ספר הפרויקט – החבאת קוד JS

Matan Dombelski - 318439981 Arad Zulti - 315240564

15 ביוני 2019

תוכן עניינים

1 על הפרויקט

1.1 רקע

האינטרנט הפך לכלי המעורב בהבטים רבים בחיינו. בעקבות כך, דפדפני האינטרנט תפסו מקום משמעותי והיו חייבים להתשתפר בביצועיהם (שיפור במהירות, חיסכון בזיכרון, וכו'). הצורך בשיפור המהירות של הדפדפנים, הוביל ליצירת אסמבלר לדפדפן ובהתאמה את הנלווה בכך. נוצרה שפת אסמבלי חדשה הנקראת Web Assembly, מהדר (Compiler) המקמפל קבצי C++ \ C לשפת האסמבלי הנ"ל, ואסמבלר היודע להריץ את קבצי האסמבלי.

1.2 מטרה

כיום, יש המון שיטות להרצת קוד זדוני על מחשב מטרה. אחת השיטות הפופלריות הינה לשלוח ל"קורבן" קוד זדוני דרך האינטרנט. כדי להתגונן מכך, דפדפנים רבים מוודאים שהקבצים שהם מקבלים אינם מכילים קוד זדוני. לדוגמא, בעזרת מעבר על הקוד ובדיקתו, הרצתו בתוך Sandbox, בדיקה בעזרת Machine Learning ועוד. אחת השיטות לתקיפה, כדי לעקוף חלק מההגנות, הינה בעזרת שליחת מפענח (Decoder). שולחים לקורבן קוד, המייצר את הקוד הזדוני בזמן ריצה. בכך הקוד הזדוני מוחבא, והזיהוי שלו הופך למסובך יותר.

מטרתנו בפרויקט היא יצירת מפענחים שונים המחבאים את הקוד הזדוני, וקימפול שלהם לשפת Web Assembly. אנו מנצלים את העובדה שטכנולוגיית ה־ Web Assembly חדשה, ולכן ההגנות הקיימות היום על קבצי Web Assembly הן מעטות וחלשות. בפרט רצינו להתמודד עם כך שבכדי להגן, חלק מהדפדפנים מריצים את הקבצים שהם מקבלים בתוך Sandbox וכך יודעים האם הם זדוניים.

2 חקירת קדם

2.1 למה הקוד הזדוני כתוב ב־ JS ולא ב־ ++C?

מוטיבציה: היינו רוצים להצליח להריץ וירוסים קיימים.
מכיוון שהדפדפן מממש סביבה וירטואלית לאסמבלר, פקודות System calls רבות חסומות לקוד ה־ ++C, ולכן חלק מהוירוסים לא יעבדו. לעומת זאת, לוירוסים הכתובים ב־ Javascript אין הבדל בדרך הרצתם, ולכן אין משהו נוסף המגביל אותם מלרוץ.

2.2 איך מריצים קוד הנוצר בזמן ריצה?

כאשר אנו מייצרים את הקוד הזדוני, אנו בסופו של דבר מקבלים מחרוזת. היינו רוצים להצליח להריץ את המחרוזת המייצגת את הקוד הזדוני ב־ Javascript. ישנן שתי דרכים לעשות זאת ב־ Javascript. הדרך הראשונה היא בעזרת הפונקציה eval. היא מקבלת ביטוי, ומחשבת את מה שיש שם. בפרט יודעת לקבל מחרוזת ולחשב את מה שכתוב בתוכה.

```
> eval('console.log("hello from eval")')
  hello from eval
```

פונקציה זו עונה על דרישותינו, אך השימוש בה נתפס בעין רעה בקרב המתכנתים.
דרך שניה שניתן להריץ דרכה מחרוזת, היא בעזרת יצירת אובייקט מסוג פונקציה. ניתן ליצור את הפונקציה ולהריץ אותה באופן הבא:

```
> new Function('console.log("hello from function")')()
  hello from function
```

בדרך זו בחרנו להשתמש.

3 למידת קדם

לימוד הטכנולוגיה הדרושה בפרויקט, הייתה חלק משמעותי ממנו.

3.1 C \ C++

קוד המקור של מהדר ה־ Web Assembly הוא בשפות C \ C++. שפות אלה אנו מכירים ולכן לא נדרשנו ללמוד את השפות עצמן. בשפות אלה כתבנו את המפענחים. כמו שהוזכר קודם, נוצרת לאסמבלר סביבה וירטואלית בדפדפן ולא כל הפקודות הקיימות ב־ C \ C++ ניתנות לשימוש, כתוצאה מכך נאלצנו לוודא כי הפקודות שאנו משתמשים ממומשות.

3.2 Javascript

מה שמנהל את סביבת האסמבלר, וגם קוד התקיפה בפועל הינם בשפת Javascript. שפה שלא הכרנו כמעט, ולכן היינו צריכים ללמוד אותה. חיפשנו תקיפות אמיתיות ב־ Javascript. בנוסף נדרשנו לחקור כיצד ניתן להריץ קוד הנוצר בזמן ריצה.

3.3 Web Assembly

הטכנולוגיה העקרית בפרויקט. נדרשנו ללמוד את כל התוספות (Features) שמהדר ה־ Web Assembly מוסיף לקוד ה־ C \ C++. דרכים לתקשר עם ה־ Javascript, העברת מופעים (Instances) של מחלקות שיצרנו בקוד ה־ C++, שימוש בקבצי מחשב בקוד ה־ C++, ספריות חדשות (כמו תמיכה בגרפיקה של הדפדפן) ועוד.

איור 1: דוגמא לקוד Javascript בתוך קוד C++

```
#include <emscripten.h>

EM_JS(void, say_hello_to, (const char* str), {
    // this is a C comment, in JavaScript code
    console.log("Hello " + UTF8ToString(str));
});
```

4 שלבי הפרויקט

נתאר את השלבים שעברנו בפרויקט עצמו לאחר לימוד הטכנולוגיה. כל שלב שנתאר הוא מפענח בפני עצמו, המייצר קוד בזמן ריצה ומריץ אותו. כל שלב המרחיב שלבים קודמים נמצאים תחת אותה הכותרת. רוב השלבים דרשו מאיתנו הליך חשיבתי ארוך. נדרשנו למצוא דרכי פעולה יצרתיות כדי להגיע לתוצרים רצויים. כחלק מתהליך החשיבה התייעצנו עם אנשים, וחקרנו באינטרנט. בכל שלב נסביר את המוטבציה שהנחתה אותנו בעשייתנו.

תקציר כל השלבים

1. Proof of concept. כתיבה של הקוד הזדוני כמחרוזת, והרצתה.
2. כתיבה של הקוד הזדוני מוצפן בעזרת צופן AES, פיענוחו בזמן ריצה והרצתו. נעזרנו ב־ DLL חיצוני. שלב שלא התאפשר בסוף.

3. הצפנה של הקוד הזדוני בעזרת צופן החלפה שאנו ממשנו.
4. הצפנה של הקוד הזדוני בעזרת צופן AES שאנו ממשנו.
5. שמירה של הקוד הזדוני בקובץ נפרד.
6. שמירה של טבלת תווים, ושמירה של מערך מספרים המייצג את בניית הקוד ע"י הטבלה.
7. דומה לשלב הקודם, כאשר שומרים את הטבלה בצורה דינאמית.
8. שמירה של מחרוזות משותפות, כדי לא לשמור רק תווים. ובניית הטבלה בעזרתן.
9. יצירת הקוד הזדוני ע"י ספריית אינטרנט.
10. החבאה של הקוד הזדוני בתוך תמונה.
11. החבאה של הקוד הזדוני בעזרת מחלקות וירושות.
12. שילוב בין שלבי ההצפנה, שלבי ה Lookup Table והחבאת הקוד הזדוני בתמונה.
13. שילוב יצירת הקוד בעזרת ספריית אינטרנט, והחבאת הקוד הזדוני במחלקות.
14. תלית הרצת הקוד הזדוני בקלט המשתמש.
15. כמו השלב הקודם, עם הגברת הקושי בזיהוי הקוד.
16. ניצול על השלבים עד כה. שילוב של התלות בקלט המשתמש, החבאת הקוד בתוך המחלקות, ויצירת הקוד ע"י ספריית אינטרנט.

Proof of Concept

4.1 שלב ראשון

מטרה. רצינו לוודא שאנו מצליחים להריץ קוד Javascript שנשלח מהמפענח שלנו הכתוב ב־ C++. כלומר, רצינו להראות ייתכנות בסיסית לפרויקט.
ביצוע. כתבנו בתור *string* את הקוד הזדוני שרצינו להריץ, וניסינו להריץ אותו דרך Javascript. ואכן זה הצליח.

איור 2: קוד ה C++ של השלב

```
#include <emscripten.h>

EM_JS(void, run_code, (const char* str), {
    eval(UTF8ToString(str));
});

int main() {
    char code[] = "alert('Hello there, General Kanobi')";
    run_code(code);
    return 0;
}
```

Code Encryption

4.2 שלב שני

מטרה. כאשר כותבים *string* בקוד הוא נשמר בתחילת הקובץ. הנחנו שלא נוכל ל"החביא" את הקוד בצורה כזו, לכן רצינו להצפין את הקוד בדרכים שונות, ולשמור את הקוד המוצפן והמפתח שלו כ־ *strings*. בזמן ריצה נפענח את הקוד הזדוני ונריץ אותו.
ביצוע. השתמשנו בהצפנת AES. בתכנון מקדים לקחנו את הקוד הזדוני והצפנו אותו בעזרת מפתח אקראי. שמרנו את המפתח והקוד כמחרוזות בקוד של המפענח שלנו, ובזמן ריצה פיענחנו את הקוד הזדוני. מכיוון שהצפנת AES היא יחסית מסובכת נאלצנו להשתמש בספרייה חיצונית המבצעת זאת.
לאחר שסיימנו את שלב זה, ראינו שהוא לא ניתן לביצוע. לא ניתן להריץ DLL חיצוניים ב Web Assembly.

4.3 שלב שלישי

מטרה. כיוון שהשלב הקודם לא עבד לנו, רצינו לממש הצפנה בעצמנו.
ביצוע. בחרנו לממש Substitution Cipher (צופן החלפה). הצפנה זו לא בטוחה אבל קל לממש אותה, ובעיני המתבונן לא פשוט להבין מה ההודעה המוצפנה. כמו קודם, בתכנון מקדים הצפנו את הקוד הזדוני בעזרת מפתח אקראי. שמרנו את המפתח והקוד בתור מחרוזות, ובעזרתן פיענחנו בזמן ריצה את הקוד והרצנו אותו.

4.4 שלב רביעי

מטרה. החסרון כאמור בשלב הקודם, הוא שההצפנה איננה בטוחה. לכן רצינו לחזור להצפנת AES.
ביצוע. לכן מימשנו בעצמו את הצפנת AES. השלב דומה לשלב הקודם, כאשר ההבדל הוא בסכמת ההצפנה שהתמשנו.

Built by Lookup Tables

4.5 שלב חמישי

מטרה. רצינו למצוא דרכים נוספות לשמירת הקוד הזדוני, חוץ מכתיבתו כמחרוזת.
ביצוע. רשמנו את הקוד בקובץ טקסט נפרד, ושלחנו אותו ביחד עם שאר קבצי הקוד. בזמן ריצה קוראים את הקובץ ומריצים את הקוד הזדוני ששם. הבעיה עם הרעיון היא שהקורבן יכול לראות את קובץ הטקסט (כי הוא מקבל גם אותו), ולכן השלב דומה כמו לשמירת הקוד הזדוני בתור מחרוזת.

4.6 שלב שישי

מטרה. השלבים עד עתה לא מייצרים קוד בזמן ריצה, והמחרוזות הינן סטטיות. רצינו לבצע יצירה של קוד בצורה יותר דינאמית, ובכך נגביל את יכולת ניתוח קובץ הקוד. לקחנו השראה מ־ Lookup Tables.
ביצוע. יצרנו מערך של תווים (Chars) של כל האותיות הרלוונטיות (לדוגמא, התו a במקום ה־ 30, התו b במקום 15, התו ; במקום 16). כעת, שמרנו מערך של מספרים המרכיב את הקוד שלנו. בזמן ריצה עברנו על מערך המספרים ובנינו את המחרוזת המייצגת את הקוד הזדוני.

בכל זאת, חשבנו על שלוש חולשות מרכזיות. נרשום בכל שלב, מה הייתה החולשה, ואיך טיפלנו בה.

4.7 שלב שביעי

מטרה. חסרון ראשון. אנו שומרים מערך סטטי של מספרים, וכמו מחרוזת גם הוא נשמר בזכרון בתחילת הקובץ.

ביצוע. שינינו לכך שאנו יוצרים מערך דינאמי, ומאתחלים את התאים (איזה תו לרשום מתי) בזמן ריצה. בצורה כזאת, המערך לא מופיע ברצף בתחילת הקובץ אלה במקומות שונים.

4.8 שלב שמיני

מטרה. חסרון שני, אנו שומרים טבלה של כל התווים הרלוונטים. דבר העלול להראות קצת חשוד.
ביצוע. חיפשנו מילים שמורות שכיחות, שהגיוני שיופיעו בקוד. (לדוגמא $success, try\ again, zero$ ועוד). בצורה כזאת, כתבנו את הקוד בזמן ריצה. לדוגמא, התו הראשון בקוד הוא התו הראשון של המחרוזת השלישית. בעצם מה שעשינו, זה בניית הקוד ע"י שימוש בטקסט שאנו כתבנו.

4.9 שלב תשיעי

מטרה. רצינו לפתח את רעיון ה־ Lookup Tables כך שנוכל לבנות את הקוד ע"י טקסט.
ביצוע. שמנו לב, שניתן בזמן ריצה להוריד סיפריות web עם גרסא ספציפית. בחרנו בספרייה נפוצה ב־ Javascript, ווידאנו שהיא מכילה את כל התווים שאנו צריכים. בתכנון מקדים ביקשנו את הספרייה ובנינו את הקוד הזדוני בעזרתה. שמרנו טבלה דו מימדית A בגודל $2 \times x$, כך שכדי לבנות את הקוד הזדוני בזמן ריצה, צריך לבקש מהספרייה החיצונית את התווים במיקום $A[i,0]$ עד ל־ $A[i,1]$, ולשרשר את הכל ביחד. בזמן ריצה אנו דורשים שנקבל את הסיפרייה בגרסא הספציפית שאנו צריכים, ואז בונים את הקוד ומריצים אותו. ובצורה כזאת, אנו בונים את הקוד ע"י טקסט שאנו לא שומרים אותו.

4.10 שלב עשירי

מטרה. החבאת הקוד הזדוני, ללא ידיעה שהוא קיים (סטגנוגרפיה).
ביצוע. בחרנו להחביא את הקוד הזדוני בתוך תמונה. רשמנו את הערך ה־ ASCII של תווי הקוד הזדוני בתור הערך של הפיקסלים באחד הערוצים. בזמן ריצה, נקרא את התמונה, ונשרשר את התווים של הפיקסלים (כל פיקסל מסמן ערך של תו) . בחרנו איפה לרשום את הקוד בתמונה, בכך שקירוב הרבועיים בין ערך תווי הקוד לערך הפיקסלים המקוריים הוא מינמלי. שמרנו בקוד את האינדקס של התו הראשון בתמונה.

איור 3: דוגמא לקוד שמוחבא בתמונה (בתוך העיגול הכחול, הקוד מוחבא)

Hide the Code in Classes

4.11 שלב אחד־עשר

מטרה. חסרון שלישי. רצינו להקשות על ביצוע Reverse Engineering. לשלבים שעד כה הצגנו, ביצוע Reverse Engineering אינה משימה קשה. לעומת זאת, לעשות Reverse Engineering לקוד C++ קלאסי הופכת להיות משימה מאוד קשה. בין כי יש Destructor וירושות ובין כי יש אופטימיזציות של המהדר. אחת

הבעיות העיקריות היא בעת יצירת ירושות ויצירת פונקציות וירטואליות. כאשר נוצר Virtual Tables קשה מאוד בכל רגע נתון לדעת לאיזה פונקציה עומדים לקרוא.

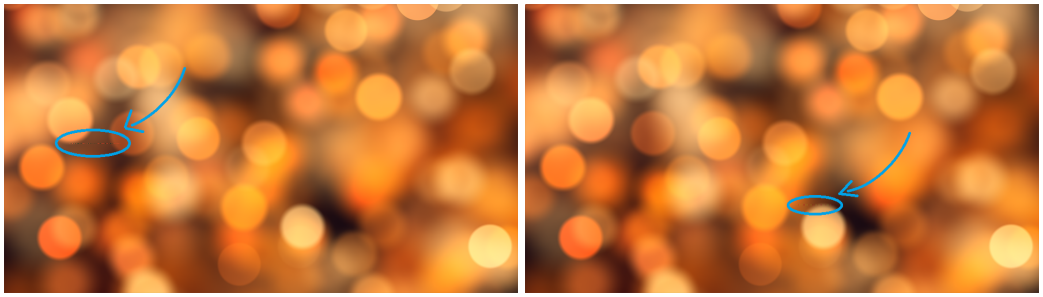
ביצוע. החבאנו את יצירת הקוד הזדוני בתוך המחלקות והפונקציות הוירטואליות שלהן. המחלקות שיצרנו יורשות אחת מהשניה, כדי שתיווצר טבלה וירטאלית כך שיהיה קשה לדעת לאיזה פונקציה קוראים בכל פעם. בתכנון מקדים אנו מגרילים את היררכיית המחלקות, ומייצרים את המחלקות בצורה אוטומטית.

Classes, Lookup Table, Encryption

4.12 שלב שניים־עשר

מטרה. רצינו לשלב בין שלבים שעשינו. **ביצוע.** בהתחלה שילבנו בין שמירה של הקוד בתמונה לבין שיטות ההצפנה. הצפנו את הקוד, ולאחר מכן שמרנו אותו בתמונה. הבעיה שנוצרה היא שהתווים שאנו צריכים לשמור הם אקראיים, לכן התמונה נראת לא טבעית באזור ההחבאה של הקוד. לכן עזבנו את הרעיון. בנוסף, ניסינו לשלב בין Lookup Table לבין ההחבאה בתמונה, אבל באינדקסים ששמרנו ב־ Lookup Table היו גדולים מידי, ודרשנו יותר מבית אחד. ושוב, התמונה הייתה נראת לא טבעית.

איור 4: השוואה בין החבאת הקוד המוצפן ללא מוצפן. (משמאל, החבאת הקוד המוצפן)



4.13 שלב שלושה־עשר

ביצוע. שילבנו בין יצירת הקוד הזדוני בעזרת ספריית אינטרנט, לבין החבאת הקוד בתוך מחלקות. בתכנון מקדים בדקנו איך ניתן ליצור את הקוד הזדוני ע"י ספריית האינטרנט, ויצרנו את המחלקות כך שישמרו את הדרך שבה נבנה את הקוד הזדוני בזמן ריצה.

Sandbox Defense

4.14 שלב ארבעה־עשר

מטרה. בשלב זה התמקדנו בלגבור על כך שהדפדפנים עלולים להריץ את הקוד ב־ Sandbox. כל המפענחים שהצגנו עד כה, מייצרים את הקוד הזדוני ומריצים אותו מיד. לכן, אם דפדפן יריץ את המפענח ב־ Sandbox הוא יצליח לזהות שהוא מריץ קוד זדוני.

ביצוע. התלנו את הרצת הקוד בקלט המשתמש. אנו מחכים ל־ Event שקורה כאשר קורדינטות העכבר הם ברדיוס מסויים מסביב ל־ x, y (המוגרלים באקראי). וידאנו שהם לא קיצוניים מידי במסך, כדי שזה יקרה מהר יותר, ושאכן הן בגבולות המסך. ברגע, שהעכבר הגיע למיקום שציפינו לו, אנו מריצים את הקוד הזדוני. Sandboxes לא יוכלו לגרום להיווצרות הקוד הזדוני, כי בדר"כ אין להם עכבר, וגם אם כן אז הם אינם מזיזים אותו.

4.15 שלב חמישה־עשר

מטרה. כדי להתגבר על השלב הקודם, Sandboxes מנסים להריץ את כל חלקי הקוד שנראים בלתי תלויים, ולכן עלולים לזהות את הרצת הקוד הזדוני.

ביצוע. שמרנו את הקוד בתור מחרוזת, והחסרנו ממנו תו קריטי (לדוגמא סוגר), כך שהקוד לא יהיה תקין בלעדיו. את התו הזה ננחש שהוא קורדינטת ה־ x של העכבר (עם מינוס offset אקראי, וברדיוס קטן). כעת תמיד ננסה להריץ את הקוד שאנו מייצרים. אם העכבר לא בקורדיטה הנכונה, הקוד שלנו לא יצליח להתקמפל ויקרוס. בצורה כזאת, רק כאשר המשתמש יזיז את העכבר לאזור הנכון, נקבל את התו הנכון והקוד הזדוני יהיה תקין וירוץ.

איור 5: המסך מחולק ל־7 חלקים. רק כאשר העכבר יהיה בחלק הנכון, הקוד יהיה תקין ויצליח להתקמפל, מכיוון והתו היחיד שיכול לגרום לקוד להתקמפל הוא '(' (בצהוב ניתן לראות שחסר התו)

```
EM_BOOL mouse_move(int eventType, const EmscriptenMouseEvent *e, void *userData) {
    char table[] = {'1', 'a', ')', 'b', 'v', ',', '9'};

    ostringstream oss("");
    int width = get_window_width();
    int index = ((e->screenX / (width / 7)) + rnd) % 7;
    oss << "(function () {alert('Hello there, General Kanobi')})(";
    oss << table[index];
    run_code(oss.str().c_str());

    return 0;
}
```

Final Stage

4.16 שלב שישה־עשר

מטרה. רצינו לנצל את תכונות כל השלבים, ולייצר מפענח שנהנה מהיתרונות של כולם.

ביצוע. שילבנו בין שלבים 13 ו־ 15. לקחנו את הקוד שמיוצר ע"י שלב 13, והחסרנו ממנו (כמו בשלב 15) תו קריטי. ובזמן ריצה מנחשים את התו על פי קורדינטת העכבר. וכאשר המשתמש יזיז את העכבר לאזור הנכון, נקבל את התו הנכון והקוד הזדוני יהיה תקין וירוץ.

5 תוצרים

התוצר של הפרויקט הינו שיטה להחבאת כל קוד Javascript ממשתמש.

אנו עושים בזאת בעזרת מפענחים. המפענחים מקומפלים ל Web Assembly ויכול להריץ כל קוד Javascript שנותנים להם. כדי לבדוק שהקוד באמת מוחבא וקשה לזיהוי, לקחנו וירוסים, והחבאנו אותם בעזרת המפענחים שלנו. בדקנו בעזרת אתרים שונים, האם הם מזהים שאנו מריצים וירוס, ואף Anti Virus לא הצליח לזהות זאת.

6 עבודה עתידית

במשך כמעט כל הפרויקט המטרה שלנו הייתה דרך להחבאת קוד זדוני. ובמיוחד בשלבי הפרויקט המאוחרים, זאת מכיוון שאנו לא מתכוונים להריץ את הקוד מיד כאשר האתר עולה, אלא תזמנו את ההרצה באופן מתוחכם

יותר. לקראת סופו, הבנו שאפשר לנצל יותר מזה. אתרים רבים משתמשים בשיטות שונות, בינהן בעזרת Obfuscator, כדי להחביא את קוד ה־ Javascript שלהם נגד גניבות של חברות אחרות ומשתמשים שונים. אנו נרצה שאתרים יוכלו להחביא את קוד ה Javascript שלהם.
עבודה עתידית: היינו רוצים למצוא שיטות יעילות יותר ליצירת הקוד הזדוני, שעדיין לא יהיו ניתנות לזיהוי.
בנוסף, אם נרצה להחביא את הקוד בצורה טובה יותר נצטרך להתמודד עם עוד בעיה. הבעיה שיש אצלנו היא שאנו מייצרים את כל הקוד שאנו רוצים להריץ, ולאחר מכן מריצים אותו בפעם אחת. נוצר מצב שיש לנו בזכרון את הקוד המלא, לכן נרצה לייצר את הקוד בחלקים והרצתם בנפרד. כך יהיה קשה יותר להרכיב אותו, אפילו אם יודעים איך הקוד הוחבא.

7 שימוש משתמש בפרויקט

בפרויקט שלנו ניתן להריץ כל שלב בנפרד. נסביר איך להתקין את סביבת העבודה, ואיך ניתן להריץ שלב.

7.1 התקנת סביבת העבודה

הפרויקט מתואם ל־ Windows,

1. תחילה, נדרש להוריד את הקוד שלנו.
   ניתן לבצע זאת ע"י $ git clone https://github.com/aradzu10/WebAssDecoders.git
   או ע"י הורדה של ה־ repository: https://github.com/aradzu10/WebAssDecoders.

2. לאחר מכן, צריך להוריד את סביבת ה Web Assembly. ניתן להסתכל במדריך
   https://emscripten.org/docs/getting_started/downloads.html.
   יש לשים את התיקייה emsdk-master יחד באותו repository עם הפרויקט שלנו.

3. בנוסף, בפרויקט שלנו השתמשנו ב Python 3.
   ניתן להוריד מכאן https://www.python.org/downloads/.

4. אנו עבדנו ב־ IDE של Visual Studio Code. אנו ממליצים להוריד אותו, כדי לאפשר שימוש בכל הסקריפטים שכתבנו. ניתן לעבוד גם בלעדיו.

7.2 הרצת המפענחים

הפרויקט מחולק לתקיות. כל תקייה מכילה את כל הנדרש כדי להריץ את המפענח של אותו השלב.

1. בתקייה ישנו הקובץ code/code.txt. בקובץ זה אנו שומרים את הקוד שאנו רוצים להחביא.

2. לאחר שבחרנו איזה קוד נרצה להחביא, צריך להריץ את התכנון המקדים, את הקובץ src/preprocessing.py (בעזרת Python 3). ה־ script ייצר קובץ src/main.cpp ואת כל שאר הקבצים הנדרשים לשלב הנ"ל.

3. לאחר מכן, נשאר לקמפל את קבצי השלב.
   במידה ולא עובדים עם Visual Studio Code, נדרש להריץ את הקובץ build/build_and_run.bat. ה־ script מקבל שני ארגומנטים. הראשון הוא התקייה של קבצי השלב, השני הוא כתובת מלאה לקובץ ה Javascript שרוצים לקבל (קובץ ה־ output). התקייה היא התקייה שקובץ ה־ main נמצא בה. אנו מניחים שהקובץ ה־ output הוא build/main.js.
   במידה וכן עובדים עם Visual Studio Code. נדרש לפתוח את קובץ ה־ main, ובסרגל הכלים לבחור Terminal –> Run Task... –> Wasm build.

4. ה־ Task או קובץ ה־ bat לא נסגר, מכיוון שהוא מריץ שרת, שצריך לרוץ כל עוד רוצים לגשת לאתר. הכנו עמוד בסיסי המריץ את קוד ה־ Web Assembly. ניתן לגשת עליו דרך localhost:8080/phaseX.html (X זה מספר השלב שמריצים).

7.3 כיצד להרחיב/לשנות את הפרויקט

הפרויקט נכתב בשלוש שפות. נציין מה נכתב בכל שפה, ובאותן שפות צריך להשתמש במידה ורוצים להרחיב/לשנות את אותו החלק.

C \ C++. כתבנו את המפענחים, אשר מקמפלים אותם ל Web Assembly.

Javascript. בשפה זו נכתב הקוד שאנו רוצים להחביא.

Python 3. לכל שלב נדרש הרצה של סקריפט תכנון מקדים. את הסקריפטים הללו כתבנו ב Python.