אבטחת מערכות ויישומים ברשת - שיעור 13#

נושא היום: XSS Attacks and Mitigation:

בהרצאות שעברו אמרנו שאחת מה-coding vulnerabilities המרכזיים הם הנחת קלטים חוקיים ללא בדיקה, וכך מתאפשר לתוקף לתת לאפליקציה קלט לא תקין שיכול לגרום לאפקטים שונים. בהרצאה הקודמת, כאשר דיברנו על משפחת ה-injections שמנו לב שהתוקף משתמש באפליקציה כדי לתקוף מערכות שהאפליקציה משתמשת בהם כמו DB, מערכת ההפעלה וכדומה.

כיום נדבר על שתי משפחות התקפות:

- Cross-site scripting: ניצול חוסר input validation לתקיפת הדפדפן של המשתמש, העברת קוד עוין שירוץ עליו. אם ב-injection ההתקפה היתה ל-server side, כאן היא לתקיפת ה-browser של המותקף (הבדל באילו מערכות תוקפים).
- עוד משהו.

Cross-site scripting:

נושא חם, דרגה A4 ב-2004 בדוח OWASP, 2007 – A1 (!), 2010 – A2. ה-injection attacks ב-server/browser side הם פגיעות ראשונה במעלה, ו-injections יחד עם cross-site scripting מהווים את המסה הגדולה של התקפות גלויות. אלו התקפות יחסית קלות לאיתור.

Origins of JavaScript:

נבין את מקורות ה-vulnerabilities הללו. במקור Netscape פיתחו את LiveScript כדי לאפשר הצגת דפי HTML דינאמיים, כדי לאפשר יכולות אינטראקציה מינימלית עם המשתמש. לאחר שת"פ עם Sun זה עבר ל-JavaScript. נוצרה שפה שמאפשרת לעשות מניפולציה ל-HTML המוצג ע"י הדפדפן, ולא ראו בזה שפת תוכנה אלא יותר extension ל-HTML, לצורכי שילוב סקריפטים קטנים.

באותה תקופה הניחו שהרצת אפליקציות על דפדפנים תהיה ע"י Java applets – הרצת קוד ב-client side. במקרה זה עולה שאלת כמה בוטחים בקוד הזה. אזי החליטו שאלו יהיו חתומים בחתימה דיגיטלית, ואז הדפדפן יבדוק את החתימה הדיגיטלית, יבקש אישור מהמשתמש לאשר את ה-certificate, ורק אז הקוד יורשה לרוץ. החתימה נותנת אישור על מי עומד מאחורי הקוד ואישור שהקוד לא שונה.

אבל, JS לא תוכננה כשפת תוכנה ולכן לא חשבו שיש צורך במנגנון וידוא שלמות, integrity ו-authenticity של ה-JS. כמו כן כניסתה לתוך ה-HTML היא לא בהכרח באופן מרוכז, אלא פיזור בכל הקוד, ולזה אין מנגנון לחתימות דיגיטליות (וגם לא פרקטי לבצע זאת). בפרט בשל שיש אפשרות ליצירת קוד JS דינאמי.

Uses of JS:

אם כן מצוי קוד משולב בתוך ה-HTML, המאפשר לקבל events של המשתמש ולהפעיל קוד, למשל submit ל-form. ה-JS מאפשר תמיכה בהרבה event handlers. בהתאם לקלט של המשתמש ניתן לשנות את דף ה-HTML, החל משינוי תצוגת הדף וכלה בהצגת תוצאות (כמו מחשבון JS).

בתפיסה המקורית ה-JS היה חלק מה-HTML, ולכן לאחר ירידתו לדפדפן מהשרת, לא אמורה להיות אינטראקציה בין ריצת ה-JS על ה-JS engine בדפדפן לשרת. גם הנחה זו נשברה עם הופעת ה-AJAX – asynchronous JS and XML, המאפשרת תקשורת עם השרת. כך התפתחה JS להיות שפה בעלת יכולות רבות, בין היתר יצירת HTTP requests תחת מגבלות מסויימות (שיתן לעקוף), לטעון דפים וכו' – שפת תוכנה לכל דבר.

הדפדפן הפך להיות מיני-מערכת הפעלה – הדפדפן הוא מערכת / פלטפורמה להרצת תוכניות ב-JS. כיום כל דפדפן מכיל JS engine היודע להריץ JS. לתוכניות אלו יש יכולות רבות עם הגבלות מועטות (בשל תרומתן ל-usability).

Cross Site Scripting (XSS):

כאשר הדפדפן מקבל דף HTML, כיוון שמקבל מאפליקציה שסומך עליה, הוא סומך גם על ה-JS ה-embedded ב-HTML. אם תוקף מצליח להכניס קוד JS עויין לקוד, הוא ירוץ בתוך הדפדפן – כלומר זו מנגנון שמאפשר לשתול קוד עוין שירוץ בדפדפן המשתמש. ה-vulnerability הוא ב-web application, אך מה שתוקפים הוא ה-browser של המשתמש (ה-end user). בעולם ה-JS ישנם שני מנגנונים חשובים:

- ביצוע input validation.
- ביצוע output encoding.

אלו שני מנגנוני אבטחת מידע למפתח. חשוב מאוד להשתמש בשניהם כדי להתגונן מפני XSS.

XSS – the trust problem:

הפגיעות טמונה בכך שהדפדפן סומך על דף ה-HTML שהגיע מה-web application, תוך שהוא מניח שכל קוד ה-JS ה-embedded בו הוא חוקי, ולא יכול להבחין בין קוד חוקי לקוד שהוזרק לתוך ה-HTML.

דוגמא: תוקף כותב בפורום קוד JS עוין. ההודעה שלו נכנסת ל-DB של הפורום. מגיע משתמש חוקי אחר, נכנס להודעה, למעשה מבקש את הדף המכיל את ההודעה עם קוד ה-JS העויין, ואז הוא רואה את ההודעה וגורם גם להרצת הקוד בדפדפן שלו.

נזכר כי XSS היא אחת הטכניקות הנפוצות לגניבת session – כי ניתן להשיג כך גישה ל-cookie של ה-web application וכך לשלוח אותו החוצה אל התוקף. בשביל להגביל ש-JS לא יוכל לשלוח אינפורמציה לכל מקום, כהגבלה, יצרו את ההגבלה הבאה: ה-JS יוכל לשלוח הודעות רק ל-web server של אותו web application, כלומר קוד JS ייצר HTTP request אך ורק ל-web server שקשור אליו. אבל, יש מקרים יוצאים מן הכלל, למשל הבאת Images משרתים אחרים – ולכן ניתן ליצור אובייקט HTMLי הפונה לשרת חיצוני.

XSS attack capabilities:

מלבד גניבת session cookies (כבסיס ל-session hijacking), אז כיוון שה-JS יכול לבצע מניפולציה על כל ה-DOM והאלמנטים בו, יש דרך לשלוט לחלוטין על תצוגת המשתמש.

- ניתן לקבל את קלט המשתמש. למשל, אם ניתן להיכנס לתוך ה-submit – event handler שהופעל לשליחת ה-credentials - אזי תוקף יכול לגנוב אפילו אותם. לא פשוט להכניס קטע קוד ל-event handler של on submit, אך אפשרי.
- כמו כן ניתן לעשות redirection למשתמש המנסה להיכנס ללינקים, ולשלב התקפת phishing.
- ביצוע defacement בצד הדפדפן – בשרת לא השתנה דבר, אך בדפדפן כן. ה-defacement, מעבר להיותו כלי למאבקים פוליטיים באינטרנט, הוא לשימוש ב-phising.

עד כאן מדברים על בעיות המשתמש מול האפליקציה, אבל ישנן עבודות שהראו שקוד JS יכול לדגום את הרשת בה נמצא המשתמש, כלומר ביצוע scanning ותקיפת מחשבים ברשת שבה נמצא הדפדפן של המשתמש.

XSS attack examples:

-

XSS Categories:

ישנן 3 קטגוריות:

- Reflected: קוד ה-JS העוין לא נשמר ב-web server. נשלחת request ל-web server, ה-web application משלב את ה-JS העויין בדף ה-HTML. כאן התוקף שולח את ה-request אל המותקף, למשל דרך מייל, בתקווה שהמותקף יכנס ללינק, ואז הוא ישלח את הבקשה עם הקוד העויין ל- web application, ומהשרת תחזור ה-response עם "reflection" של הקוד העוין מהבקשה ב-response.
- Stored: קוד ה-JS העויין נשמר ב-web server, למשל הדוגמא של הפורום מקודם.
- DOM: בניגוד לשניים הקודמים, שם הפגיעות היא ב-server side code (חוסר ב-input validation), כאן הפגיעות היא בקוד הרץ ב-browser.

Reflected Cross Site Scripting:

המשתמש מקבל קוד עוין ועושה לו reflection לתוך ה-HTML response שמחזיר ל-browser ששלח אליו את ה-HTTP request. הבעיתיות היא שנעשה שילוב ללא ולידציה או ולידציה מספיק טובה. אם מנסים לבצע negative security logic למניעת XSS, צריך לכסות אזורים רחבים מאוד שכן הגמישות רבה. הדרך הקלאסית להפיץ reflected XSS היא לשלוח אימייל עם לינק. גם אתר סטטי יכול להיות חשוף ל-XSS.

דוגמא:

- תוקף יצר לינק לאפליקציה בנקאית עם שילוב קוד עויין בלינק ושלח זאת באימייל למותקף.
- המותקף קיבל אימייל, מקליק על הלינק, והמשתמש שולח את הסקריפט טמון בבקשה אל אתר הבנק.
- הוא לא מזהה זאת ולכן שולח response עם הסקריפט והוא יוצא לפועל ע"י הדפדפן של המותקף.
- הסקריפט הרץ על דפדפן המותקף שולח את ה-cookie וה-session info אל התוקף.

Stores cross site scripting:

נשלח קלט עויין לאפליקציה הנכתב ל-DB (האפליקציה נכשלה בזיהוי הקלט כעויין). מגיע משתמש תמים שלא קשור, ומקבל מהאפליקציה דף עם קוד JS עויין הרץ אצלו בדפדפן.

DOM based XSS:

למותקף מגיע לינק שיש בו קוד עויין. הקוד העויין לא נשלח אל השרת אלא נשאר בדפדפן, אך כאשר חוזר דף ה-HTML מהשרת אל הדפדפן, הוא מכיל קוד JS שמתייחס ל-request ושותל את התוכן העויין בתוך דף ה-HTML. ישנו אלאמנט הנקרא fragment שאמור להיות label בדף ה-HTML, לא נשלח ל-web server ונשמר בדפדפן. לרוב משתמש בו כ-label בתוך דף ה-HTML.

המבחן:

- אמריקאי, 40 שאלות מתוכם צריך לענות על 33.
- לא מבחן קשה למי שיודע את החומר.
- יועלו שאלות לדוגמא.
- יתכנו שאלות על דברים בתוך הפרוטוקולים שלמדנו.
- כל החומר מתוך ההרצאות בלבד.

התגוננויות:

(1) Protect against XSS – input validation:

מנגנון ראשון להתמודדות – כל שדה או קלט שה-JS בעמוד משלב לתוך הבקשה חייב לעבור input validation. מאוד מומלץ שהוא יהיה ב- **positive security logic** כיוון שמאוד קשה לבצע negative אפקטיבי כדי להמנע מ-XSS, כיוון שיש וריאנטים רבים שקשה לתפוס. בדוגמא של הפורומים יש בעיה – כי שם הקלט הוא free text, ולכן צריך שם negative. אך איפה שלא חייבים, נעדיף positive. התמודדויות:

- הגבלת האורך: אם מגבילים את האורך, מגבילים את הפקודות שאפשר לכתוב.
- צמצום ה-characters לסט תוים חוקי לשדה מסויים.

אם אותר קלט עויין:

- לא משלבים אותו בדף ה-HTML. אם ישולב ב-error page כמו ב-reflected או DOM המוחזר למשתמש, התוקף השיג את מבוקשו. ב-stored, אם מזהים קלט לא חוקי הנכתב ל-log file, אז יתכן שהכלי שדרכו האדמין עובר על ה-log הוא מבוסס דפדפן, אז זה ירוץ אצלו. לכן חשוב לבצע לו סניטציה ו-encoding – נדבר בהמשך.
- לא לנסות לתקן את הקלט כך שיהיה לא עויין (יש התקפה מבוססת content replacement).

בנוסף:

- לא לשתמש ב-blacklist validation שהוא סוג של negative – כלומר זיהוי תבניות עויינות – כי ניתן להכניס כל מיני דברים באמצע, למשל רווחים וכו' ויותר מכך ישנן דוגמאות שבהן לא צריך script tag כדי להכניס JS לדף ה-HTML.
- יש להשתמש בספרייה טובה לחיפוש וריאנטים שונים לגילוי XSS.

Content replacement as attack vector:

ניתן לשלוח scr<scriptipt … - שלאחר ניקויו יקבלו … script>: כלומר הקלט הראשוני אינו התקפה. נעשה ניסיון לנקות את התוכן העויין, ותוקף שיודע את צורת ההתקפה יכול לתכנן קלט שנראה לא עויין, ויתוקן לקוד עויין – כאשר לאחר התיקון לא מתבצעת בדיקה חוזרת של הקלט.

(2) Protect agains XSS – string output Encoding:

מנגנון שני הוא קידוד תווים לדפדפן כך שהדפדפן יבין אותו כבעל משמעות תצוגתית ולא executionable. משמעות כל צורת קידוד תלויה במיקום שלה בתוך דף ה-HTML, ולכן חשוב להשתמש ב-library היודע לבצע output encoding בהתאם למקום בו הולכים לשלב את הקלט שמשלבים בדף.

דוגמאות: < הוא < - משמעות התו היא לתצוגה בלבד. כדי שהדפדפן לא יתבלבל, חשוב שהמפתח יגדיר בצורה מפורשת מה ה-encoding של הדף, כדי שהתוקף לא יוכל לבצע מניפולציה על ה-encoding של הדף.

דוגמא לספרייה של מיקרוסופט:

יש ב-ASP.NET ספרייה המאפשרת למפתח לבצע output encoding בצורה טוב ללא הבנת כל הדקויות. הספרייה מכילה פונקציות המקודדות את הקלט בהתאם למקום בו משלבים אותו – למשל האם משולב ב-HTML, ב-JS, ב-XML, ב-XML element וכו'.

התקפה נוספת: Cross Site Request Fogery (CSRF / XSRF):

התקפה זו מתרחשת בדפדפן, ומטרתה לייצר פקודות מזוייפות בשם המשתמש שישלחו מהדפדפן של המשתמש אל ה-web application. מבחינת האפליקציה בקשה זו תהיה חוקית מהמשתמש – הגיע מהדפדפן של המשתמש וה-credentials שלו, אך בפועל לא נוצר ע"י המשתמש (נוצר אולי ב-background ללא ידיעת המשתמש).

ההבדל בין XSS ל-CSRF:

אלו אמנם שתי התקפות הרצות בדפדפן, אך כל אחת מהן מנצל אמון שגוי בדברים שונים:

- XSS: אמון שגוי בדפדפן בדף HTML שהתקבל מה-web application.
- CSRF: אמון שגוי של ה-web application ב-request שהתקבל מהמשתמש.

הפקודה שמגיעה מהדפדפן לאפליקציה היא נכונה סינטקטית ועם הקרדנצ'לס של המשתמש. זו לא בעיה של input validation כי זו פקודה חוקית לחלוטין. אין שימוש בקוד עויין (יש וריאנט שמשלב XSS). אם אפליקציה חשופה ל-XSS היא חשופה גם ל-CSRF ואף אין לה יכולת להתגונן מפניה. כמו כן, חסינות בפני XSS לא מקנה חסינות ל-CSRF.

כיצד אפליקציה מזהה את המשתמש ששלח את הבקשה? ע"י cookie המכיל את ה-session id, המצורף אוטומטית ע"י המשתמש.

כל אפליקציה שלא מבצעת authorization אפקטיבי על הפקודות חשופה ל-CSRF, אך גם אפליקציה שמסתמכת על credentials שנשלחים אליה אוטומטית ע"י הדפדפן. כעקרון אין דרך אולטימטיבית להגן בפני זה. בעצם רוצים לוודא שהמשתמש אכן נתן את זה, וניתן למשל להשתמש ב-captcha כחלק מהבקשה. זה מעצבן את המשתמש ולכן לא usable כל כך, לכן צריך לנתח אילו בקשות באמת יוצרות איום כלפי ה-.web app ועליה לעשות הגנה.