. עבור כל אילוץ נוסיף לו את המסלול אליו הוא שייך וננתח עבור כל מסלול בנפרד. באופן זה אם בסוף הקוד לא נעשה שימוש במסלול מסויים, לא נקח אותו בחשבון וכך נוכל להימנע מאזעקות שווא במקרה והמסלול הזה גורם לבעיות.
החסרון: לרוב המימוש והמעקב אחר מסלולים יהיה מורכב.
דוגמה:



- **רגישות** $context\ sensitivity$**:** באה לטפל **במקרים בהם אנו קוראים לפונקציה אחת מספר פעמיים,** בכל פעם עם קלט אחר כך שאחת מהפעמים עם קלט חשוד. וכך למרות שיש השמה מספקת, התכנית תשלח לנו אזעקת שווא
הפתרון: נפריד בין קריאות שונות לפונקציות, ע"י כך שנשנה את בניית האילוצים. נוסיף תוויות שמסמנות את מספר הקריאה לפונקציה או מספר שורת הקוד בה היא נקראת , בנוסף נסמן ב "-" קריאה לפונקציה (השמת משתנים לארגומנטים של הפונקציה), וב "+" חזרה מפונקציה (השמת ערך ההחזרה של הפונקציה למשתנה).
הערה: רגישות זאת לא יכולה להבטיח השמה מספקת, והאם יש זרימה חוקית. כדי להתמודד עם קושי זה נוכל להשתמש בשיטה זו באופן חלקי - עד עומק מסויים או רק עבור חלק מהפונקציות.
דוגמה:



כיצד נימנע מפספוס זרימות לא חוקיות (FP) **- זיהוי** $implicit\ flows$**:**

- **זיהוי זרימת מידע בין מצביעים:** כאשר יש השמה של מצביעים הזרימה כפי שהגדרנו אותה תראה כי כל האילוצים יכולים להתקיים על אף שיש זרימה לא חוקית, משום שהשמות למצביעים יכולים לגרום לזרימה דו כיוונית.

הפתרון: לכל השמה בין מצביעים נוסיף אילוץ דו כיווני.

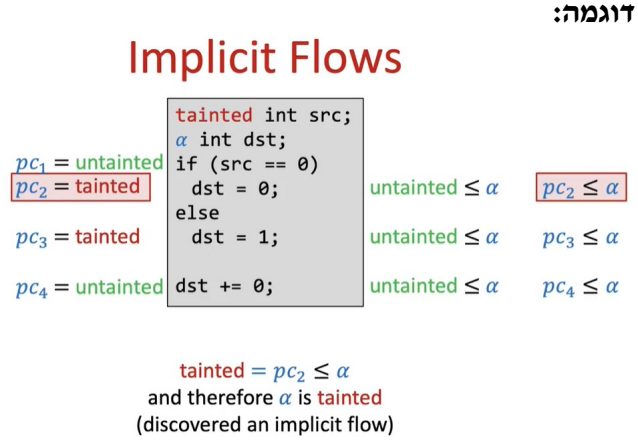
החסרון: שיטה זו יכולה לגרום לאזעקות שווא אך כמובן שנעדיף FN על FP.

- **שיטה נוספת לזיהוי** $implicit\ flows$ **-** $information\ flow\ analisis$**:** יש זרימות לא מפורשות שהשיטה הראשונה לא תזהה, לכן נגדיר את השיטה הבאה.

הפתרון: לכל שורת קוד המערבת ערכים מהתכנית - נוסיף משתנה pc, והוא יסמן את שורת הקוד. כעת נשים אילוץ אחד על הזרימה $q_y \leq q_x$, ואילוץ נוסף - $pc \leq q_x$ המסמן כי עצם הגעת התכנית לשורה זו עשויה להשפיע על ערך המשתנה x. נגדיר את המשתנה pc באופן הבא - אם הגענו לשורה זו כתנאי במעבר דרך משתנה חשוד, אזי pc יוגדר כחשוד.

החסרון: שיטה זו גורמת לאזעקות שווא רבות. בנוסף המאמץ החישובי הנוסף גדל ויעילות הניתוח יורדת.

דוגמה:



- **אתגרים:** סכום משתנים - אמין ולא אמין, מצביעים לפונקציות, אובייקט המורכב ממספר חלקים שחלקם חשודים.

פתרונות נוספים: הוספת אילוצים לערכים - $buffer\ overflow$. לאמשר שינוי ערכים כך שיהפכו לאמינים. כיסוי מפגעי אבטחה אחרים - החלפת אמינות בסודיות.

0.9 תרגול 10 - $flow\ analisis$:

- **נשתמש ב** $flow\ analisis$ בכדי לעקוב אחר האופן שבו מידע זורם בין מקומות שונים בזכרון. הרעיון הבסיסי הוא - התקפות רבות מתבצעות כי הקוד נותן אמון במידע לא אמין.
ראינו כי ערכים שמתקבלים מהמשתמש הינם חשודים, בעוד שהקוד שלנו מניח כי הפונקציה מקבלת ערכים אמינים.
נרצה לבדוק האם הקוד שלנו מבצע שימוש בערכים חשודים תחת ההנחה שהם אמינים.

- נרצה שיתקיים $untainted \leq tainted$.

0.10 הרצאה 11 - $Symbolic\ Execution\ and\ Fuzzing$:

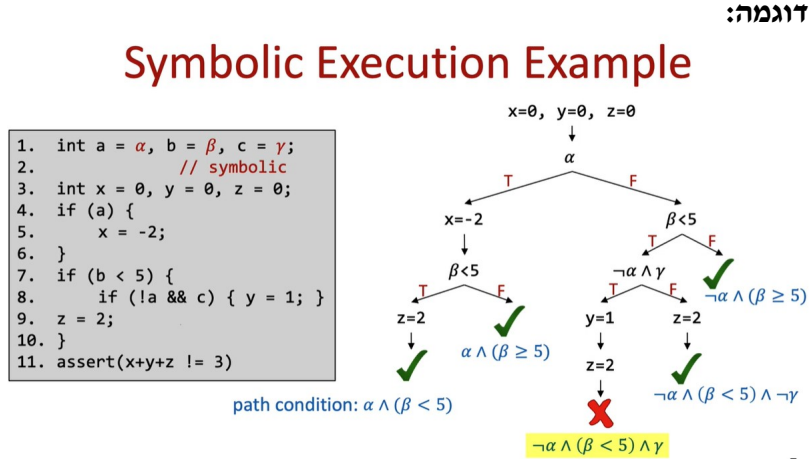
0.10.1 הרצה סימלית - $Symbolic\ Execution$:

- **טסטים:** כפי שאמרנו טסטים בודקים לנו את הקוד עבור קלט ספציפי. לכן אם לא נריץ טסט מסויים נפספס ריצה של חלק מהקלטים.
בנוסף, טסטים הם **שלמים** (אם יש התראה אזי יש כשלון) אך לא **תקפים** (אם אין התראה זה לא אומר שאין באגים, אלא יכול להיות שלא הרצנו מספיק טסטים).
נרצה לפתח מערכת שמבוססת על טסטים למציאת מפגעי אבטחה, שיותר תקפה מטסטים.

- **הרעיון:** נתייחס לחלק מהמשתנים כאל משתנים סמליים, לאחר מכן נוכל להוסיף עליהם $assert$ ולדעת אם מתקיים תנאי מסויים. באופן זה אנו יכולים לעקוב אחר מסלולים בתכנית במידה והיא מתפצלת.
כלומר בכל תנאי שכולל משתנה סמלי אנו נריץ את כל האופציות שמשתנה זה יכול לקבל, כך אנו נוכל לעקוב אחרי כל ריצה של התכנית.
הערה: מבחינה חישובית המורכבות גדולה.

- **הגדרה - תנאי המסלול** $(path\ conditon)$: מעקב אחר מסלול בתכנית שהביא אותנו אל הנקודה הנוכחית, בעקבות תנאים מסויימים שהתקיימו.
תנאי המסלול יכילו רק משתנים סמליים ולא רגילים.

- **דוגמה:**



- **ההבדל מטסטים:** כך אנו מצליחים להריץ את כל המסלולים האפשריים של התכנית. בנוסף אנו יכולים לעקוב אחרי המסלול שהוביל אותנו למיקום בו התכנית נכשלה.

- **בניגוד לאנליזה סטטית:** בהרצה סמלית אין אזעקות שוא משום שכל מסלול מתאים ללפחות ריצה אמיתית אחת, לכן היא כוללת כל סוג של רגישות שיש באנליזה סטטית.

החסרון: בתכניות מסובכות מעבר על כל המסלולים היא בעיית $NP-complete$. בעוד שאנליזה סטטית תמיד מגיעה לסיומה ומכסה את כל המסלולים.

פתרון: נריץ רק חלק מהמסלולים או שניתן להרצה הסמלית לרוץ מבלי לדעת מתי היא תסיים. הבעיה היא שלא נעבור על כל האפשרויות ונוכל לפספס מפגעי אבטחה.

- **מה נגדיר כמשתנים סמליים:** כל משתנה שמגיע מהמשתמש ועשוי להיות לא בטוח.

- **הגדרה - בדיקת פזיביליות של מסלולים**: נוסיף תנאים לפני המקומות החשודים ונבדוק האם המסלול אליהם פיזבילי - ניתן להגיע אליו, אם כן - אזי יש אופציה למפגע אבטחה בתכנית.

- **אלגוריתם ההרצה של הרצה סמלית עבור פיצולים:** נחזיק שלשה משתנים -

$pc = 0$: מחזיק את שורת הקוד שרצה.

$\pi = \emptyset$: תנאי המסלול הנוכחי.

$\sigma = \emptyset$: יחזיק את כל מצב הזכרון הסמלי של התכנית - משתנים רגילים וערכיהם, בנוסף יחזיק השמות לביטויים סמליים.

פסואודו קוד:

1. Initilization:
   Program counter $pc = 0$, path condition $\pi = \emptyset$,
   symbolic state $\sigma = \emptyset$
2. Insert task (pc, π, σ) to worklist
3. While (worklist is not empty)
   A. Remove and execute some task (pc, π, σ) from worklist [this step may modify pc and σ but not π]
   B. If execution potentially forks at (pc_0, π_0, σ_0):
      i. If $\pi_0 \wedge p$ is satisfiable, insert the task $(pc_1, (\pi_0 \wedge p), \sigma_0 \cup \{p = \text{True}\})$ to worklist
      ii. If $\pi_0 \wedge \neg p$ is satisfiable, insert the task $(pc_2, (\pi_0 \wedge \neg p), \sigma_0 \cup \{p = \text{False}\})$ to worklist

```
pc0.  if (p) {
pc1.     ...
pc2.  } else {...
...
```

ניתן להגדיר את ה $worklist$ להריץ עם BFS - תור, או DFS - מחסנית, בהתאם למה שאנו רוצים למצוא. האם אנו רוצים חיפוש במסלולאחד לעומק, או חיפוש רוחבי על מספר מסלולים.

באופן כללי נעדיף BFS כדי לסרוק כמה שיותר מסלולים, אך החסרון הוא שלא ניתן להריץ אותו עם דינמיות.

גישה נוספת היא הרצת מסלולים באקראי - אקראיות המתקבלת מפונקציה פסואודו אקראית על קלט המתקבל ממצב התכנית, כך נוכל לשחזר את ריצת התכנית בהינתן המפתח.

- **מה נעשה כשהתכנית קוראת לקוד חיצוני:** לא נוכל לבצע הרצה סימלית באופן מושלם אם אין לה את היכולת לעקוב אחר הקוד החיצוני.

אפשרות 1: נכלול את הקוד החיצוני בקוד שלנו בצורה מפורשת.

אפשרות 2: נמשיך בהרצה הסמלית עם גרסה פשוטה יותר של הקוד החיצוני.

אפשרות 3: נחליף אותו בקוד שימדל אותו.

אפשרות 4: נשתמש בהרצה סמלית דינמית.

- **הרצה סמלית דינמית:** שתי הרצות במקביל, הרצה אמיתית של התכנית כשברקע יש הרצה סמלית שמתאימה לה. מצב הזכרון האמיתי יקבע את המסלול. כך תהיה עקביות בין הזכרון האמיתי למסלול הסמלי.

0.10.2 $Fuzzing$:

- **מה זה:** גישה המבוססת על טסטינג, ממומשת באמצעות קלטים אקראיים או אקראיים למחצה. המטרה היא לגלות התנהגות לא תקינה שיטחית כדוגמת התרסקויות, חריגות ולולאות אינסופיות.

- **מימוש:** יש שלש אפשרויות ליצירת קלטים.
  - **קופסה שחורה:** לא נדרשת הכרות מוקדמת עם התכנה או הקלטים שלה, מתקילים אותה עם קלטים אקראיים.
  - **קלטים בעלי מבנה בהינתן דקדוק:** נגדיר דקדוק מסויים (חסר הקשר או ביטוי רגולרי) ונדגום ממנו קלטים. הרמה כאן יותר גבוהה ויש יותר קשר לתכנית.
  - **קופסה לבנה**: ניצור קלטים בהתבסס על הבנה של התכנית, ניתן לעשות זאת ע"י הגדרת דקדוק, אך מבנה הדקדוק ותהליך הדגימה מושפעים מקוד התכנית.

- **יצירת הקלטים:** יש כמה אפשרויות ליצירת הקלטים.
  - מוטציה אקראית של קלט ראשוני נתון, הקלט חוקי ויכול להיקבע ע"י המשתמש או באופן אוטומטי (עם דקדוק או $SMT\ solver$).
  - יצירת קלט מראש באופן אוטומטי עם דקדוק.
  - יצירת קלט ראשוני באופן אוטומטי, ויצירת מוטציות עם דקדוק.

- **מה נעשה במקרה של התרסקות:** לאחר שהרצנו את התכנית והיא התרסקה עם קלט כלשהו, נרצה להגיע לקלט קצר ופשוט יותר שיגרום להתרסקות כדי שנוכל לפשט את הבעיה. בנוסף נרצה לדעת האם מספר קלטים שונים נגרמו מאותה התרסקות. והאם הסיבה להתרסקות עשויה להוביל למפגע אבטחה (לדוגמה באפר אוברפלואו).

- **כיצד נרסק תכניות במקרה של אוברפלואו:**
  - נשתמש ב $addres\ sanitizer$, תכנית הבודקת כי לא מתבצע אוברפלואו.
  - נריץ $fuzzer$, ואם תהיה התרסקות כתוצאה מאוברפלואו היא תתרחש **מיידית**. כך נוכל לגלות מפגעי אבטחה פוטנציאלים.

חלק III

טריקים למבחן:

0.11 קריפטוגרפיה:

0.11.1 שיטות הוכחה:

- **הוכחה ברדוקציה - שלבים:**
  1: נניח בשלילה כי הטענה לא מתקיימת.
  2: נראה כי קיים מבחין D שרץ בזמן פולינומיאלי ושובר את בטיחות Π לפי הגדרות הטענה.
  3: נבנה יריב A שמשתמש ב D, ושובר את בטיחות הטענה לפי הגדרתה.
  4: נגיע לסתירה.

- **שיטת הוכחה:** כאשר נצטך להוכיח $|A-B| \leq \varepsilon$ נוכל לעשות זאת באופן הבא:

$$|A-B| = |A-B+C-C| \leq |A-C| + |C-B|$$

- **הוכחה היברידית:** כשנרצה להוכיח בטענה היברידית נשתמש באש"מ כך - נקח את האובייקט הראשון מהארגומנט הימני, ואת האובייקט השני מהארגומנט השמאלי.

$$|A^{H(\cdot)||F(\cdot)} - A^{h(\cdot)||f(\cdot)}| \leq |A^{H(\cdot)||F(\cdot)} - A^{h(\cdot)||F(\cdot)}| + |A^{f(\cdot)||F(\cdot)} - A^{h(\cdot)||f(\cdot)}|$$

0.11.2 נוסחאות שימושיות:

- **נוסחת ההסתברות השלמה:** $P(B) = \sum_{A \in A} P(B \mid A_i) P(A_i)$
- **כלל בייס:** $P(B \mid A) = \frac{P(A|B) \cdot P(B)}{P(A)}$
- **הסתברות מותנית:** $P(A \mid B) = \frac{P(A \cap B)}{P(B)}$
- **חסם האיחוד:** עבור קבוצת מאוריות A_i מתקיים - $Pr\left[\bigcup_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} Pr\left[A_i\right]$

0.11.3 סודיות מושלמת:

- **הגדרה** - מערכת תיקרא סודיות מושלמת אם לא ניתן לגלות על m שומדבר בהינתן c: כלומר מתקיים - $P(M = m|C = c) = P(M = m)$.
- Π מערכת סודיות מושלמת אמ"מ לכל m_0, m_1 מתקיים $P(C = c \mid M = m_0) = P(C = c \mid M = m_1)$.

- **מערכת** OTP**:** היא סודית מושלמת, עבור $c = m\ XOR\ k$
- **טענה:** אם Π מערכת סודיות מושלמת אזי $|K| \geq |M|$.
- **טענה:** אם Π מערכת סודיות מושלמת אזי $P(C = c \mid M = m) = P(C = c)$
- **טענה:** מערכת שלא יכול להתקיים בה $m = c$ (לא ניתן להצפין הודעה לעצמה), היא לא סודיות מושלמת.

0.11.4 בלתי ניתנות להבחנה:

- **הגדרה** - מערכת תוגדר **בלתי ניתנת להבחנה** אם בהינתן שתי הודעות מוצפנות לא ניתן להבחין בניהן בהסתברות יותר מזניחה:

$$P(IND_{\Pi,\mathcal{A}}(n) = 1) \leq \frac{1}{2} + \nu(n)$$

כאשר הניסוי $IND_{\Pi,\mathcal{A}}$ **מוגדר באופן הבא:** אנו מצפינים אחת משתי הודעות m_0, m_1 שהיריב בחר, והוא אמור לנחש איזו הודעה הצפנו.

- **פונקציות זניחות:** סכום של פונקציות זניחות הוא גם פונקציה זניחה.
  - מכפלת פולינום ופונקציה זניחה הם פונקציה זניחה.
- **סודיות מושלמת ובנ"ל:** אם מערכת Π היא סודיות מושלמת אזי היא גם בלתי ניתנת להבחנה.

0.11.5 יצרן פסואודו אקראי:

- **ייצרן פסואודו אקראי** G**:** מקיים שתי תכונות - $|G(n)| > n$. וגם לכל מבחין D מתקיים:

$$\left| \underset{s \leftarrow \{0,1\}^n}{P}(\mathcal{D}(G(s)) = 1) - \underset{r \leftarrow \{0,1\}^{\ell(n)}}{P}(\mathcal{D}(r) = 1) \right| \leq \nu(n)$$

כלומר - בהינתן פלט של היצרן, ופלט רנדומי, המבחין לא יוכל להבחין בניהם ולזהות איזה פלט שייך ל G יותר מהסתברות זניחה, משום ש G מתנהג כמו התפלגות אחידה.
הניסוי: המבחין D מחזיר 1 עבור פלט של G ו 0 עבור פלט רנדומי.

- **הנחה בשלילה עם מבחין** D**:** כאשר נרצה לסתור הנחה נצטרך לחשב את שני המקרים עבור המבחין, גם כאשר הקלט רנדומי וגם כשהוא פלט של היצרן הפסואודו אקראי.
עבור הצפנות בנ"ל: נרצה לנצל את היכולת של המבחין להבחין בין פלט רנדומי לפלט של היצרן, לכן ניצור לו שתי הודעות כך שאחת תהיה רנדומית ואחת תהיה דומה ליצרן.
כשנניח בשלילה נניח כי הטענה מתקיימת עבור אינסוף ערכי n.

- **הוכחה שיצרן** G **הוא פסואודו אקראי:** נוכיח ש -
  1: G חשיבה בזמן פולי'
  2: G היא פונקציה מרחיבה - עבור קלט באורך n מתקיים $|G(n)| > n$.
  3: G היא פונקציה פסואודו אקראית (בעזרת שלילה או היברידיות).

- **הוכחה על פ"א וקלט רנדומי:** כשניצור קלט רנדומי למבחין ע"י אורקל, נצטרך להוכיח כי אכן הפלטים שנוצרו רנדומים וב"ת.

- **שינוי של יפ"א בנקודה אחת הוא גם יפ"א:** עבור G יפ"א שמרחיב את הקלט פי 2 מתקיים כי H הבא גם יפ"א

$$H(s) = \begin{cases} 0^{2|s|} & \text{if } s = 0^{|s|} \\ G(s) & \text{otherwise} \end{cases}$$

- **שרשור של יפ"א שונים** G_0, G_1 **עם אותו הקלט: אינו** בהכרח יפ"א, גם אם הם ממפים לקלטים שונים (כי אפשר לבחור $G_0(s) = \overline{G_1(s)}$).

- **שרשור של קלטים עם אותו יפ"א:** עבור יפ"א G מתקיים כי $H(s_1, s_2) = G(s_1) \| G(s_2)$ גם יפ"א.

- **יפ"א ובנ"ל:** אם אנו מצפינים עם ייצרן פסואודו אקראי $c = m \oplus G(k)$, אזי המערכת בלתי ניתנת לאבחנה. ההפך גם נכון, מערכת בנ"ל עם יצרן גוררת כי הוא בהכרח פסואודו אקראי.

- **הגיגים על** XOR**:** עבור הודעה $m \in \{0,1\}^n$ מתקיים: $(m \oplus 0^n) = m, \quad (m \oplus 1^n) = \overline{m}$
  וגם: $c = m \oplus k \iff k = m \oplus c \iff m = c \oplus k$

- **יצרנים שאינם פסואודו אקראיים:** $G(s) = s_1 \cdots s_n \| s_1 \oplus \cdots \oplus s_n, \quad G(s) = s_1 \cdots s_n \| s_1, \quad G(s) = s_1 \cdots s_n \| 0$

0.11.6 בטיחות סמנטית:

- **הגדרה - בטיחות סמנטית:** נאמר כי מערכת היא בטוחה סמנטית אם ההסתברות לדעת משהו על הודעה m בהינתן מידע על m שווה להסתברות ללא המידע.

$$|P(\mathcal{A}(1^n, Enc_k(m), h(m)) = f(m)) - P(\mathcal{S}(1^n, h(m)) = f(m))| \leq \nu(n)$$

  כאשר $f(m)$ הוא המידע שנרצה לגלות על m, בהינתן מידע $h(m)$ על m.

- **בטיחות סמנטית ובנ"ל:** מערכת Π בטוחה סמנטית אמ"מ היא היא הצפנה בלתי ניתנת להבחנה.

0.11.7 בלתי ניתנות להבחנה חישובית:

- **הגדרה - בלתי ניתנות להבחנה חישובית:** עבור שתי **התפלגויות** X, Y נאמר כי הן בנ"ל חישובית אם:

$$\left| \underset{x \leftarrow X_n}{P}(\mathcal{D}(1^n, x) = 1) - \underset{y \leftarrow Y_n}{P}(\mathcal{D}(1^n, y) = 1) \right| \leq \nu(n)$$

- **טענה:** עבור פונקציה f חשיבה ביעילות (בזמן פול) אזי $f(X), f(Y)$ גם בנ"ל חישובית.

- **ניסוי** CPA**:** היריב יוכל להצפין כל הודעה m' שירצה ויקבל עבורה הודעה מוצפנת c'. לאחר מכן הוא בוחר שתי הודעות, אנו מצפינים אחת מהן ומחזירים לו. הוא אמור לנחש איזו הודעה הצפנו.

- **הגדרה - מערכת בנ"ל** CPA**:** אם הצלחת היריב בניסוי קטנה מחצי + פונקציה זניחה:

$$\mathbb{P}\left[IND^{\text{CPA}}_{\Pi,\mathcal{A}}(n)=1\right] \leq \frac{1}{2}+\nu(n)$$

היריב לא יוכל לבקש להצפין הודעה אחת ואחכ להגיש את אותה הודעה בניסוי ולהצליח לנחש טוב יותר, משום שיש אלמנט הסתברותי בהצפנה ושתי ההצפנות יהיו שונות.

- **חסם על הודעה אחת בניסוי** $IND^{\text{CPA}}_{\Pi,\mathcal{A}}$**:** עבור $q(n)$ מספר הבקשות של יריב A מהאורקל, אזי הסיכוי של הצפנה לחזור על עצמה חסומה ע"י: $P(repeate) \leq \frac{q(n)}{2^n}$.

- **מערכת הצפנה עמידה בפני מתקפת** CPA**:** נדגום קלט רנדומי r נצפין עם פפ"א כך - $s = m \oplus F_k(r)$ ונחזיר $c=(r,s)$, האלגוריתם $Dec(r,s)$ יפענח כך $m = s \oplus F_k(r)$.

0.11.8 פונקציות פסואודו אקראיות:

- **הגדרה - פונקציה פסואודו אקראית:** פונקציה $F : \overbrace{\{0,1\}^n}^{key} \times \overbrace{\{0,1\}^n}^{r} \to \{0,1\}^{\ell(n)}$ שלא ניתן להבחין בינה לבין פונקציה h הנדגמת באופן אחיד ממרחב הפונקציות, ביותר מהסתברות זניחה:

$$\left|\mathbb{P}\left[D^{F_k(\cdot)}\left(1^n\right)=1\right]-\mathbb{P}\left[D^{h(\cdot)}\left(1^n\right)=1\right]\right| \leq \nu(n)$$

- **טענה:** תהי F פונקציה פסואודו אקראית, ותהי H הפונקציה הבאה $H_k(x) = F_{F_k(0^n)}(x)$, אזי H פונקציה פסואודו אקראית.

- **גישת אורקל לפונקציה:** כשיש לנו גישת אורקל לפונקציה F_k אזי הפונקציה F ידועה לנו **ורק המפתח אינו ידוע**, לכן אם יש שרשור של F עם מפתחות שונים, הגישה תהיה לאורקל של המפתח הספציפי ואת שאר המפתחות נדגום אחיד.

- **שינוי של פפ"א בנקודה אחת אינה פפ"א:** בניגוד ליפ"א, אם נגדיר פונקציה $F_k(x)$ שמתנהגת באופן הבא:

$$F_k(x) = \begin{cases} 0^n, & x = 0^n \\ H_k(x), & else \end{cases}$$

עבור H_k פונקציה פסואודו אקראית. אזי הפונקציה F **אינה** פסואודו אקראית, משום שהתוקף יכול לשלוח את הקלט 0^n ולהבחין בינה לבין פונקציה אקראית.

- **נשים לב:** בכל שרשור של פונקציות בו יש דמיון בין המפתח של החלק הראשון לשרשור בחלק השני, הפונקציה המתקבלת אינה פסואודו אקראית.
למעשה בכל מקום בו יש דמיון בין שני חלקי השרשור כך שאחד מגלה פרטים על השני אין פסואודו אקראיות.

דוגמה לשרשור: הפונקציה $W_k(x) = F_k(x)||F_{\overline{k}}(x)$ אינה פ"א. כי ניתן להגדיר פונקציה $\phi(k)$ על המפתח כך ש $\phi(k) = \phi(\overline{k})$ ע"י השמטת הביט הראשון והיפוך שאר הביטים אם $k_1 = 0$, אחרת רק נשמיט את הביט הראשון. והגדרת $F_k = H_{\phi(k)}$ כאשר H פפ"א.

- **שרשור של פונקציות פ"א:** עם מפתחות ב"ת לכל אחת מהפונקציות, היא פפ"א.
- **יצרן ופונקציה פסואודו אקראיים:** בהינתן פפ"א ניתן ליצור ממנה יפ"א .
- **הצפנה עם פפ"א:** יוצרת לנו מערכת Π בטוחה CPA.
  ההפך לא בהכרח נכון - אם מערכת היא בטוחה CPA אזי הפונקציה **לא בהכרח** פסואודו אקראית.
- **הצפנות בנ"ל ופפ"א:** הצפנה בנ"ל עם פונקציה **לא בהכרח** שהפונקציה פפ"א.

0.11.9 מערכות לאימות הודעות:

- **מערכת לאימות הודעות** מורכבת משלשה אלגוריתמים: Gen- יצירת מפתח k. Mac- יצירת ערך אימות t. $Vrfy$ - מקבל m, k, t ומחזיר 1 אם יש התאמה, 0 אחרת.
- **ניסוי** $MacForge$ **למערכת אימות הודעות:** ניתן ליריב A גישת אורקל לאלגוריתם Mac שייצור ערכי אימות כרצונו, לאחר מכן נבדוק האם הודעה m (שהוא לא בדק עבורה ערך אימות בעבר) והצפנה t שהוא ייצר בעצמו (ללא אלגוריתם Mac), אכן תואמים.

ה - מערכת Mac **בטוחה:** נאמר במערכת בטוחה אם הסיכוי להצלחה בניסוי קטנה מפונקציה זניחה: $\mathbb{P}\left[MacForge_{\Pi,\mathcal{A}}(n) = 1\right] \leq \nu(n)$

- **הוכחה בסתירה למערכת אימות:**
  נוכיח כי האלגוריתם פולינומי - מריץ אלגוריתם פולינומי, פוקנציה שעושה חישוב פולינומי או שרשור של פולינומים.
  נניח כי ההודעה שנשלחה $m \notin Q$, כי אחרת ההוכחה לא תקפה.
- **מערכת אימות ופונקציות פ"א:** מערכת אימות המוגדרת באמצעות פפ"א היא בטוחה.
- **אימות הודעות מאורך משתנה:**
  פתרונות גרועים:
  - אם נצפין כל בלוק בנפרד, תוקף יוכל לאמת חלק מההודעה (בלוק בודד) ולעבור את הניסוי.
  - אם נצפין גם לפי גודל בלוק, תוקף יוכל לשנות את סדר הבלוקים וסדר המאמתים ולעבור את הניסוי.
  - אם נקח בחשבון גם את סדר הבלוקים, תוקף יוכל להרכיב הודעה היברידית משתי הודעות שהוא אימת עם האלגוריתם.

  פתרונות טובים:
  - נגדיר את אלגוריתם Mac עם פרמטר r הסתברותי, עם גודל בלוק וסדר בלוקים. **החסרון:** יש לנו d ערכי אימות

וזה מלא.

- $CBC-MAC$: נגדיר סדרה רקורסיבית, כל הודעה מוצפנת היא ערך האימות של ההודעה הבאה

$$t_0 = 0^n, \quad t_i = F_k\left(t_{i-1} \oplus m_i\right)$$

החסרון: זמן ריצה ארוך. **הערה:** אורך ההודעות צריך להיות **קבוע** מראש.

- פונקציית האש H שתקטין את m ואלגוריתם Macיצפין את $H(m)$.

0.11.10 פונקציות האש:

- **פונקצית האש:** מוגדרת כך: $\Phi(Gen, H)$ - אלגוריתם יצירת מפתחות Gen, ופונקציה H_k שמכווצת את x עם k.
- **ניסוי** $HashColl_{\Phi,A}(n)$: עבור יריב A ניתן לו גישת אורקל **למפתח ולפונקציה** וניתן לו לבחור שתי הודעות x, x' כך שאם $H(x) = H(x')$ הוא ניצח.
- **מערכת האש בטוחה:** נאמר כי פונקציית האש בטוחה (אמידה בפני התנגשויות) אם היריב לא מצליח למצוא שתי הודעות שממופות אל אותו המקום:

$$Pr\left[\text{ HashColl }_{\Phi,\mathcal{A}}(n) = 1\right] \leq v(n)$$

כיום מגדירים פונקציות $H : \{0,1\}^n \Rightarrow \{0,1\}^{128}$ משום שמתחת ל 128 הן לא בטוחות.

- **טענה:** מערכת אימות הודעות לא בהכרח מבטיחה הצפנה, ולהיפך.
- **מערכת אימות הודעות והצפנה** - $Authenticated\ Encryption$: נצפין את ההודעה m, ולאחר שנקבל הודעה מוצפנת c, נחשב לה מפתח אימות t.

0.12 אבטחת תכנה - מתקפות:

- **כיצד נראית מחסנית בתחילת ריצת פונקציה:**

1: בהתחלה יהיו במחסנית ה $callers\ data$ - מידע שהיה במחסנית בעת קריאת הפונקציה.

2: לאחר מכן ארגומנטים של הפונקציה נכנסים בסדר הפוך. (אם הפונקציה מקבלת מערך, ישלח **מצביע** למערך בגודל $4B$).

3: אחריהם יהיו המצביעים eip - הכתובת אליה נחזור בסיום הריצה.

4: ebp - הכתובת שמתאימה ל $frame\ pointer$ של הפונקציה הקוראת, הערך שהיה שמור ברגיסטר ebp בעת ריצת הפונקציה הקוראת (ישמר מצביע לקצה השמאלי של הערך במחסנית, לכן גישה למשתנה לוקאלי $loc2$ במחסנית תהיה $ebp - size(loc1 + loc2)$). הפונקציה הנקראת אחראית על שלב זה.

5: אחכ יכנסו המשתנים הלוקלים בסדר הופעתם.

בכל קריאה נוספת לפונקציה אחרת: יתווספו גם ערכי ה ebp וה eip של הפונקציה הנקראת.

בקריאה לפונקציה $printf$: נשמור מקום ארגומנטים של הפונקציה כמו פונקציה רגילה, לאחר מכן ייכנסו ערכי ה ebp, eip. אם נכניס קלט יחיד עם מציני פורמט, הפונקציה תחשוב שהכנסנו לה עוד ארגומנטים, ולכן עבור כל מציין

פורמט היא תזוז ארבע בתים ימינה במחסנית החל מהמיקום $ebp + 12$ שאמור להיות מוקצה לארגומנט השני של הפונקציה, ותקח את הערכים שנמצאים שם.
בקריאה לפונקציה $fgets$: לא נשמור מקום במחסנית לבאפר, אלא נשתמש במיקום שנשמר לבאפר באתחול, המידע רק יכתב לבאפר.

- **מתקפה עם הזרקת כתובת של הפונקציה** $printf$: כאשר אנו מזריקים למחסנית את הכתובת של הפונקציה $printf$ היא תשמור את המצב הנוכחי ברגיסטר ebp ותסתכל 8 ביטים ימינה (אחרי המקום של ebp, eip) שם אמורים להימצא הארגומנטים שלה, ותדפיס את מה שנמצא שם.

- **משיכת מידע בעזרת הפונקציות** $get, print$: נכניס מצייני פורמט לפונקציה get והם יישמרו בבאפר. לאחר מכן כשנרצה להדפיס את הבאפר הוא ימשוך מידע מהמחסנית (מימין לבאפר). אם נשתמש בפונקציה אחרת שקוראת לפונקציה $print$, אזי יכנס גם מצביע ebp של הפונקציה הראשית וכך נוכל לחשב כתובות באופן יחסי במחסנית של הפונקציה הראשית.

- **מצייני פורמט:** $\%n$ - כותב לזיכרון במיקום $ebp + 12$ ($12B$ **ימינה**, המיקום שאומר להיות שייך לארגומנט השני של הפונקציה) את מספר התווים שהודפסו לפני מציין הפורמט, לדוגמה עבור $printf\ (100\%\ n)$ יכתב לזיכרון הערך 3.

- **כתובת של משתנה**: מצביעה על הצד השמאלי קיצוני של המשתנה. המרחק היחסי של באפר בגודל $12B$ מהכתובת add (הקצה השמאלי של הכתובת) הוא $add - 12$.

- **כיצד הפונקציה מוצאת את המשתנים במחסנית:** היא יודעת בזמן קומפילציה היכן כל משתנה נמצא ביחס למיקום שלו ל ebp, לכן אם נשנה את ערך ה ebp היא תבחר משתנה אחר.

- **מבנה המחסנית:** הכתובות השמאליות הן הכתובות החדשות והנמוכות יותר במספרן, בעוד הכתובות הימניות הן גבוהות במספרן. לכן אם נכניס מציין פורמט ולא יהיה קלט תואם, אנו נקרא את כל מה שנמצא במחסנית מצד **ימין** לכתובת הנתונה. ואם נחסר מכתובת מסויימת (-) נזוז **שמאלה**, לכן גישה למשתנה לוקאלי $loc2$ במחסנית תהיה $ebp - size(loc1 + loc2)$.

- **מתקפת באפר אוברפלואו:** תחילה נקמפל קוד C ונעביר אותו לשפת מכונה עם כתובות מדוייקות. לאחר מכן נדרוס את כתובת ה eip (ערך החזרה מהפונקציה), ונשתול שם את כתובת הקוד המוזרק. ניתן להשתמש בשורות $noOp$ שיביאו אותנו אל הקוד המוזרק.
הערה: אם בקוד יש שורה שכולה אפסים אזי המתקפה יכולה להיכשל, כי פונקציות של C מפסיקות להעתיק כשהן מגיעות לתו 0.

- **מתקפות אופציונליות:**
- **מתקפת באפר אוברפלואו** שדורסת את המצביע eip ומריצה קוד עויין. אם נדרוס את eip עם שרשרת של מצביעים לפונקציות, כולן ירוצו אחת אחרי השניה, כי כל אחת בסיומה צריך את האחרת בתור eip שלה.
- **פויינטר ששוחרר** ועדיין מצביע למקום מסויים, במידה ונשתמש בו שוב בטעות, תוקף יוכל לנצל זאת.
- **הדפסה עם מצייני פורמט** - אם נשתמש בפונקציות בצורה לא בטוחה, תוקף יוכל לשלוח מצייני פורמט ולשאוב מידע מהמחסנית בהדפסה.

- **מתקפת** $libc$**:** הכתובות של הפונקציות של הספריה ידועות כבר בזמן קומפלציה, כך תוקף יכול להריץ פונקציית $exec$ שתבצע קוד שהתוקף יזין לה כארגומנט.

- **מתקפות:**

אם נרצה להגיע למצביע כלשהו במחסנית: נוכל להדפיס $\%p$ או לחשב את הערך של $ebp - i$ ולהגיע למצביע שאנו מחפשים.

אם נרצה להדפיס סטרינג ששמור במחסנית: נוכל לשלוח את מציין הפורמט $\%s$ כך יודפס הסטרינד ששמור במחסנית.

אם נרצה להריץ פונקציה מסויימת: נוכל לדרוס את eip ולהחליף אותו בכתובת של הפונקציה. **או** אם הפונקציה שנרצה להריץ שמורה במחסנית, נוכל לשנות את ערך הרגיסטר ebp, מכיוון שהוא מחשב את הכתובת היחסית של הפונקציה, אזי כשהוא יבוא להריץ פונקציה אחר, הוא יחשב ביחס לערך החדש ויריץ את הפונקציה שנבחר.

0.12.1 הגנה:

- **בטיחות זכרון במרחב:** לכל פויינטר p נגדיר כתובת בסיס - b, וכתובת מקסימלית - e. נבדוק בכל שלב האם $b \leq e \leq p$, אחרת - נמנע את הגישה של p לכתובת.

- **הגנה מהרצת קוד עויין:** אם תוקף החדיר קוד עויין למחסנית או לעקרימה, נוכל להגדיר כי אנו לא מריצים קוד אלא אם הוא מאיזור הטקסט.

- **ערכי קנרית:** עוזר לנו עבור מתקפות שמוחקות את ערך ה eip, נכניס את ערך הקנרית בין הערך של ebp והמשתנה הלוקאלי הראשון. כך אם נגלה שערך הקנרית נדרס, נקריס את התכנית.

- **הגנת** $ASLR$**:** הגנה כנגד מתקפת $return\ to\ libc$, בשיטה זו אנו נשנה את סדר הפונקציות בסיפריה באופן רנדומלי, וכך תוקף לא יידע איפה הפונקציה ממוקמת וכיצד להריץ אותה.

- **הגנת** CFI**:** ניצור CFG - גרף של **קריאות עקיפות** לפונקציות (קריאות בעזרת מצביעים) וחזרות מפונקציות, כך בזמן קומפלציה יהיה לנו גרף של התכנית.

הגרף: כל פונקציה תהיה קודקוד. וכל קריאה עקיפה תהיה צלע. בנוסף נשים תוויות על היעד, כך שרק אם התווית מתאימה נרשה קפיצה.

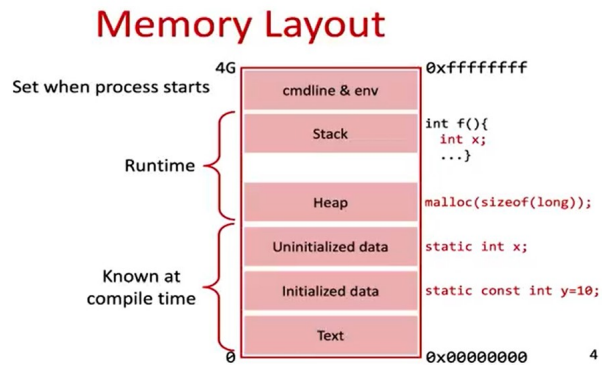
קשתות: נתאים קשת לכל **שורה ספציפית** בקוד. לדוגמה אם פונקציה f קוראת לפונקציה g בשורה 8 , אזי נשים חץ (עבור ערך החזרה - קריאה עקיפה) מהשורה האחרונה של g לשורה 8 של f.

תוויות: אם פונקציה קופצת לכמה מקומות, נתן לכל המקומות אליהם מותר לה לקפוץ את אותה התוית, משום שלפני שהיא קופצת למקום מסוים היא בודקת התאמה של התוית הספציפית הזו.

במהלך ריצת התכנית נוודא את הריצה בגרף, אם אין מסלול כזה - התכנית תעצר.

- **כיצד נכתוב קוד בטוח:**
  - נוודא את אורך הקלט לבאפר אליו אנו נותנים למשתמש לכתוב.
  - שימוש בפונקציות בטוחות - $strlcpy, fgets$ שמקבלות פרמטר של אורך הקלט.
  - להקצות מקום לתו "$\backslash 0$" בסוף סטרינג.
  - בשחרור מצביע נעשה לו השמה ל $null$.

- **מבנה הזכרון המחשב:**



0.12.2 שאלות ממבחנים:

- **מתקפות על ערכי חזרה של פונקציה:** כששואלים אותנו כיצד ניתן לבצע מתקפה שתריץ פונקציה אחרת, הכיוון יהיה לדרוס את eip של הפונקציה ולשים שם את הכתובת של הפונקציה שנרצה להריץ. כך כשהפונקציה תסיים את ריצתה היא תריץ את הפונקציה שאנו רוצים.

- **האם הוספת ערך קנרית יעזור או שימוש ב $ASLR$:**
שימוש בערכי קנרית יעזור רק כאשר המתקפה שלנו כללה דריסה של הערך של ebp או eip.
שימוש ב $ASLR$ יעזור רק כאשר המתקפה שלנו כללה ידיעה מוקדמת על כתובת של פונקציה שהרצנו (לדוגמה פונקציית ספריה), ולא אם השגנו את כתובת הפונקציה מהמחסנית.

0.12.3 אבטחת תכנה ברשת:

- **מתקפת** $SQL\ injection$**:** כאשר אתר מבקש ממנו שם משתמש וסיסמה נוכל לכתוב שם משתמש עם תנאי בוליאני OR שיחזיר לנו $true$ בכל מצב, כך נוכל להיכנס לשרת עם שם משתמש וסיסמה לא נכונים.

- **הגנה נגד** $SQL\ injection$**:** הבעיה הייתה כי הקוד והמידע היו מעורבבים, לכן נצטרך להפריד בניהם.
  - לא נאפשר תוים חשודים, או נחליף אותם בתוים אחרים.
  - נדחה קלטים חשודים.
  - **הצהרות מוכנות מראש:** נפריד את חלק הקוד מהדאטה, נגדיר לשאילתה מהו חלק הקוד ומהו חלק הדאטה שייכנס אליו המידע מאוחר יותר. באופן זה כל מידע שהמשתמש יכניס יוגדר כדאטה ולא כקוד.

- **שדות חבויים:** בתקשורת בין הלקוח לשרת, השרת מעביר ללקוח את פרטי הבקשה שלו כדי שיחזיר לו אותם בבקשה הבאה, המידע יהיה שמור בתוך $session\ ID$, כך שהשרת ייגש למקון ה ID בזכרון שלו ויוציא את הפרטים.
החסרון: כשנעזוב את האתר המידע יימחק.

- **עוגיות:** השרת שולח ללקוח עוגיה עם המידע שלו, והלקוח שומר את העוגיה אצלו במחשב ושולח אותה לשרת בכל התחברות.

- **מתקפת** $sesion\ hijacking$**:** מתקפה שבה התוקף גונב את העוגיה מהלקוח.
  הגנה: נגדיר לעוגיות תוקף קצר, ונחדש אותן בתדירות יותר גבוהה.

- **מתקפת** $SCRF$**:** התוקף גורם לקרבן להתחבר לאתר מזייף ששולח אותו ע"י תמונה לאתר המקורי, בעת החיבור לאתר המקורי נשלחת עוגיה, כך התוקף מתחבר עם עוגיה מקורית.
  הגנה באמצעות שדה ה $referer$**:** שדה זה מצביע על האתר ממנו הגענו לאצר הנוכחי, נוכל להגדיר שאם הוא שונה מהאתר המקורי לא נאשר את הגישה.
  הגנה באמצעות $Secretized\ Links$**:** נגדיר לכל עוגיה סטרינג אקראי שיקשר אותה לאתר, כך תוקף יצטרך להוסיף את הסטרינג הזה כדי להיכנס.

0.12.4 $JavaScript$:

- האתר כתוב ב $JavaScript$, וכאשר לקוח מבקש להציג את האתר, **הלקוח** מריץ את הקוד אוטומטית (ולא השרת כמו בדף דינמי).
  גישת SOP**:** דפדפן יתיר לסקריפט מאתר A לגשת לנתונים באתר B רק אם A, B מאותו הדומיין.

- **מתקפת** XSS**:** נועדה לעקוף את SOP, רעיון המתקפה הוא לגרום לדפדפן של המשתמש לחשוב שהסקריפט הזדוני הגיע מאתר מאובטח. יש שתי גישות למתקפה.
  סקריפט מאוחסן: התוקף שותל את הסקריפט באתר הפגיע, כך שהוא יישלח למשתמש מהאתר האמיתי ולכן יזוהה כמאובטח.
  סקריפט משוקף: התוקף שותל את הסקריפט הזדוני אצל המשתמש (המשתמש גולש באתר זדוני שמפנה אותו לאתר פגיע), ומקבל אותו בחזרה מהאתר הפגיע, כך הוא בטוח שקיבל מהשרת סקריפט בטוח.

- **הגנות נגד מתקפת** XSS**:** נסנן סקריפטים שלא נשלחו מהשרת. אך לא תמיד נצליח לעלות על הסקריפטים הללו. לכן הפתרון הוא לדחות בקשות חשודות.

0.13 מציאת חולשות:

0.13.1 אנליזה סטטית:

- **אנליזה סטטית:** מתבצעת בדיקת הקוד ללא הרצת התכנית (באופן סטטי).

- $Flow\ Analisis$**:** נגדיר ערכים בתכנית כחשודים (עשויים להישלט ע"י תוקף) ואמינים (אסור שישלטו ע"י תוקף), כך שקלטים המתקבלים מהמשתמש חשודים. לאחר מכן נבדוק אם מתקיימת השמה של ערך חשוד לערך אמין, אם כן - תיתכן פרצת אבטחה.
  עבור כל שורת קוד נגדיר אילוץ, לאחר מכן נבדוק אם ניתן לספק את האילוצים, אם לא - תתכן פרצת אבטחה.

- **הגדרת ערכים לפונקציה** $printf$**:** אם הפונקציה מופיעה ללא מצייני פורמט מתאימים אזי נגדיר את המשתנה $x \leq untainted$. אחרת, אם יש מצייני פורמט - לא נגדיר כלום.

- $Flow\ sensitivity$: שיטה זו נועדה להפחית התראות שווא. הרעיון הוא לתת ערך **לכל השמה** ולא לכל משתנה , באופן זה אם ההשמה האחרונה של המשתנה הייתה חוקית, הוא יקבל ערך חוקי ולא תהיה אזעקת שווא.
  מתי נשתמש: כשיש כמה השמות שונות למשתנה אחד.

- $Path\ sensitivity$: שיטה זו נועדה להפחית התראות שווא. הרעיון הוא לתת לכל לכל אילוץ מסלול משלו. כך אם יש לנו כמה אילוצים והאילוץ חוקי, אנו לא נתריע סתם.
  מתי נשתמש: כשיש כמה מסלולים אפשריים בקוד - תנאים לוגים.
  בהוכחה: נראה כי **על כל מסלול** האילוצים מתקיימים.

- $Context\ sensitivity$: שיטה זו נועדה להפחית התראות שווא. הרעיון הוא להפריד בין קריאות שונות לאותה הפונקציה - נגדיר משתנה **לכל קריאה,** בנוסף נבדיל בין ערכי קריאה לפונקציה - העברת ארגומנטים (-), וחזרה מפונקציה $x = func$ (+).
  מתי נשתמש: כשיש כמה קריאות לאותה הפונקציה.

- $information\ Flow$: שיטה זו באה לטפל במקרים של פספוס פרצות. במקרה של זרימה בין מצביעים נגדיר את הזרימה להיות דו כייונית, כך לא נפספס השמות שמשנות את הערך אח"כ.
  מתי נשתמש: כשיש לנו השמה למצביעים.

- **זרימה לא מפורשת -** $Implicit\ Flows$: שיטה זו באה לטפל במקרים של פספוס פרצות. נוסיף משתנה pc לכל שורה, וכל שורה שהגענו אליה דרך משתנה חשוד תוגדר גם כחשודה $(pc_i = tainted)$. בנוסף נוסיף אילוץ של זרימה מהשורה - למשתנה (x) באותה השורה $(pc_i \leq x)$.
  מתי נשתמש: כשיש השמה שהיא לכאורה סטטית, אך המידע מגיע ממשתנה.

0.13.2 הרצה סימלית ו $Fuzzing$:

- **הרצה סמלית:** ניתן לכל משתנה חשוד ערך סמלי, ונבדוק בכל תנאי את כל הערכים שמשתנה זה יכול לקיים בתנאי, לפי זה נבנה את עץ התכנית. אם יש מסלול בעץ שלא מקיים את הדרישות - נוכל לעקוב אחר הקלט הבעייתי ולדעת איפה מתרחשת הפריצה.

- **תנאי מסלול:** הוא התנאי שהביא אותנו אל המסלול הנוכחי, נגדיר את התנאי הבוליאני ונבדוק האם ניתן לספק אותו. פיזיביליות של מסלולים: היא בדיקת הסיפוק של מסלולים.

- $Fuzzing$: נספק לתכנית קלטים רנדומלים לתכנית והיא מריצה אותם בכדי למצוא מפגעי אבטחה. קיימים שלשה סוגי $fuzzing$
  - **קופסה שחורה:** נשלח קלטים רנדומלים בלי לדעת שומדבר על התכנית.
  - **דקדוק:** נגדיר דקדוק מסויים ונשלח קלטים לפי כללי הדקדוק.
  - **קופסה לבנה:** נבנה דקדוק לפיו נתקדם עם הקלטים, וגם נייצר קלטים שרלוונטים לתכנית.