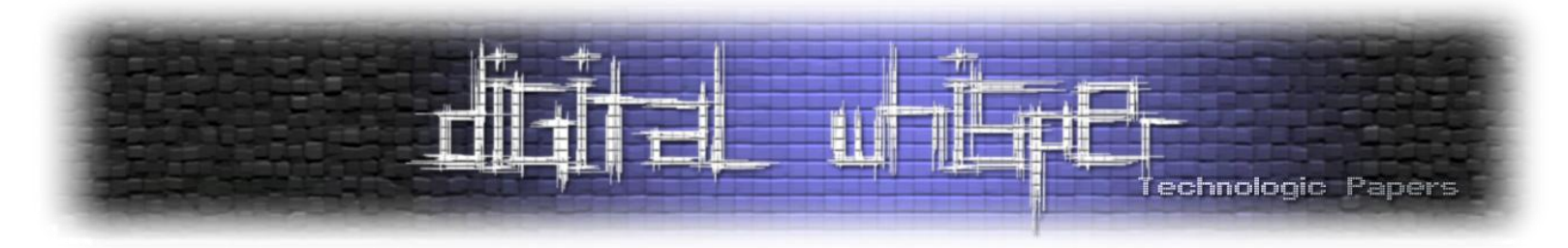

תכנות בטוח – חלק ב'

מאת עידו קנר

הקדמה

מאמר זה הינו חלק ההמשך של המאמר העוסק בנושא "התכנות הבטוח" אשר פורסם בגליון השביעי של Digital Whisper. בחלק הקודם הצגתי מקרים מאוד פשוטים וברורים אודות כמה מגישות בתכנות אשר גורמות לכך שהקוד הנכתב הופך לבעיית אבטחה בבאגים שהוא מעלה.

בחלק זה אנסה לשפוך אור כיצד ניתן להימנע מהבעיות השונות ובכך ב עצם לגרום לתוכנה אותה אנחנו כותבים להיות בטוחה יותר. חשוב להבין כי הבעיות שהצגתי בחלק הקודם מתחלקים בעצם ל-2 חלקים:

- הנחות יסוד שגויות.
- חוסר זהירות בכתיבת הקוד, בלי לנסות להבין את המשמעויות השונות.

גלישות

בשביל למנוע את בעיית הגלישות, בין אם מדובר בגלישות חוצץ או גלישות מסוג אחר, צריך דבר ראשון להבין מה הפעולה שרוצים לבצע, מה הבעיה שיש צורך להתמודד עימה ומה סוג המידע שצריך לטפל בו. כל אלו יעזרו למנוע את הגלישות השונות במידה ותיבחר פעולה נכונה בטיפולם.

גלישת חוצץ

נחזור רגע לדוגמה שהוצגה בחלק הקודם:

```
var
  iNums : array [0..9] of integer;
  ...
  FillChar (iNums[-1], 100, #0);
  ...
  for i := -10 to 10 do
     readln (iNums[i]);
  ...
```

ניתן לראות בדוגמה כי ישנו טווח שמשוכתב על ידינו (בצורה ידנית במקרה הזה) בלי בדיקה האם הטווח חורג מתחום הזכרון המוקצה.

בשפת פסקל קל מאוד לדעת מה הטווח של מערכים (ובכלל, ניתן לדעת את הטווח של כל דבר האפשרי למניה). כך שבקלות ניתן לשכתב את הקוד:

```
var
  iNums : array [0..9] of integer;
  ...
  FillChar (iNums[Low(iNums)], High(iNums), #0);
  ...
  for i := Low(iNums) to High(iNums) do
     readln (iNums[i]);
  ...
```

שתי פונקציות מאוד חשובות בשם High ו-Low מחזירות האינדקס הגבוה והנמוך ביותר במערכים ובכך אנחנו דואגים כי גם אם ישתנה הגודל של המערך, הקוד שלנו ידע תמיד לטפל בו.

לפני שיתחילו החגיגות, נשים לב כי יש כאן עוד בעיה!

יש כאן בעיה של "גלישת מספרים". הפקודה readln מקבלת עבור כל בקשה, מספר אין סופי של מידע אשר יכול להיות מספרים או סימנים אחרים.

גלישת מספרים

מחרוזת הן בעצם מערך. המהדר ינסה למצוא את הטווח המקסימלי של המחרוזת כאשר משתמשים ב - readln, אך בדוגמה למעלה הוא לא יצליח למצוא אחת. הסיבה היא שמספרים הם לא מחרוזות ולא ניתן לדעת בצורה פשוטה מה הטווח המקסימלי של כל טיפוס של מספר, כך שאין הגנה על פני גלישה.

למחשב קיים גודל מקסימלי של זיכרון אותו הוא יכול להקצות בצורה טבעית עבור מספרים ובכלל למידע (בעבודה עם אוגרים). המחשב מסוגל לתת מעט מקום עבור המידע. אומנם ניתן להגדיל את הכמות על ידי חיבור "ידני" של כמה אוגרים שיתנהגו כמידע אחד, אבל חשוב להבין כי יש הגבלה. יותר מזה הרבה פעמים אין צורך לכל הזיכרון, כלומר לפעמים יש צורך בביט או בית בודד ולא מילה או גודל שהוא גדול יותר (למשל בערך בוליאני, בו יש צורך ערך של 0 או 1 ולא מעבר), כך שיש צורך לדעת להתמודד עם שני המצבים הקיצוניים האלו.

בדוגמה של "גלישת החוץ" יצרנו עוד סוג של גלישה, הפעם היא של המספרים, היות והקלט שנוצר שם יכול להכיל כאמור כל גודל של מספר.

מה ניתן לעשות בנידון? בראש ובראשונה אפשר לשקול עבודה עם מחרוזות אשר מוגבלות באורך שלהן, ובמידה ויש צורך בסימנים למספרים (כלומר הסימנים מינוס או פלוס). לאחר מכן נשתמש בפרוצדורה val

או בפונקציה StrToInt בכדי לנסות להמיר את המחרוזת למספר שלם (מתוך ההדגמה בתעוד של FPC):

```
Program Example74;

{ Program to demonstrate the Val function. }
Var I, Code : Integer;
begin
  Val (ParamStr (1),I,Code);
  If Code <> 0 then
    Writeln ('Error at position ',code,' : ',Paramstr(1)[Code])
  else
    Writeln ('Value : ',I);
end.
```

באמצעות השימוש ב-val ניתן לדעת מתי יש שגיאה בניסיון ההמרה ובאיזו נקודה היא התרחשה . כאשר משתמשים בפונקציה StrToInt חשוב לזכור שצריך לתפוס חריגה שיכולה להעלות כאשר נמצאת שגיאה בעת ניסיון ההמרה.

עוד דרך לקבל קלט מספרי היא ליצור מערכת קלט שבה יש שליטה מלאה על כל תו שנכנס מההתחלה למערכת, כולל האורך והמיקום:

```
program MyReadln;
uses CRT;

procedure MyIntReadLn (var Param : Integer; ParamLength : Integer);
var
  Line  : string;
  ch    : char;
  Error : Integer;

begin
  Line  := '';

  repeat
    ch := readkey;
    if (Length (Line) <> ParamLength) then
     begin
      if (ch in ['0'..'9']) then
       begin
         Line := Line + ch;
         write (ch);
       end
      else
      if (ch = '-') and (Length (Line) = 0) then
       begin
         Line := '-';
```

```
          write (ch);
        end;
      end;

    if (ch = #8) and (Length(Line) <> 0) then // backspace
     begin
      Line := copy (Line, 1, Length (Line) -1);
      gotoxy (WhereX -1, WhereY);
      write (' ');
      gotoxy (WhereX -1, WhereY);
     end;
  until (ch = #13);

  val (Line, Param, Error);

  if (Error <> 0) then
    Param := 0;

 writeln;
end;

var
 Num : Integer;

begin
  write ('Number: ');
  MyIntReadLn (Num, 2);
  writeln ('The number is: ', Num);
end.
```

(חשוב להבין כי זו הדגמה, והקוד לא נועד להיות קוד שירוץ בתוכנה קיימת)

הסכנה בגלישות:

הסכנה שיש לנו בגלישות אלה, היא האפשרות שגורמת לקוד לחרוג מהגבולות שהוצבו למידע, ובכך ניתן להריץ כל קוד שתוקף ירצה להריץ לאחר שהקוד ניגש אל הדגלEIP של המעבד.

מניעת שירות

מניעת שירות זו אחת מצורות ההתקפה הקשות ביותר לטיפול ומניעה בגלל:

- ניתן למצוא מצב שבו גם כאשר אין חולשה ספציפית באפליקציה ניתן יהיה לבצע מתקפת מניעת שירות כדוגמת מתקפת מניעת שירות מבוזרת (DDoS), אשר גורמת להרבה בקשות בו-זמנית של משאבים עד אשר לא ניתן יותר להגיש את המשאב המבוקש.

- כאמור, כל משאב מערכת יכול בתורו לגרום לבעיה של מניעת שירות כאשר זה אוזל מסיבות שונות. למשל ניצול הזיכרון עד תומו, ניצול מקום בדיסק הקשיח עד תומו, פתיחת שקעים רבים מידי לתקשורת, ניצול מלא של רוחב פס והרשימה עוד ארוכה.
- מחיקת מידע או קבצי מערכת שונים יכולים בתורם להפוך למניעת שירות לכל דבר ועניין. למשל מחיקת מודול של גרעין המערכת יגרום לכך שלא יהיה התקן מערכת שידע לעבוד עם חומרה אשר כן עבדה.
- שינוי הגדרות או מחסור בהן יכול לגרום למניעת שירות, כאשר למשל צריך להקצות יותר זיכרון ממה שמורשה בהגדרות המערכת, או שינוי נתיב הקבצים אשר אותם מחפשים השתנה לנתיב בו אין הרשאות קריאה או שהקבצים פשוט לא קיימים גורם למניעת שירות
- מניעת הרשאות, או עודף הרשאות, יכולים אף הן לגרום למניעת שירות.
- שינוי עבודה של פונקציות מערכת שונות (פונקציות API) כאשר התכנה אינה יודעת לעבוד איתם נכון יגרום גם לבעיה של מניעת שירות.
- בברירת מחדל כמעט כל באג גורם למניעת שירות אם מנצלים אותו.

כאשר מביטים ברשימה הלא מלאה הזו ניתן להבין כי כמעט כל דבר יכול להוביל למניעת שירות, ומאוד לא פשוט להתמודד עם הנושא.

עם זאת, כמפתחים יש לנו כוח לנסות ולהתמודד עם בעיות רבות הקשורות לנושא מניעת השירות. נביט שוב בקוד מהחלק הקודם של מניעת השירות:

```
 ...
begin
  while (True) do
    begin
      Getmem (OurPtr, 10);
      OurPtr := Something;
    end;
end.
```

הדוגמה מקצה זיכרון בצורה אין סופית, אך במקום לשחרר את הזיכרון הישן, היא מאבדת את ההקצאה הישנה ושומרת במקום את ההקצאה החדשה, ובכך עד שהתוכנה לא תסיים את הפעולת הריצה, יגמר הזיכרון במחשב. כאשר התכנה תסיים את הריצה התגובה של מערכת ההפעלה שונה ממערכת אחת לשנייה. ב-Microsoft Windows יהיה למשל צורך לאתחל את המערכת בשביל לקבל חזרה את הזיכרון שלא שוחרר לבד.

ישנן הרבה דרכים ושיטות להתגבר על הבעיות האלו, אך כולן יגרמו אף הן לסוג של מניעת שירות כזו או אחרת, כי הן או ימנעו עוד הקצאת משאבים, או שהם ישחררו משאבים כך שבכל מקרה משהו יפגע מזה. בשביל לפתור את הבעיה צריך לתת מענה לצורך מסוים, היות ולא ניתן לתת פתרון שמתאים לכל מצב

הזרקות קוד

ישנם הרבה דרכים וצורות להזריק קוד לתוכנה. הבעיה הגדולה של הזרקת קוד היא שניתן להריץ בעצם קוד שהתוקף רוצה להריץ ובכל להשיג תוצאה שהיה אסור לו להשיג

בדוגמהשהוצגה בחלק הקודם:

```
User Input:
  Please enter your name: a' OR 1=1

  ...
  write ('Please enter your name: ');
  readln (sName);
  Query1.SQL.Add ('SELECT Password FROM tblUsers WHERE Name='#32 +
sName + #32);
  ...
```

אפשר לראות שיש קבלת קלט ממשתמש, אשר מוזן לשאילתה כפרמטר. הבעיה בקוד הוא שהקוד אינו בודק האם התווים שהתקבלו מהשתמש מורשים, או האם מצב ברירת המחדל שלהם, במידה והקלט כן תקין, אינו פוגע בשאילתא עצמה.

כאשר מדובר ב-SQL ישנה דרך אחת מאוד מקובלת כאשר כותבים קוד בצורה ישירה, וזה להשתמש ב-bind parameters. הכלי שנקרא bind parameter מאפשר למפרש של מנוע מסד נתונים לבצע פעולת escaping (פעולה אשר בעצם מטפלת בתווים בעייתיים על ידי הפיכתם לקוד לא בעייתי שעדיין מכיל את אותו המידע בדיוק) וכך יש הגנה מלאה על המידע שמוזן לקוד.

ישנם 2 סוגים של bind parameters:

1. Anonymous Bind
2. Named Bind

Anonymous bind מאוד נפוץ ונתמך בד"כ בכל המנועים השונים, בעוד ש-Named Bind פחות נפוץ אך לדעתי האישית עדיף.

Anonymous bind נראה ככה:

```
Query1.SQL.Add ('SELECT Password FROM tblUsers WHERE Name=?');
Query1.param(0).AsString := sName;
```

סימן השאלה הוא בעצם הפרמטר שמוחלף בתוכן. חשוב להבין כי יש שאילתות כאשר מדובר במחרוזת, אשר ידרשו אותו מוקף בתווי מחרוזת, ויש כאלו שלא, אך השימוש בסימן השאלה פרט לכך נשאר זהה.

Named Bind נראה ככה:

```
Query1.SQL.Add ('SELECT Password FROM tblUsers WHERE Name=:name');
Query.param(':name').AsString := sName;
```

הנקודתיים ומיד לאחריהם שם כלשהו (במקרה שלנו name:) הם בעצם ה-Named Bind, אשר מאפשר לתת בפעם אחת שם לערך מסוים, ואז כל מקום בו השם נמצא הוא מוחלף בערך המבוקש.

ההבדל בין Named Bind ו-Anonymous הוא ש-Anonymous מבוסס מיקום, ולכן דורש יותר עבודה כאשר יש יותר ממקום אחד בו צריך להשתמש במידע.

כאשר אין שימוש בשאילתות, יש כמה דברים שחשוב לקחת בחשבון:

1. מה התווים המורשים- תמיד להעיף תו לא מורשה ובכך מראש מורידים הרבה מאוד סיכונים.
2. האם התבצע escaping על המידע- אלא אם יש עניין לתת למשתמש להריץ כל דבר אפשרי, יש לבצע פעולות escaping בהתאם לסוג הפעולה שעושים , היות וכל פע ולה דורשת התנהגות ודרכים שונות לביצוע escaping.

בדוגמהשהוצגה בחלק הקודם של הזרקת html:

```
url: http://example.com/?paramA=a”><script>alert('bla');</script>

...
<input type=”text” name=”paramA” value=”<%template
write(var['paramA']); %>” />
...
```

אפשר לראות שיש כתיבה ישירה של paramA, דבר אשר גורם לכך שאפשר להזריק Javascript למשל, וליצור בעיית XSS (המוכרת גם בשם Cross Site Scripting).

במידה ו-paramA היה מגיע דרך Javascript כדוגמת עבודה עם טכנולוגיית AJAX (או דרך HTTP או דרך XML), היה ניתן להשתמש בפונקציות Javascript אשר מאפשרות escaping כדוגמת:

```
escape(“<script>alert('bla');</script>”);
```

התוצאה תהיה:

```
"%3Cscript%3Ealert%28%27bla%27%29%3B%3C/script%3E"
```

בדוגמה למעלה אבל, יהיה צורך להשתמש בפונקציית escaping של המנוע בו נעשה שימוש.

חשוב להבין כי במידה וצריך להכניס את הפרמטר ל-URI כלשהו, יש גם פונקציות אשר יודעות לבצע את הפעולה המתאימה לזה.

במידה ואין צורך בלאפשר סוגריים משולשים, או גרשיים וכו', רצוי להסיר אותם הרבה לפני שיש שימוש במידע עצמו למשל בצורה הבאה:

```
function trim_chars(const s : AnsiString; AllowedChars : TCharset) :
AnsiString;
var
  I : integer;
begin
```

```
  Result := '';
  for I := 1 to Length(s) do
    begin
      if s[i] in AllowedChars then
        Result := Result + s[i];
    end;
end;
```

הפונקציה למעלה עוברת על כל תו (יש לשים לב שהיא לא בנויה לעבוד עם תווים שה ם מעל בית בודד) ובמידה והוא מופיע ברשימה של AllowedChars, אז היא שומרת את התו, ובסוף היא מחזירה את כל התווים לפי הסדר שהתקבלו רק ללא התווים הבעייתיים.

עוד דרך לבצע את אותה הפעולה היא להשתמש באפשרות ה-replace של regular expression בצורה הבאה:

```
s ~= s/[^a-zA-Z\s]//;
```

הקוד למעלה הוא קוד perl אשר מאפשר את כל התווים שם A עד Z כולל אותיות קטנות או רווח. כל תו שאינו ברשימה לא ישמר בעצם וכך השגנו את אותה הפעולה . חשוב להבין כי שימוש ב-Regex (כלומר Regular Expression) איטי יותר מהשימוש בפונקציה לינארית שהצגתי למעלה בשפת פסקל.

Format String

הבעיה הגדולה ב-Format String היא בכך שכאשר מבצעים פעולת concat, כלומר חיבור עוד מחרוזת עם מחרוזת המכילה קוד של Format String בתוכה, אפשר לנצל את הדבר בשביל להגיע לדגל ה-EIP במכונה, ובכך להריץ קוד במקום לשים מחרוזת שרוצים.

כאשר מדובר בשפת C, ישנם 2 דרכים עיקריות לחיבור מחרוזות:

1. עבודה עם strncat- חיבור מחרוזות עם הגבלה על אורך המחרוזת.
2. עבודה עם Format String- כלומר חיבור מחרוזת למחרוזת אחרת באמצעות Format String.

כאשר עובדים עם Format String מההתחלה, מומלץ מאוד להשתמש באפשרות זאת גם בשביל לחבר עוד מחרוזת כלומר במקום הדוגמה של החלק הקודם:

```
...
char * some_variable;
...
printf("Hello %s" + some_variable, "world");
...
```

צריך להשתמש ב:

```
...
char * some_variable;
...
snprintf("Hello %s%s", 32, "world", some_variable);
...
```

השימוש שבוצע בדוגמה עם snprintf מוסיף כפרמטר שני את האורך המקסימלי של מחרוזת. בצורה כזו, יש הגבלה במידה ו-some_variable יצור מחרוזת ארוכה יותר מ-32 תווים.

ככלל, ב-C מאוד מומלץ להשתמש בכלים אשר לא רק עובדים עם זיכרון דינאמי, אלא שניתן להגביל את גודל העבודה שם, ובכך גם יהיה קל יותר להגביל בעיות של גלישות חוצצים למיניהם.

מיתוסים והנחות

הבעיה העיקרית שיש עם מיתוסים והנחות היא שכאשר הן מתרחשות, מאוד קשה לכתוב קוד טוב, יעיל וכמובן בטוח. היות ויש כל כך הרבה דברים המפריעים לפיתוח (דוגמאות על נושאים כאלו כתבתי בחלק הקודם). חשוב לזכור ולהבין כי אין תוכנה ללא באגים, אבל זו הנחה שלצערי עוד לא נכשלה.

סיכום

תכנות בטוח הוא עניין של גישה נכונה. כל הנושאים שהובאו במאמר זה הוזכרו על קצה המזלג, בשביל לנסות לתת לכם הבנה איך גישה וחשיבה שונה על פיתוח יכולה לשנות לגמרי את התשובה לשאלה האם הקוד יוכל לשמש תוקפים לבצע פעולות על המערכת או לא
כמתכנתים חשוב תמיד להבין את נקודות הכשל השונות ולראות היכן אפשר להשתפר.