איך ניתן למנף התקפות Web באמצעות Service Workers?

מאת דניאל אבלס ושי שביט

הקדמה

אי-שם במהלך ההיסטוריה תוקפים נבונים הבינו שלפעמים כדאי לשלוח את המטען הזדוני שלהם באופן עקיף, למשל בקישור המצורף להודעה. אנחנו חשופים היום לאמצעים רבים באמצעותם ניתן לשלוח לנו קישורים כאלה - לרבות: דואר אלקטרוני ותוכנות instant messaging למיניהן.

דפדפנים מודרניים (כמו Chrome, Firefox וכו׳) מציעים מגוון אפשרויות וכלים למפתחים על מנת לשפר את הביצועים, הנראות וחווית המשתמש של אפליקציות ה-Web שלהם. אפשרויות אילו נקראות Web APIs, והן משמשות אבני בניין לאפליקציות Web מודרניות.

במחקר שלנו, בחרנו להתמקד ב-Web API ספציפי בשם Service Workers. גילינו מספר דרכים בהם תוקף פוטנציאלי יכול לשלב חולשות בדרגת חומרה נמוכה-בינונית ובשילוב עם Service Workers לייצר התקפה משמעותית ורחבה הרבה יותר. בתרחישים מסוימים, התוקף עלול להזליג מידע רגיש מהדפדפן של המשתמש אל מקור חיצוני.

עיקרי המחקר שלפניכם הוצגו בכנס BSides Las Vegas 2019, וניתן למצוא את ההרצאה בקישור הבא:
https://www.youtube.com/watch?v=b7V5_xnXA1U

כיצד נולד הצורך ל-Service Workers?

דפדפנים משתמשים ב-JavaScript כמנוע הלוגי של אפליקציות Web, ובכך מאפשרים למפתחים לספק ללקוחות אפליקציות Web אינטראקטיביות. באופן כללי, JavaScript הינה שפה Single Threaded - כלומר היא מריצה פיסות קוד בסדר מסוים, כך שכל פיסת קוד חייבת להסתיים לפני שהבאה תוכל לרוץ.

לכל Tab בדפדפן שלנו יש Thread אחד שמייחד אותו. נהוג לכנותו ה-Main Thread. כאמור זהו Thread יחיד, ומכאן בסיטואציות שהן CPU Intensive, אותו Thread יהיה עמוס ויוביל לחווית משתמש ירודה.
אם כן, כיצד נוכל להריץ מספר משימות תובעניות מבלי לפגוע בחווית המשתמש ושימושיות האפליקציה?

Web Workers

למזלנו, פתרון לבעיה זו ניתן על ידי מפתחי דפדפנים מודרניים באמצעות ה-Web API שנקרא Web Workers המאפשרים להריץ משימות "קשות" בדפדפן ברקע במקביל ל-Main Thread. כך נוכל לעשות שימוש ב-Thread נוסף שייטפל בכל המשימות הכבדות, ובו זמנית להשאיר את ה-Main Thread פנוי לטיפול בכל המשימות הקשורות ל-User Interaction.

לאותם Threads חדשים ברקע יש כמעט את אותן יכולות שיש ל-Main Thread מלבד כמה החרגות. תחילה, אנו מחוייבים לטעון אותם מתוך אותו המקור (Origin) - טעינה ממקור שונה תגרום לכשל בהעלאת ה-Thread.

שנית, לאותו Thread אין גישה ל-Document Object Model. כל רצון לבצע שינוי ב-DOM חייב להיות מבוצע על ידי ה-Main Thread. ניתן לגשר על הפער הזה ע"י שימוש במנגנון ה-postMessage, כלי תקשורת בין ה-Threads השונים.

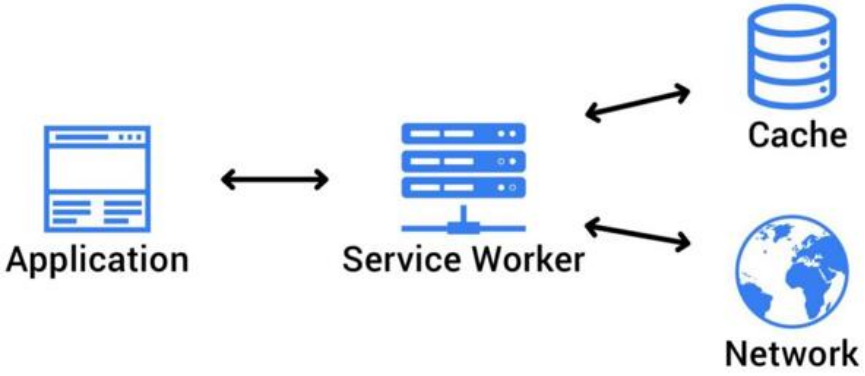
Web Workers גישרו על מגוון פערים שהיו קיימים בעולם ה-Web, אך עדיין נותרו כמה סוגיות ללא מענה הנובעות מהמגבלות הכרוכות בהפעלתם.

ראשית, כל Thread שנפתח כפוף אך ורק לדף שיצר אותו. כלומר, אם נרצה שאותו Thread ישרת מספר עמודים שונים, כל עמוד ידרש לפתוח את ה-Thread בנפרד באופן שעלול לפגוע ב-Maintainability של אפליקציות מורכבות. שנית, ביצועי האפליקציה עלולים להיפגע ולסבול מתקורה מיותרת בשל "צעדי התקנה" המתרחשים בכל פעם שה-Thread חדש נוצר.

Service Workers

כמענה לפערים שנוצרו משימוש Web Workers יצרו מפתחות הדפדפנים API חדש יחסית בשם Service Workers. הם דומים במהותם ל-Web Workers, אך הם בעלי יכולות משופרות ועוצבו כדי לשמש באפליקציות Web מורכבות הפועלות באמצעות מספר רב של עמודים. למעשה, Service Workers משמשים כ-Proxy בין רכיבי האפליקציה הרצים בדפדפן לבין התקשורת עם שרת האפליקציה המרוחק, באופן המאפשר להם יירוט בקשות ותשובות על מנת לעבד אותם. יכולת זו שימושית במיוחד לאפליקציות הרצות בדפדפן כאשר הוא במצב offline - ה-service worker יכול לשמור אלמנטים מרכזיים באתר

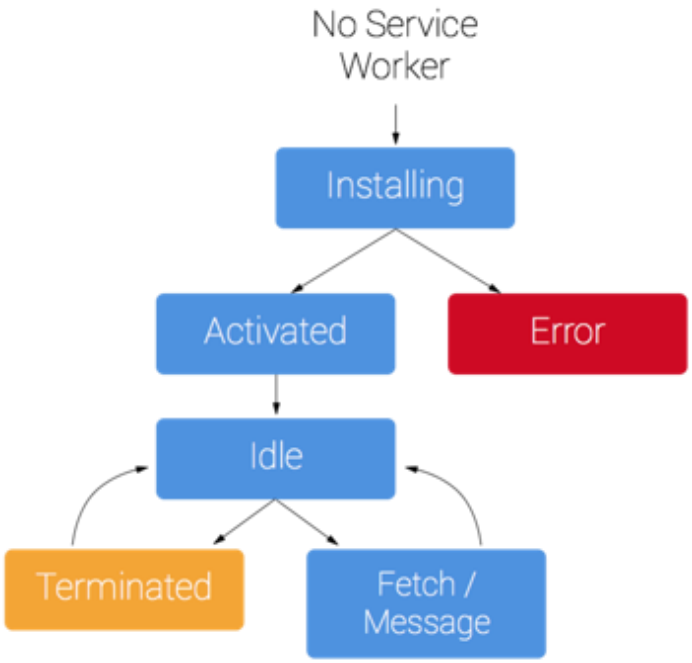
במטמון, כמו קבצי HTML ו-CSS סטטיים, ובכך כאשר נאבד קישוריות אינטרנט, הדפדפן יטען לנו את הרכיבים הסטטיים הללו, כך שנראה גרסה מסוימת של האתר, למרות שאיבדנו יכולת להתחבר לאתר המקורי.



ל-Service Worker יש תוחלת חיים ארוכה הרבה יותר מל-Web Worker. מרגע שהותקן Service Worker לראשונה, הוא יפתח את עצמו ללא צורך בהרשמה מחדש בכל עמוד תחת התחום המוגדר לו (Scope). ה-Service Worker ישוב לפעול בעת פתיחה מחודשת של הדפדפן ואף לאחר הדלקה מחודשת של המחשב.

מחזור החיים של Service Worker

ל-Service Worker יש מחזור חיים ייחודי ומורכב:



ראשית, יש לרשום (register) את ה-Service Worker באמצעות הפונקציה הבאה:

```
// main.js

navigator.serviceWorker.register('/sw.js')
  .then((reg) => {
    console.log(`Registered! Scope: ${reg.scope}`)
  });
```

הפונקציה מקבלת כפרמטר ראשון את מיקום הקובץ שמכיל את הקוד של ה-Service Worker. מיקום זה חייב להיות מיקום פיזי על אמצעי אחסון ולא ניתן להעביר Blob. הפונקציה מחזירה Promise וכאשר היא Resolved, אנחנו מקבלים מידע אודות ההרשמה.

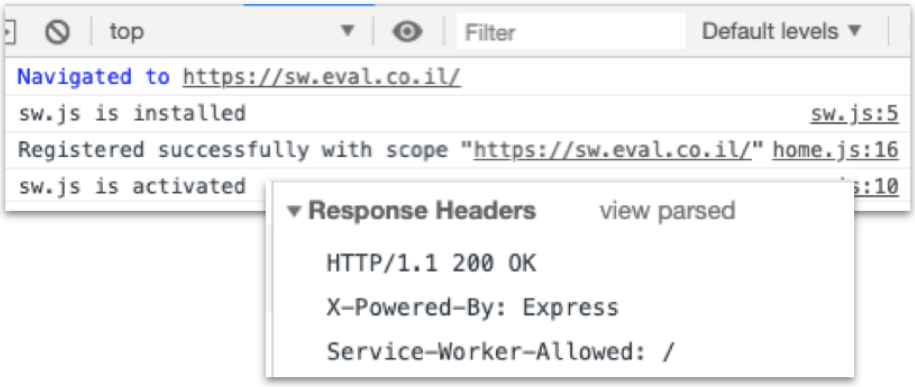
תחום הגדרה (Scope)

לתחום ההגדרה חשיבות מהותית בעולם ה-Service Workers. תחום ההגדרה מגדיר תחת אילו נתיבים (Paths) ה-Service Worker שלנו יפעל. תחום ההגדרה משמש הלכה למעשה כתנאי הפעלה:

```
// main.js

navigator.serviceWorker.register('/sw.js', {
    scope: '/profile/'
})
```

בדוגמא שלעיל קבענו את תחום ההגדרה ל- "/profile/", כלומר, כל דף בהיררכיה מתחת לנתיב זה יטען ויריץ את ה-Service Worker. כל מיקום גבוה ממיקום זה בהיררכיה (למשל "/") ידחה את ההרשמה של ה-Service Worker ולא יאפשר את הרצתו. כברירת מחדל, תחום ההגדרה נקבע למיקום שבו מאוחסן קוד ההרצה של ה-Service Worker. ניתן לצמצם את תחום ההגדרה בקלות יחסית, כפי שהראנו בדוגמא למעלה, אך גם ניתן להרחיבו בעזרת Response Header מיוחד בשם Service-Worker-Allowed:



ישנם מספר היבטים שיש לקחת בחשבון בעת קביעת תחום ההגדרה. למשל, רק Service Worker אחד ויחיד יכול לחיות תחת אותו דף. אם נרשמו שני Service Workers תחת אותו תחום הגדרה, האחרון שנרשם מקבל את זכות ריצה.

באופן דומה, כאשר שני Service Workers מופעלים בתחומי הגדרה חופפים, תחום ההגדרה המדויק ביותר יהיה זה שייתקבל.

אפשרויות נוספות

בתוך Service Worker ניתן לטעון קוד נוסף באמצעות שימוש בפונקציה "importScripts". עם שיטה זו תוקפים יכולים לטעון את ה-Service Worker הזדוני שלהם ועדיין לשמר את הפונקציונליות האמיתית של האתר (בהנחה שלאתר היה Service Worker מלכתחילה).

שלב ההתקנה (Installation)

מרגע שנרשם Service Worker בהצלחה, שלב ההתקנה מופעל באמצעות הפונקציה הבאה:

```
1 // sw.js
2
3 self.addEventListener('install', (event) => {
4     // handle installation
5 });
```

מאורע (Event) זה יתרחש בכל פעם שהדפדפן יבחין בשינוי קוד ההרצה של ה-Service Worker, למשל בעת שדרוג היכולות שלו.

שלב ההפעלה (Activation)

משהסתיים שלב ההתקנה בהצלחה, שלב ההפעלה יצא לפועל:

```
1 // sw.js
2
3 self.addEventListener('activate', (event) => {
4     // handle activation, like cleanup of old SW
5 });
```

במאורע זה ניתן לנקות קבצים קודמים המצויים ב-Cache באמצעות ה-Service Worker. נקודה חשובה להדגיש בהקשר למאורע זה - Service Worker לא יוכל לפעול בדפים שהיו פתוחים בדפדפן במהלך רישומו. על מנת לעקוף מגבלה זו, ניתן לקרוא לפונקציה "self.clients.claim" אשר דורשת זכויות הפעלה עבור ה-Service Worker על אותם דפים.

Fetch Event

לאחר רישום והפעלת ה-Service Workers, ניתן לגשת ולהגדיר כיצד יפעל ה-Service Worker כאשר מתרחשים אירועים שונים בדפדפן שבאמצעותם ניתן להשפיע על התנהגות האפליקציה כגון ה-Fetch Event.

מאורע זה מתרחש בכל פעם שהדפדפן נדרש לבקש משאב מרוחק כגון תמונות, CSS, JavaScript, וכדומה. כאמור, ל-Service Worker הרשאות יוצאות דופן ביחס ל-APIs אחרים, והן מאפשרות לו שליטה על הבקשות שנשלחות על ידי הדפדפן, ואף חשוב מכך, בתשובות המתקבלות במענה לבקשות אל (ספוילר להמשך) ו:

```
// sw.js

self.addEventListener('fetch', (event) => {
    console.log(`URL: ${event.request.url}`);

    event.respondWith(
        fetch(event.request);
    );
});
```

תרחישי תקיפה

כפי שתיארנו, ה-Service Workers API משמש לצורך שיפור חווית משתמש (למשל באמצעות Caching offline experience). במחקר שערכנו חשפנו מספר שיטות אותן יכולים תוקפים לנצל על מנת למנף השפעת חולשות טריוויאליות במהלך ההתקפה. חילקנו את תרחישי התקיפה לשלוש קטגוריות:

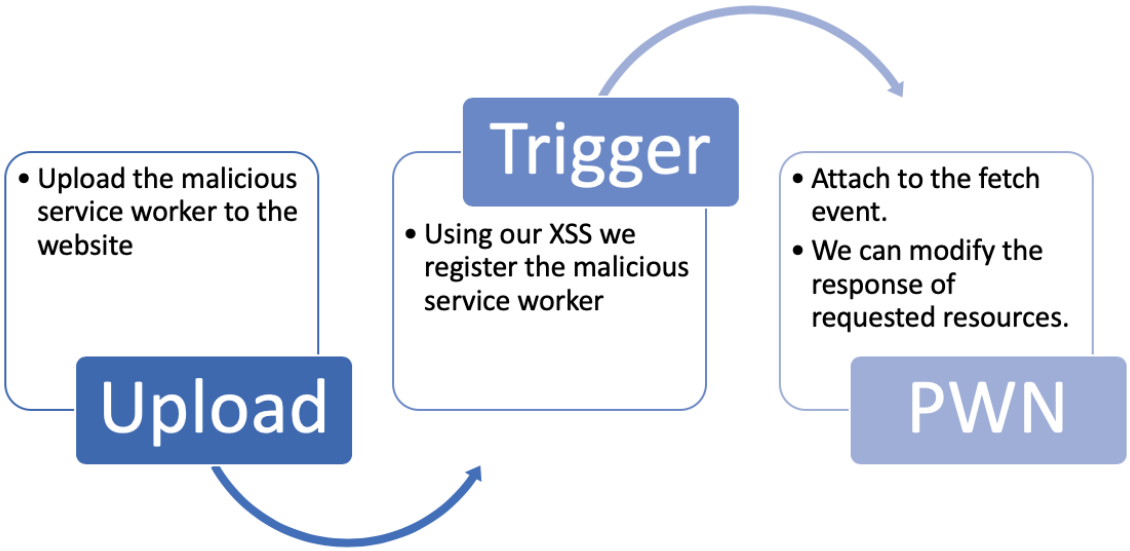
- שינוי תשובה (Response Modification)
- זליגת מידע מ-Sandboxed domains
- מינוף Self XSS

שינוי התשובה (Response Modification)

כל התרחישים תחת קטגוריה זו מסתמכים על יכולותיו של Service Worker להאזין ל-Fetch Event ולשנות את גוף התשובה שמגיעה מהשרת לדפדפן. לצורך הדוגמאות הבאות, יש להניח מספר הנחות יסוד בנוגע לאתר הנתקף:

- יכולת העלאת קבצים - לתוקף יש אפשרות להעלות קבצי JavaScript והוא מסוגל לגשת אליהם.
- קיימת חולשת Cross Site Scripting באתר המאפשרת לתוקף הזרקת קוד JavaScript זדוני משלו והרצתו בדפדפן של משתמשי האתר.

ניתן להמחיש את אופן התקיפה באמצעות התרשים הבא:



תרחישי הניצול האפשריים בקטגוריה הזו הינם:

שימור חולשת **XSS** - ברגע שהתוקף שולט ב-Fetch Event הוא מסוגל להזריק את קוד ה-XSS שלו מחדש בכל בקשה לעמוד HTML, גם במידה והחולשה המקורית בצד השרת טופלה. לסרטון המציג את המתקפה:

https://youtu.be/jE1nCdMHG8o

מניעת שירות מקומית **(local denial of service)** - ברגע שהדפדפן מבקש משאב מרוחק, התוקף יכול לשנות את ה-HTTP Response Code של התשובה ל-404 ובכך הלכה למעשה, למנוע את הצגתה למשתמש וטעינתה. לסרטון המציג את המתקפה:

https://youtu.be/H6Q81clc0gg

מתקפות דיוג (**phishing**) ושינוי האתר **(defacement)** - התוקף יכול לערוך את גוף התשובה ולהחליפו בתוכן אחר. כך למשל, יוכל לשנות את הכתובת של טופס התחברות, ובכך להזליג מידע רגיש החוצה. בשונה ממתקפות דיוג רגילות תרחיש זה חמור במיוחד כיוון שלתוקף אין צורך ביצירת אתר מתחזה ובחוויית המשתמש יראה התוכן החלופי כאילו הוגש מהאתר המקורי. לסרטון המציג את המתקפה:

https://youtu.be/XmCAvUptDGc

זליגת מידע מ-Sandboxed Domains

כדי להבין את תרחישי התקיפה בקטגוריה זו, חשוב לדון בטיבם של sandboxed domains והאופן בו נעשה בהם שימוש ברחבי האינטרנט. אתרים אלה נועדו לארח סוגים שונים של קבצים, כגון תמונות, קבצי אודיו, סקריפטים וכדומה. קבצים אלו מאוחסנים ב-domain ייעודי, נפרד מהאתר המרכזי על מנת להגביל את השפעתן של חולשות או תוכן זדוני על ה-domain המרכזי (תוך ניצול הגנת ה- same origin

policy שנאכפת על ידי דפדפנים). בנוסף, לא מעט sandbox domains מיועדים לאירוח קבצים ותוכן המועלה על ידי משתמשים נחשב כלא בטוח.

על אף היותם של sandbox domain ציבוריים, ארגונים עושים בהם שימוש לאחסון קבצי משתמשים, והדרך להבטחת פרטיותם הינה ערבול שמות הקבצים ויצירת נתיבי גישה אקראיים אשר אינם ניתנים לניחוש על משתמשים אחרים שלא העלו את התוכן.

עיקר מטרת הפרדת הדומיינים הינו העדר שיתוף עוגיות (cookies) בין הדומיין הראשי של האתר ל-sandbox domain, ובכך הגבלה של השפעת XSS או חולשות אחרות ב-sandbox domain על הדומיין הראשי.

בשל הפרדה זו, אתרים רבים נוטים להתעלם מחולשות XSS שמדווחים להם.

דוגמא ל-sandbox domains נפוצים:

- googleusercontent.com (Google)
- fbcdn.com (Facebook)

כעת נניח מספר הנחות העומדות בבסיס תרחיש התקיפה.

לצורך המחשה, נניח אתר שמאחסן מידע רגיש עבור משתמשים כגון תמונות. נקרא לו:
example-photos.com

קבצים שמועלים לאתר דרך הממשק, מוצאים את דרכם ל-sandbox domain בו עושה שימוש האתר. כמו כן, נניח כי האתר אינו מגביל העלאת סוגי קבצים נוספים, כגון קבצי JavaScript או HTML. הנחה נוספת הינה כי קיים XSS בדומיין ה-sandbox המאפשרת הרצה של Javascript.

להלן תרחיש התקיפה:
ראשית, התוקף מעלה את ה-service worker שלו ל-sandboxed domain. במסגרת הנחות המחקר ניתן לעשות זאת, כיוון שהאתר מאפשר העלאת קבצים ל-sandbox domain. שנית, התוקף מטריג את ה-XSS על הקורבן, ובתמורה יתקין את ה-service worker של התוקף על ה-sandboxed domain.

כעת, שהקורבן יכנס לאתר האמיתי ויבקש לצפות בתמונה, הדפדפן ישלח בקשה להבאתה מה-sandboxed domain. באמצעות ה-service worker שהתקנו בשלב מוקדם יותר, נוכל ליירט את הבקשה ולהזליג החוצה את ה-URL המורכב והמעורבל של התמונה הרגישה. לסרטון המדגים את המתקפה:
https://youtu.be/bkFCPl5bOB8

מינוף חולשת Self-XSS

לצורך הדוגמא נניח את התרחיש הבא: יש אתר בעל פונקציית העלאת קבצים. בנוסף, ישנו גם Self-XSS שבשונה מ-XSS רגיל, הוא מוטרג אך ורק בדפים המשפיעים על המשתמש הנוכחי, ובגלל זה אינם ממצאים חמורים בד"כ. במקרה הזה, למשל בדף "/profile/". לצורך הדוגמא, נניח שיש לאתר גם login/logout CSRF.

ראשית, התוקף מעלה את ה-service worker הזדוני שלו לאתר באמצעות פונקציית העלאת הקבצים של האתר. אח"כ, הוא גורם לקורבן להיכנס לאתר שנמצא בשליטתו (דומיין אחר לגמרי, למשל attacker.com). הדף הזה מכיל iframe שגורם למשתמש לבצע ניתוק באמצעות logout CSRF ו-iframe נוסף שרץ אחריו שמחבר את הקורבן לחשבון של התוקף באתר הפגיע. עד לנקודה זו, רמת החומרה של הפגיעות לא השתנתה.

אך כאן זה מתחיל להיות מעניין: ברגע שהתוקף חיבר את קורבן לחשבון שלו והפנה אותו לדף הפרופיל הלגיטימי לכאורה ("/profile/") התוקף בעצם מטריג את ה-self-XSS כדי להתקין service worker על הדפדפן של הקורבן. בשלב הזה, אותו XSS גורם לניתוק של הקורבן מהחשבון של התוקף.

לאחר מכן, כאשר הקורבן יכנס לחשבון שלו, הוא יגלה הפתעה לא נעימה. כעת, יש service worker זדוני שמותקן על הדפדפן שלו, ככה שלמעשה התוקף הסלים חולשת Self XSS פשוטה יחסית ו-login/logout CSRF לחולשה רחבה הרבה יותר - Persistent XSS ללא יכולת תיקון מצד האתר (אלא אם כן, הוא דורס את ה-service worker). לסרטון המדגים את המתקפה:

https://youtu.be/4cQcVN3NBFw

תמיד יש בעיות

למרות שהתרחישים למעלה נראים מבטיחים מאוד, אסור לשכוח את ההגבלות הבאות:
תחום הגדרה (Scope) - אם ה-service worker שהתוקף מעלה לאתר נמצא עמוק מדי בהיררכית הקבצים, סיכוי גבוה שלא תהיו לו שום משמעות בגלל ההגדרה של תחום ההגדרה. ה-service worker מופעל רק על קבוצת דפים מוגבלת.

סוגי קבצים (MIME Types) - על מנת שהדפדפן ירשום בהצלחה את ה-service worker של התוקף, הוא חייב להגיע עם MIME Type מתאים של JavaScript, למשל "application/javascript" (לכל דפדפן יש רשימה של סוגי MIME Types שמתאימים לקבצי JS).

מניעה

כעת, שהצגנו כל מה שיכול להשתבש, הגיע הזמן לדבר על איך אפשר למנוע תקיפות שכאלה. במחקר שלנו בחרנו להתמקד בשלושה דרכים עיקריות למנוע חלק מהתקפות אלה.

אכיפת תחום הגדרה

פתרון לגיטימי ראשון מדבר על להגדיר את תחום ההגדרה של ה-service worker בצורה "הרזה" והמיטבית ביותר עבור האפליקציה. בצורה הזו, גם אם תוקף יעלה service worker לאתר, אנחנו נצמצם את משטח הפגיעה רק למספר דפים מינימליים (אם בכלל). עבור המקרה של sandboxed domains, ההמלצה היא לבחור היררכיית דפים ייחודית לכל קובץ, ככה שאם תוקף יעלה service worker זדוני, הוא יוכל לשלוט רק על עצמו.

באופן כללי ההמלצה היא להימנע משימוש ב-"/" בתור תחום ההגדרה, דבר נפוץ שראינו בשטח (על ידי שליחת Response Header בשם service-worker-allowed).

מצב קטסטרופלי זה יוצר מצב שה-service worker יוכל לשלוט על כל דף באתר.

ניטור משתמשים אמיתיים (RUM)

ניתור משתמשים אקטיבי (Real User Monitoring) היא דרך פסיבית למדוד את חווית המשתמש של משתמשים באתר שלנו. בדרך זו אנו מודדים את האינטראקציה של משתמשים עם האתר מהרבה סוגי דפדפנים, ממקומות שונים בעולם, ובכך מעלים את השונות בבדיקת חווית המשתמש.

בצורה זו, נוכל לנטר עבור מצבים בהם חווית המשתמש ירודה עבור משתמשים מסוימים, אחת הסיבות לכך יכולה להיות service worker אשר גורם למניעת שירות מקומית.

ניטור Browser Events

על ידי ניטור מאורעות (Events) אשר מתרחשים בדפדפן, נוכל להחליף את הלוגיקה פונקציות מסוימות. במקרה שלנו, נוכל לבצע "מעקף" על מנת לבדוק מי קרא לפונקציה שרושמת service workers לדפדפן.

דוגמא לקוד שכזה:

```
// main.js

let oldReg = navigator.serviceWorker.register;

Object.defineProperty(navigator.serviceWorker, 'register', {
   get: function() {
      console.log('will be invoked before every register!');
      return oldReg;
   }
});
```

בצורה זו, נוכל לפעול ברגע שמישהו מנסה לרשום service worker שאינו לגיטימי, למשל לבטל את הקריאה לפונקציה במידה ונעשתה עבור קבצים שאינם מורשים.

לסיכום

ה-Service Worker API משרת נאמנה את עולם ה-web באתרים רבים על ידי שיפור חווית המשתמש והעשרתה ביכולות שבעבר יכולנו רק לחלום עליהם. יחד עם זאת, חוסר מודעות של מפתחים לסיכונים הכרוכים בשימוש Service Workers יכול להוות סיכון אמיתי למשתמשי האתר. אנו מקווים שבעזרת המאמר הזה מפתחי Web יוכלו לקבל זווית נוספת ושונה על API זה, ולהבין שעם כוח גדול באה אחריות גדולה ☺

מי אנחנו?

- דניאל אבלס, ראש צוות מחקר ב-Akamai, מתעסק בעיקר במחקר בעולמות ה-Web אבל אוהב מאוד לחקור גם בעולמות נוספים :) לשאלות, הארות, הערות או אם סתם בא לכם לדבר, אנא פנו אליי:
  - https://twitter.com/Daniel_Abeles
  - https://www.linkedin.com/in/daniel-abeles
  - daniel@abeles.me

- שי שביט, חוקר אבטחת מידע ב-Akamai, מתמחה בעולם ה-Web Security, ניתן למצוא אותי ב:
  - https://twitter.com/Chapakpuk
  - https://www.linkedin.com/in/shay-shavit-236b54a4/