The Husky Code

מאת גל תא שמע

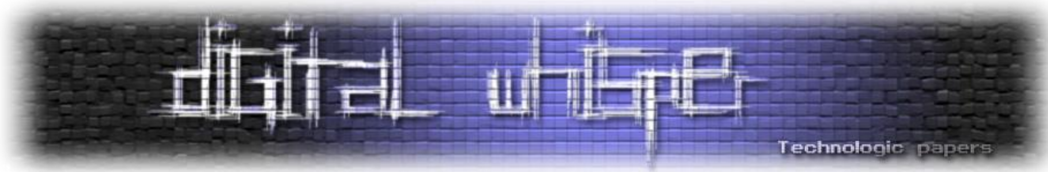
הקדמה

בזמן האחרון התחלתי להתעסק מעט ב-javascript. ככל שעבר הזמן למדתי לאהוב את התכונות המוזרות של השפה, אמנם לרוב הן לא שימושיות אך לפעמים הן מאפשרות דברים נחמדים. התכונות האלו אפשרו לי ליצור את היצירה הזאת. הגיתי את הפרויקט לאחר שחבר הראה לי כל מיני סוגים של אובפוסקציות ב-javascript. המאמר של Ender (גיליון 56), נתן לי את ההשראה לכתוב על הפרויקט. את הפרויקט עשיתי ביחד עם אחי, נועם תא שמע. נהננו מכל רגע (:

הפרויקט

מטרת הפרויקט היתה ליצור קוד javascript המורכב מקבוצה מוגבלת של אותיות. הקוד צריך להיות בנוי כך שנוכל לכתוב אותו בצורות, ולא יחייב מבנה פיזי ספציפי. על הפרויקט נועם ואני עבדנו בערך 24 שעות, את שאר השבועות אחר-כך השקענו בניסיון להסביר מה לעזאזל עשינו ולמה זה עובד ☺.

התוצר הסופי הוא ה"האסקי הסקי" וקומפיילר ההופך את כל תהליך היצירה של ההאסקי שבמאמר לאוטומטי.



כתיבת javascript חלקי עם אוצר מילים מוגבל

בפרויקט שלנו החלטנו להשתמש רק באותיות A,S,C,I (ובסימנים), אך יש להבין שמדובר בהחלטה שרירותית, היינו יכולים לבחור כל קבוצה אחרת של אותיות והקוד עדיין אמור לעבוד. אם היה מדובר בשפה של בני אנוש, כנראה שלא היינו מצליחים לתקשר אחד עם השני, למזלנו מדובר בשפה של מחשבים ולפעמים הם קצת יותר טובים מאיתנו בתקשורת. למען האמת, כנראה שרוב השפות לא היו מאפשרות לעשות את מה שאנחנו מבקשים לעשות אך למזלנו javascript היא שפה מיוחדת בפן הזה.

אותיות ללא אותיות

חלק מהדוגמאות בחלק הבאה בהשראת הפוסט של Patricio Palladino. בדוגמה למטה אפשר לראות התנהגות מוזרה של השפה. בשביל להבין את השורות הבאות צריך להבין עקרון מפתח: כאשר מבצעים חיבור על האובייקט [] הוא קורא באופן implicit לפנקציה toString של האובייקט המשורשר. בפשטות, הוא ממיר את שני האובייקטים למחרוזות ומשרשר ביניהם. התכונה הזאת פועלת גם על כמה אובייקטים אחרים אך היתרון ב-[] הוא שהערך של עצמו הוא מחרוזת ריקה, לכן הוא "שקוף" כאשר משרשרים אותו:

```
// makes the letters a,b,c,d,e,f,I,i,j,l,N,n,O,o,r,s,t,u,y,[,]
[console.log([]+{})         //[Object object
console.log([]+![])        //false
console.log([]+!![])       //true
```

הדוגמאות הבאות קצת יותר מורכבות:

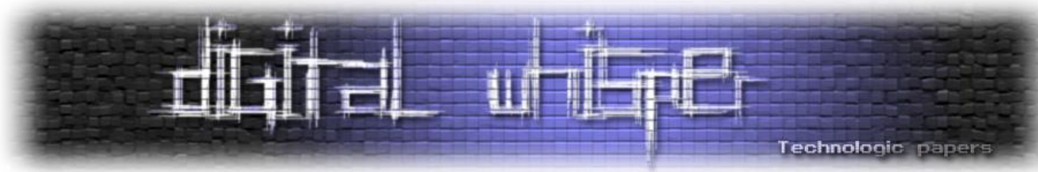
```
console.log(([][+[]])+[]) //undefiend
console.log(+{}+[])        //NaN
console.log(!![]/[]+[])    //Infinity
```

הסבר על 3 השורות האחרונות:

- [][+[]] - אנו מנסים לגשת למערך במקום לא מוגדר, לכן מוחזר לנו: undefined.
- +{} - הסוגריים המסולסלים יוצרים אובייקט, בגלל שאנחנו מנסים להמיר אותו למספר אנחנו מקבלים NaN (לא מספר).
- !![]/[] - מערך ריק הוא שווה ערך ל-0, לכן כאשר נחלק מספר ב-0 (שהוא לא 0 בעצמו) נקבל אינסוף.

נשתמש במה שלמדנו למעלה וניצור את המחרוזת "alert 1", קיבלנו את השורה הבאה:

```
(![]+[])[1]+(![]+[])[2]+(![]+[])[4]+(!![]+[])[1]+(!![]+[])[0]+"(1)"
```



מספרים ללא אותיות

כמובן שזה לא מספיק, אסור לנו השתמש במספרים, לכן יהיה עלינו לייצר אותם כמו שיצרנו את האותיות קודם. לייצר מספרים זאת משימה קצת יותר קלה. הפעם יש לנו שני עקרונות מנחים:

- הפעולה + ממירה אובייקט לערך המספרי שלה.
- האובייקט false שווה ערך ל-0 ו-true ל-1 .

לדוגמה:

```
+[]                                             //0
+!![]                                           //1
!+[]+!![]                                       //2
!+[]+!![]+!![]                                  //3
!+[]+!![]+!![]+!![]                             //4
!+[]+!![]+!![]+!![]+!![]                        //5
!+[]+!![]+!![]+!![]+!![]+!![]                   //6
!+[]+!![]+!![]+!![]+!![]+!![]+!![]              //7
!+[]+!![]+!![]+!![]+!![]+!![]+!![]+!![]         //8
!+[]+!![]+!![]+!![]+!![]+!![]+!![]+!![]+!![]    //9
```

[] הוא שווה ערך ל-false, לכן כשנמיר אותו למספר נקבל 0. ![]! שווה ערך ל-true לכן נקבל 1. גם הביטוי ![]+ שווה ערך ל-true. עכשיו אנחנו יכולים לעמוד בתנאי הראשון שהגדרנו. נכתוב מחדש את הדוגמה הקודמת, הפעם ללא הספרות ונקבל את השורה הבאה:

```
(![]+[])[+!![]]+(![]+[])[!+[]+!![]]+(![]+[])[!+[]+!![]+!![]+!![]]+(!![]+
[])[+!![]]+(!![]+[])[+[]]+"("+(+!+[])+")" // "alert(1)"
```

עוד קצת אותיות

בעזרת שילוב של המספרים והאותיות מלמעלה ניתן להשיג אותיות נוספות. כדי לעשות זאת ננצל את העובדה שניתן לגשת לתכונות של אובייקט בשני דרכים:

1. גישה לאובייקט תכונה.
2. גישה לאובייקט [״תכונה״].

בפועל משמעות שני השורות זהה. יתרה מזאת, פונקציה יכולה להיות תכונה של אובייקט. העיקרון הזה מאפשר לנו לגשת לפונקציות מבלי לכתוב במפורש את שם הפונקציה שאליה ניגש. כדי להמחיש:

```
obj.funcName === obj["funcName"]
```

נוכל להשתמש בשורות הבאות כדי להשיג עוד אותיות. (כמובן שאת כל המחרוזות נבטא כמו שלמדנו קודם).

```
[]["sort"]["constructor"]   //Function
[]["constructor"]           //Array
(![])["constructor"]        //Boolean
(+![])["constructor"]       //Number
{}["constructor"]           //Object
([]+[])["constructor"]      //String
```

כתיבת javascript מלא

כדי שנוכל לכתוב בתחביר המלא של javascript, נצטרך למצוא דרך לבטא לפחות את כל אותיות ה-ASCII. אמנם ב-javascript כל אות מיוצגת ע"י תו utf-16 אבל ה-syntax של השפה עצמה מבוסס על אותיות ASCII ולכן מספיק להשיג רק אותם (למרות שאפשר גם יותר). בפסקאות הבאות נסקור שלוש דרכים שונות להשלים לנו את כל האותיות החסרות. כמובן שכל השיטות מסתמכות על דרך להריץ מחרוזת כ"פקודה".

- escaping בעזרת פונקציה.
- התנהגות מוזרה של מנוע ה-javascript V8 והפונקציה toString.
- escaping טבעי ב-javascript.

escaping בעזרת פונקציה

באופן טבעי, כדאי לנו לחפש את מבוקשנו בפונקציות escaping כמו unescape ו-decodeURI. לשתיהן לא נוכל לקרוא עדיין, כיוון שאין לנו מספיק אותיות. מבין שתי הפונקציות אנחנו יותר קרובים להצליח לכתוב את הפונקציה unescape. להזכירכם, בינתיים יש לנו רק את האותיות הבאות. האותיות הכתומות הן האותיות האותיות הנוספות שהשגנו.

```
A,a,B,b,c,d,e,F,f,g,I,i,j,l,m,N,n,O,o,r,S,s,t,u,y.
```

כדי לקרוא לפונקציה חסרה לנו רק האותיות w ו-p. כדי להשיג את אותה, נוכל להשתמש בפונקציונלית של javascript בהמרה בין בסיסים. השורה הבא תיתן לנו את המחרוזת "p":

```
(25).toString(26)  // p
(32).toString(33)  // w
```

הפונקציה window.unescap עובדת בצורה הבאה:

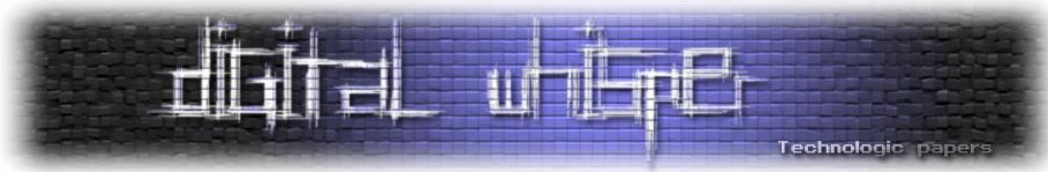
```
window.unescape("%" + HEXA_ASCII_VALUE)
```

[ניתן לקרוא עוד על הפונקציה כאן]

התנהגות מוזרה של V8 (מנוע הjavascript של כרום)

שיטה נוספת מנצלת באג מוזר במנוע ה-V8 של כרום. משום מה כאשר מחשבים מספר גדול ומנסים לייצג אותו בתור מחזרות עם בסיס אי זוגי (כמו 33, או 35), המחשב פולט יותר אותיות מהדרוש. אני חייב לציין שאני לא מבין לגמרי מה הולך שם בדיוק, אתם מוזמנים לקרוא עוד פה. את ההתנהגות הזאת אנחנו יכולים לנצל לקבל את כל האותיות חוץ מ-z (האות ה-36)

```
(1.1536999999997645e-10).toString(35)
0.0000007erxla0hn1fm2728fnw2y3orr9g8u5j1c1ootamq9nkgtvxgt00hsjgp03mm7v09
fw4ul5n80xjjekf1aotqr5mqlbkdqg6suffum5rvutblwwu0f2uxpqr0u0460emjl8g7pe01
e29gf7wfmqwj2jsgr31ewao653dld7vfd8q2ljm7p6rktaogc6pt8piae0jfi5d6k9u8wmeb
8b0j1sxjto035qcwso7od3anm4blotchaqs850ht95x1xnsblbi6fyd10yewenm9bd0ch3je
```



```
uc1av7752tj6nr1w8u3s2mxbv19a3cljw8a23x924fjgksoq1g8tqh6ct099e5nxgsiercop
sk9mqlr5qi048o43di4y7ehgbxt3904549cx7x51ve8xsibvgrygpcxa3u2df6t9qs0c5bkr
p9993d7n4ggipslednjm72186sk2ixdx0cd04g11yamawmdh3ov00vvnnqgh3v1awuuagv76
ycqfaqa2wurln1xnio3c9p1d35xoj6on5icfpec9vj84xwgvghvf9ix2k6vww3huicclkvmo
0rjmvm282ikjdce0ai80vs8v4dc4nlfr5xrxjtaodefwarcybk1p7ixj6pwnyfhmyq28f2ls
mhswn7gwebldnivyf6adumf5yf43nd6b4jm9kah7kcu85dniffs56y9dnp0ylax74r4ffsub
x96ukq5y82r9lb3gxqxxbdv829nlrxhxjeac4ey8vhdlyitxiq4tbu9pmwpx8ulbd64fcune
ejn0yu79flrsft3tbjx4nqk3gg1es0b1ifl2qikssff9oso1ni9ge65i7l05af41rvhcmudb
lw35gna70ld7ksmxonxir14rqv908p4joepjmrys6g2cpm8u87adwx6l8lxmc90fpbssyvkq
b6tq8bh0x4go7vsoowgo66bkgrywkwnduimk77tak9q3qxffu083n9634rt9fir0o7la9ifm
c601kik08l3dva6tomrt4spn8u2tkwxhx5qxsx7c3he3mdi4kv8ppi3c04nayngpo0b468bn
7e2l1nqkbgn4nhnltcew4
```

בכלל, כל אות שחסרה לנו מהאותיות הקטנות (lower) אפשר גם להשיג בעזרת הפונקציה toString.

לדוגמה:

```
(12).toString(36) // "c"
(35).toString(36) // "z"
(17).toString(18) // "h"
(22).toString(23) // "m"
```

השיטה הקלה - implicit escaping

בשיטה הבאה נשתמש ב-"character escape sequences". ב-javascript יש כמה סוגים שונים, ההבדל המרכזי ביניהם הוא בסיס הספירה שבו מיוצגות האותיות. שלושת השיטות לאסקפינג שכזה הן: אוקטלי, הקסהדסימלי, ו-unicode.

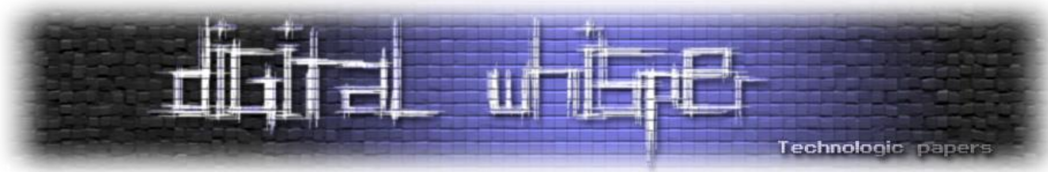
- octal: "\110\105\114\114\117"
- hexadeciaml "\x48\x45\x4c\x4c\x4f"
- unicode "\u0048\u0045\u004c\u004c\u004f"

הקסהדסימלי לא כדאי לנו כיוון שנצטרך להשיג בשבילו את התו x. נשארנו עם unicode ואוקטלי, אני בחרתי להשתמש בשיטה האוקטלית כיוון שהיא משתמשת בפחות תווים מיותרים באופן משמעותי, הדבר נובע מהאפסים המיותרים (לרוב) לאחר ה-u\.

הסבר קצר על השיטה האוקטלית: השם אוקטלי (octal) נובע מהעובדה שהייצוג הוא בבסיס 8, בפרט זה אומר שנשתמש בספרות 0-7 בלבד. היתרונות בשיטה הזאת הם הפשטות מצד אחד והיכולת לכתוב כל סימן מצד שני. להלן קוד שממיר מחרוזת לייצוג האוקטלי שלה:

```
function f(s) {
    var msg = "";
    for(index in s){
        msg += '\\' + s.charCodeAt(index).toString(8);
    }
    return msg;
}
```

[מקום להעשרה על character escape sequences: https://mathiasbynens.be/notes/javascript-escapes]



איך להריץ מחרוזת

הדרך הידועה ביותר להריץ מחרוזות כקוד ב-javascript היא בעזרת הפונקציה eval. למעשה הפונקציה עושה בדיוק את מה שרצינו אך היא משתמשת באותיות "אסורות", לכן לא נשתמש בה. אמנם נוכל ליצור את המחרוזת "eval" אך לא נוכל להריץ אותה כ"פקודה". יש מגוון רחב של פקודות כאלו ואחרות המאפשרות להריץ קוד כמו eval שלא נשתמש בהן מאותה הסיבה. בין היתר: ""=location או setTimeout.

אני אחדד את הצורך שלנו: אנחנו צריכים פונקציה שבהינתן מחרוזת, תריץ אותה, מבלי שהשם שלה יכיל אותיות. יש לציין שלא נוכל להשתמש באותיות שכבר קיבלנו למעלה לאור העובדה שאי אפשר להריץ שרשור של אותיות כ"פקודה". בדיוק בשביל זה בא לעזרתנו בחור נחמד בשם Yosuke HASEGAWA שפיתח שיטה להסתרת קוד javascript ללא שימוש בתווים אלפא-נומריים:

http://utf-8.jp/public/jjencode.html

הרעיון שלו מתבסס על העובדות הבאות:

- כל דבר ב-javascript הוא אובייקט.
- כל אובייקט מממש פונקציה המחזירה את הבנאי שלו.

תאורטית, אם נלך מספיק אחורה בשרשרת הבנאים נגיע לאב הקדמון שאחרי על בניית פונקציות. הוא יהיה מסוג פונקציה כיוון שכל בנאי הוא פונקציה והפונקציה הזאת תהיה בעלת יכולת לעצור פונקציות.

ניקח את ה-flow הבא לדוגמה:

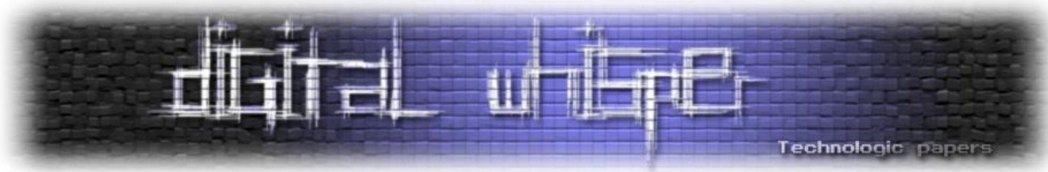
- ◄ אובייקט מסוג מספר. הבנאי שלו:
  - ◄ בנאי מספרים מסוג פונקציה. הבנאי שלו:
    - ◄ בנאי פונקציות מסוג פונקציה.

כלומר לאחר שני קפיצות נוכל להגיע לפונקציה שתפקידה לבנות פונקציות, מעניין. בפועל המימוש של הפונקציה מקבל מחרוזת ומחזיר פונקציה שמכילה את המחרוזת. לדוגמה:

```
// returns a function builder
(function(){})["constructor"]

myFunc = 0["constructor"]["constructor"]("alert(42)")
// function anonymous() {
// alert(42)
// }
myFunc() // will show a popup (42)
```

איך זה עוזר לנו? אנחנו יכולים להריץ כל מחרוזת כקוד. בשילוב עם ה-escaping מבסיס אוקטלי נוכל להריץ כל קוד ללא תלות באותיות המשמשות את הקוד הנראה לעין. כדי שהכל יעבוד, נבנה את



המחרוזות constructor ו-return מראש, משם והלאה נוכל להשתמש בכל אות שעולה בדעתנו כמו בדוגמה הבאה:

```
([]["constructor"]["constructor"]("return'"+"\\"+"1"+"1"+"0"+"\\"+"1"+"0"+"5"+"\\"+"1"+"1"+"4"+"\\"+"1"+"1"+"4"+"\\"+"1"+"1"+"7"+"'"))()
```

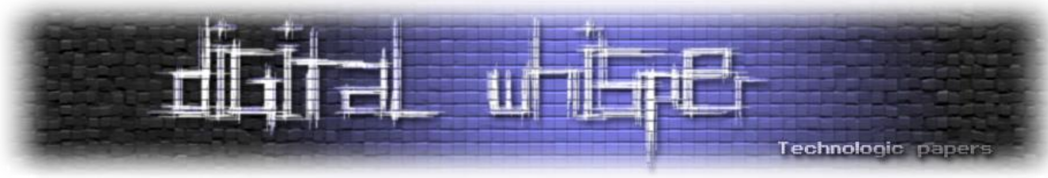
השתמשתי במספרים ובאותיות כדי לחסוך בתווים ולהקל על ההדגמה. לא רציתי לכתוב 34 תווים כל פעם שהספרה 7 נכתבת. נוכל להשתמש באותו רעיון כדי לקצר את הקוד גם בפועל. נשמור משתנים בהתחלה עבור כל הספרות והמילים הנחוצות להתחלה. מימוש אפשרי לזה יהיה:

```
A=+[], I='\'', C=[], S='', CIAA='\\';
// make 0-9 a-f characters
CIA=(!C+S)[A], CCA=A++, CSA=(!C+S)[A], CAI=A++, CII=({}+S)[A],
CCI=((C[+C])+S)[A], CSI=A++, CAC=A++, CIC=(!C+S)[A], CCC=A++,
CSC=({}+S)[A], CAS=A++, CIS=A++, CI=([]+{})[A], CCS=A++, CSS=A++,
CAAA=A++,
// make the constructor and reutrn strings
CC=(!!C+S)[CAI]+CIC+(!!C+S)[CCA]+(!!C+S)[CSI]+(!!C+S)[CAI]+((C[+C])+S)[CAI],
CS=CSC+(({}+S)+S)[CAI]+((C[+C])+S)[CAI]+(!C+S)[CAC]+(!!C+S)[CCA]+(!!C+S)[CAI]+(!!C+S)[CSI]+CSC+(!!C+S)[CCA]+(({}+S)+S)[CAI]+(!!C+S)[CAI],
CA=CAI[CS][CS]; //make the function maker
```

עכשיו רק נשאר לנו לקרוא ל-CA ולהשתמש במשתנים שהגדרנו קודם בשביל המספרים.

Yosuke HASEGAWA עשה את אותו הדבר בדרך קצת שונה. הוא שמר את כל המשתנים בתוך dictionary. בפועל לא מדובר בהבדל גדול, מצד אחד הוא מוסיף לקוד הסופי שלנו הרבה נקודות, מצד שני הוא יצור הרבה פחות משתנים גלובליים ומקטין את הסיכוי להתנגשות עם משתנים אחרים.

כשנועם ואני מימשנו את ה"קומפיילר", השתמשנו בכל מיני שיטות בגרסאות השונות. בסוף בחרנו להשתמש באותה השיטה כמו של Yosuke HASEGAWA ביחד עם שמות מג'ונרטים לאיברים של ה-dictiornary, ככה אנחנו לא מוגבלים לסט אותיות ספציפיות (אצלנו ASCI). אתם מוזמנים להסתכל על הקוד ואפילו לתרום =]



סיכום ביניים

ראינו שיש הרבה דרכים לבטא ביטויים ב-javascript מבלי לכתוב אותם ב״אופן מפורש״. כמובן שלא צריך את כל הפתרונות, אפילו לא את רובם. בפועל יצא שהמימוש שלנו דומה לזה של Yosuke HASEGAWA, כמובן שלא מדובר בהפתעה כיוון שלטעמי המימוש שלו מאד אלגנטי.

כמו שראיתם עד עכשיו, ניתן לחלק את הקוד לחלקים שונים בעלי תפקידים שונים. הנה סקירה של החלקים השונים ותפקידם:

- סגול וסגול מודגש: מגדיר את המשתנים שישמשו בהמשך. החלק המודגש הוא ה-dictionary.
- אדום: יוצר את הפונקציה היוצרת, כמו שמוזכר ב"איך להריץ מחרוזת".
- ירוק: יוצר פונקציה המכילה את המחרוזת הכתומה. כאשר נקרא לפונקציה שיצרנו הדבר יהיה שקול להרצה של הפקודה eval רק שהקונטקסט שלנו יהיה של פונקציה, מה שיאפשר לנו להחזיר ערך. הפונקציה נקראת מיד לאחר ההגדרה שלה והערך יוחזר לתוך הפונקציה הכחולה.
- כחול: פונקציה, כאשר נקרא לה תריץ את הפלט של הפונקציה הירוקה.
- כתום: התו המודגש הוא בעצם המחרוזת "return". אחריו יש את כל האותיות שירכיבו יחד את הביטוי האוקטלי של הקוד שרצינו. כמובן שאם יש אות שיצרנו בהתחלה לא נצטרך לעשות לה escaping ולכן פשוט נוסיף אותה כמו שהיא.

```
A=+[];I='\'';C=[];S='';A={AA:[],IA:(!C+S)[A],CA:A++,SA:(!C+S)[A],AI:A++,
II:({}+S)[A],CI:((C[+C])+S)[A],SI:A++,AC:A++,IC:(!C+S)[A],CC:A++,SC:({}+
S)[A],AS:A++,IS:A++,I:([]+{})[A],CS:A++,SS:A++,AAA:A++};A.I+=A.I;A.AA=A.
I+A.I+A.I+A.I+A.I;A.IAA='\\';A.C=(!!C+S)[A.AI]+A.IC+(!!C+S)[A.CA]+(!!C+S
)[A.SI]+(!!C+S)[A.AI]+((C[+C])+S)[A.AI];A.S=A.SC+(({}+S)+S)[A.AI]+((C[+C
])+S)[A.AI]+(!C+S)[A.AC]+(!!C+S)[A.CA]+(!!C+S)[A.AI]+(!!C+S)[A.SI]+A.SC+
(!!C+S)[A.CA]+(({}+S)+S)[A.AI]+(!!C+S)[A.AI];A.A=A.AI[A.S][A.S];A.A(A.A(
A.C+I+A.SA+A.IAA+A.AI+A.AS+A.CC+A.IC+A.IAA+A.AI+A.IS+A.SI+A.IAA+A.AI+A.I
S+A.CC+A.IAA+A.AS+A.CA+A.IAA+A.CC+A.CS+A.IAA+A.AI+A.AI+A.CA+A.IC+A.IAA+A
.AI+A.AS+A.CC+A.IAA+A.AI+A.AS+A.CC+A.IAA+A.AI+A.AS+A.CS+A.IAA+A.CC+A.CA+
A.IAA+A.AI+A.CA+A.CC+A.IAA+A.AI+A.AS+A.AI+A.IAA+A.AI+A.CC+A.CS+A.IAA+A.A
I+A.AS+A.AI+A.IAA+A.AI+A.IS+A.CC+A.SA+A.IAA+A.AI+A.AS+A.CC+A.IAA+A.CC+A.
CA+A.IAA+A.AI+A.SI+A.CS+A.IAA+A.AI+A.AS+A.CA+A.IAA+A.AI+A.AS+A.AI+A.IAA+
A.AI+A.IS+A.AC+A.IAA+A.AI+A.IS+A.CA+A.IC+A.IAA+A.AI+A.IS+A.SI+A.IAA+A.CC
+A.AI+A.IAA+A.CC+A.CS+A.IAA+A.AS+A.AI+I)())()
```

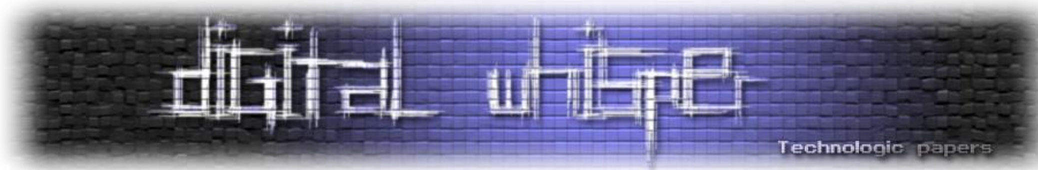
בפועל הריצה תראה כך:

בשלב הראשון, נחבר את כל התווים הכתומים ונקבל את הפלט הבא:

```
"return'a\154e\162\164\50\47\110e\154\154\157\40\104\151\147\151\164a\15
4\40\127\150\151\163\160e\162\41\47\51'"
```

בשלב השני, ניצור פונקציה ונריץ אותה. נקבל את הפלט הבא:

```
"alert('Hello Digital Whisper!')"
```



<u>בשלב האחרון</u>, ניצור עוד פונקציה המכילה את הקוד הנ״ל ונריץ אותו :)

איך להכניס תמונה לקוד

נתחיל מהתמונה. תחילה נצטרך להמיר את התמונה לתבנית, התבנית תורכב משני סימנים (אצלי התו # ורווח). אם היה זה עולם מושלם היינו יכולים פשוט להחליף כל סולמית באות מהקוד המקורי, לצערנו הקוד בצורה הזאת עלול לא לעבוד. javascript לא מאד נוקשה בנהלים לירידת שורה למעט שני מקרים, ירידה באמצע שם משתנה וירידה באמצע מחרוזת. הפתרון הוא פשוט:

1. נחלק את הקוד שלנו לחלקים הקטנים ביותר.
2. נחלק את התבנית לקבוצות של חלקים זהים.
3. נחליף את הקבוצות מהתבנית בקבוצות מהקוד. אם ביטוי הjavascript ארוך מהמקום בתבנית, נשים במקום הקצר ביטוי חסר משמעות כמו הערה /**/ ונקווה שהמקום הבא יהיה ארוך יותר.

לדוגמה התבנית:

```
  ###############   ###############   ###
  ###############   ###############   ###
  ###               ###         ###   ###
  ###               ###         ###   ###
  ###         ###   ###############   ###
  ###         ###   ###############   ###
  ###         ###   ###         ###   ###
  ###############   ###         ###   ###############
  ###############   ###         ###   ###############
```

אם נחלק את הקוד שלנו לחלקים נקבל:

```
["A", "=", "+", "[", "]", ";", "I", "=", "'''", ";", "C", "=", "[", "]",
";", "S", "=", "''", ";", "A", "=", "{", "AA", ":", "[", "]", ",", "IA",
...]
```

עכשיו ביחד:

```
  A=+[];I=''';C=[   ];S='';A={AA:[]   ,IA
  :(!C+S)[A],CA:    A++,SA:(!C+S)[A   ],
  AI:               A++         ,II   :({
  }+S               )[A         ],    CI:
  ((C         [+C   ])+S)[A],SI:A++   ,AC
  :           A++   ,IC:(!C+S)[A],    CC:
  A++         ,SC   :({         }+S   )[A
  ],AS:A++,IS:A++   ,I:         ([]   +{})[A],CS:A++,
  SS:A++,AAA:A++}   ;A.         I+=   A.I;A.AA=A.I+A.
```

קוד שמדפיס את עצמו (Quine)

נהוג לומר לפני ריצה: "תתחיל חזק ותגביר את הקצב לאורך הדרך". לאור שהעובדה שהקוד שלנו סוף כל סוף רץ, החלטנו להקשיב למשפט ולהוסיף לקוד שלנו טריק נוסף. היכולת להדפיס את עצמו, או כפי שנהוג לומר Quine. ויקיפדיה מגדירה quine כקוד לא ריק שלא מקבל קלט ומייצר עותק של עצמו כפלט היחיד. בפועל, לא רצינו להגביל את הקוד שלנו לזה, לכן החלטנו לאפשר לו לממש תכונות של quine. בעצם אפשרנו לתוכנה לגשת לקוד של עצמה הנמצא תחת המשתנה quine.

בקהילה נהוג לכתוב תכניות שכאלו בצורה שלא בונה על התכונות הספציפיות של השפה בה הם מומשו. אלא שלאור העובדה שרצינו שהקוד ימלא תפקיד מלבד מלייצר את עצמו, החלטנו להשתמש ביכולות השפה לטובתנו. לכן השתמשנו בעובדה שלפונקציות ב-javascript יש המרה יפה למחרוזת, לדוגמה:

```
function f(){
    console.log(1)
}

console.log(f+'')
// function f(){
//     console.log(1)
// }
```

אם נסתכל על המבנה של הקוד שלנו, נשים לב שלא כל הקוד עטוף בפונקציה לכן החלקים שבחוץ לא יופיעו כשננסה להדפיס את הפונקציה המרכזית. לכן נשרשר את ההגדרה של המילון בהתחלה ואת הקריאה לפונקציה בסוף כדי לקבל את "התמונה המלאה".

הקוד למטה ידע להדפיס את עצמו כל פעם. רק כדי להדגים שהשיטה עובדת ולא מדובר סתם בקריאה ל-alert על אותו הערך כל הזמן, עדכנתי בכל ריצה את המשתנה s. השיטה שהשתמשנו בה בפועל דומה מאד לדוגמה למטה. אפשר לחשוב על המשתנה s כעל ה-dictionary בתחילת הקוד ועל q כמשתנה המחזיק העתק של הקוד.

```
s = "somthing before"
f = function(){
    s += "q";
    alert(s);
    q = "s = \""+s+"\";\nf="+f+";f();";
};f();
```

בפועל יש עוד כמה בעיות בהדפסה עצמית, הראשית מביניהם היא כזאת: אנחנו רוצים את הקוד של הפונקציה, אך הוא יכיל את התוכן של הפונקציה לאחר חיבור המשתנים ולא ייצג את איך שהפונקציה נראית במציאות.

כדי להסביר את הנקודה אני אדגים: איך שהיינו רוצים שהמחרוזת תיראה:

```
console.log(f) // A.C+I+A.SA+A.IAA+A.AI ...
```

איך היא תראה בפועל:

```
console.log(f) // "return'a\154e\162\164 ...
```

הדרך לפתרון:

1. כדי להתגבר על הבעיה עלינו לשמור את הקוד הפנימי שלנו כמחרוזת. בשביל זה יצרנו רמה שלישית של פונקציות, כאשר הכי פנימית מחזירה את המחרוזת של הקוד (כמו בצורה מהדוגמה הראשונה).
2. עכשיו אחרי שפתרנו את הבעיה הראשונה יצרנו בעיה חדשה, אין ב-javascript תמיכה במחרוזות המתפרשות על יותר משורה אחת, לכן במצב הנוכחי לא נוכל לעצב את הקוד שלנו בצורות. כדי לפתור את הבעיה נוסיף בסוף כל שורה סלאש-אחורי (\), רק כך ה-javascript מאפשר למחרוזת להתפרש על כמה שורות.
3. אך עדיין לא סיימנו, ברגע שנריץ את הקוד הפנימי נזהה שחסרים לנו שוב הסלאשים בסוף. הסלאשים לא נחשבים חלק מהמחרוזת ולכן כאשר המחרוזת נשמרת למשתנה, ה"ירידות שורה" עדיין יהיו שם אבל הביטוי לא יהיה ביטוי תקין, שכן אסור לרדת שורה באמצע מחרוזת ב-javascript. בעצם במקום ירידת שורה וסלאש בסוף תהיה לנו רק ירידת שרה, כמו במצב 2.
4. את הבעיה האחרונה נפתור ע"י הוספת פונקציה פנימית שתדאג להוסיף כל פעם את סלאש בסוף השורה. אני לא אכנס למימוש של הפונקציה לאור העובדה שאני לא בדיוק זוכר מה הולך שם, נועם ואני כתבנו אותה ב-4 בבוקר. בכל זאת אזכיר שאנחנו בפונקציה ברמה 4 (בתוך 3 פונקציות קודמות) לכן כל עניין ה-escaping נהיה מסובך. אם נרצה לכתוב את התו \ נצטרך בפועל לכתוב אותו פעמיים עבור הרמה הראשונה ו-4 פעמים עובר הרמה השניה, 8 פעמים עבור הרמה השלישית וחוזר חלילה.

את הקוד לתמונה הסופית אפשר להשיג כאן. אתם מוזמנים להציץ בפרויקט ה״קומפיילר״ ב-github. אני מקווה שנהניתם מהמאמר. לסיום, מצאתי חידה חביבה מרחבי האינטרנט, אני לא זוכר את המקור: תנסו להכניס למשתנה a ערך ככה שהחלון יקפוץ. בהצלחה!

```
a = <PUT_VALUE_HERE>;

// if something is not equal to itself =]
if (a !== a){
      alert("you win");
}
```