אבטחת מערכות ויישומים ברשת - שיעור 7#

נושא היום: Web Application Threat Analysis and Intro to Web Application Security:

פע ששעברה עצרנו בתפר בין התשתיות ובין האפליקציה. דיברנו על איך מקשיחים את התשתיות דרך הפרדת ה-web app. לכאלו הנגישים רק למשתמשים פנימיים (store \ market admin) ובין אלו הנגישים לכל העולם – קטלוג, ביצוע רכישה. ההפרדה לשני שרתים שכל אחד מהם מאזין על ip אחר תאפשר ליישם access-control לגבי הגישה לאותם web servers המריצים את ה-web app. באמצעות firewall – וכך מקבלים הגנה של access control ב-apache.

דיברנו על כך שנרצה להוריד את כמות ה-assets הנגישים לכל העולם, בין היתר תוקפים. חישוב סיכון משקלל את הסיכוי להתקפה מוצלחת ואת הנזק שייגרם אם תהיה התקפה מוצלחת. הנזק הפוטנציאלי שיכולה לגרום התקפה מוצלחת נמוך גורר סיכון נמוך יותר. נרצה להתקין על המערכת executables ולא source code כי אותם הרבה יותר קשה לנתח, וערכם אל מול תוקפים יורד. כמו כן אמרנו שה-asset המרכזי לתוקף הם פרטי כרטיס האשראי איתם נעשתה רכישה באתר, ולכן לא נשמור אותם או שנצפינם באמצעות public key כך שרק אם ירצו לנתח אותם יעבירו אותם למערכת offline ושם יפוענחו עם private key. כך גם אם יגנבו הם מוצפנים וחסרי משמעות לתוקף.

עד כאן התייחסנו לתשתיות ברמת מערכת הפעלה, web server, ההגנות שיכולות לתת חומות אש ו-access control. לא דיברנו על תקיפת מערכת תוך ניצול פגיעויות ברמה האפליקטיבית. בהרצאה הקודמת דיברנו על כך שפעמים רבות ה-web app. הוא ה"בטן הרכה" של המערכת – קל יותר להקשיח ולהגן עליה ברמת התשתיות, אך אם ה-web app. מכיל פגיעויות אלו יכולים לאפשר לתוקף לתקוף לא רק את ה-web app. אלא גם מרכיבי תשתית אחרים שהאפליקציה עובדת איתם. למשל, לאפליקציה יש גישה ל-DB, ל-File System במערכת ההפעלה ולכן תוקף יכול דרך האפליקציה לגשת לאותן מערכות. תוקף יכול לגרום לאפליקציה לבצע בעבורו בשמה גישות ל-DB או פעולות RWX במערכת ההפעלה.

עד כה התעסקנו בעיקר ב-infrastructure של המערכת – firewalls, SSL, שימוש בהצפנה וכו'. מעתה הפוקוס עובר לרמת האפליקציה. נראה שפגיעויות רבות באפליקציה נובעות מכך שהיא לא מבצעת ולידציה נכונה או בכלל על קלט שמקבלת מהמשתמש – ישנן input validation vulnerabilities באפליקציה, הניתנת לניצול ע"י תוקף. תוקף יכול לשלוח לאפליקציה תכנים עויינים שיכולים לגרום לאפליקציה לבצע בשמו שאילתות ל-DB היכולות להכיל למשל SQL injection עויין. קריאות למערכת ההפעלה יכולות גם כן להיות מנוצלות לטובת תוקף. פעמים רבות קלט המשתמש ישולב בדפי HTML שיוחזרו למשתמשים אחרים, ותכונה זו מנוצלת לצורך התקפות cross-site scripting – תקיפת משתמשים אחרים על אותה רשת דרך האפליקציה. גישות ל-File System יכולות לגרום ל-privilege escalation – ניצול הרשאות האפליקציה שבד"כ גבוהות יותר מהרשאות המשתמשים, או File Disclosure. בנוסף ל-input validation ישנם אספקטים נוספים, אך זה אחד העיקריים בהם.

דרכי התמודדות עם הבעיה:

ישנן שתי דרכים עיקריות להתמודדות עם הבעיה:

- פיתוח קוד מאובטח: זו התפיסה העדיפה, ועוד נעסוק בכיצד מפתחים web application מאובטח שאין בו vulnerabilities.
- הנחת קופסא שחורה: מניחים כי האפליקציה היא קופסה שחורה שאין לנו גישה לקוד שלה, דבר שנכון במידה והאפליקציה היא מוצר חיצוני. פיתוח קוד מאובטח כמובן לא ניתן במקרה זה, והאלטרנטיבה, הפחות טובה, היא לנטר את הקלטים אל האפליקציה, האינטראקציה של האפליקציה עם ה-DB וה-FS, הדפים שהאפליקציה שולחת חזרה אל המשתמשים. דוגמא: network firewalls – מסתכלים רק על רמת ה-IP,Port ולא על רמת ה-application data. ניתן להשתמש ב-web application firewall שהוא ייבחן את ה-data האפליקטיבי שנשלח אל האפליקציה, יחפש בתוכו תקינות, ואם יתקל בקלט לא תקין הוא יסונן – וכך האפליקציה תישמר מפני קלט לא תקין, גם אם האפליקציה לא מבצעת זאת בעצמה.

Web app. Firewalls (WAF): מוצרים שיושבים בד"כ בין ה-web client ל-web server, מבוססים על proxy (reverse proxy) התופסים את ה-HTTP request (כלפי ה-web server הוא web client), ובוחן אותם כנגד policy המגדיר מהו קלט תקין. כאן מתבצע ה-input validation, במקום בתוך האפליקציה. כפי שלמדנו, חוזק ה-app. Firewall יהיה כחוזק ה-policy שתוגדר עליו.

כיוון שלא ניתן במקרים אלו להגדיר positive security logic, צריך להגדיר negative security logic בה ינוטרו ויסוננו. הבעיה כאן היא שמאוד קל באופן יחסי לתוקף "להטמין" קלט עויין בתוך בקשות לגיטימיות. כאשר עובדים עם negative security logic, עובדים על בסיס תבניות. הבעיה בכך היא שיש tradeoff בין דיוק הבדיקות ובין תפיסת בקשות עויינות – ככל שהתבניות יותר כלליות, כך יותר התקפות נתפסות, אך בתוכן false-positive, וככל שהתבניות יותר מדוייקות, כך פחות התקפות נתפסות אך אלו שנתפסות בסיכוי גבוה יותר אכן עויינות.

מנהלי מערכת לרוב ישאפו למניעת false positive, ולכן יגדירו תבניות פרטניות ומדוייקות, ולכן האפקטיביות של כלי WAF תהיה נמוכה יחסית. בנוסף חולשתו נובעת מכך שצריך להכיר את הפרמטרים של הקלטים האפשריים לאפליקציה, כאשר ללא קוד מקור לא ניתן להכיר זאת. ישנם כלי למידה שע"י requests מבינים את הפרמטרים, ולפיה בונים באופן אוטומטי security policy. הבעיה בכך היא שכיום כלים אלו מוגבלים ביכולתם ותוצאותיהם מוגבלות.

<u>שורה תחתונה</u>: כדי להסיר איום באופן מירבי צריך להגן על הקוד של האפליקציה.

כמו שיש כלים לניטור הקלטים לאפליקציה, ישנם כלים לניטור השאילתות אל ה-DB מהשאילתא, שמטרתם לאתר שאילתות החורגות מפרופיל מוגדר לשאילתות חוקיות. המוצרים בתחום זה גם כן מוגבלים שכן מאוד לא טריוויאלי להגדיר מהי שאילתא חוקית, ולכן רוב מוצרים אלו עובדים בתפיסת monitoring – מנטרים, מתריעים אך <u>לא</u> חוסמים. מוצרים אלו חזקים יותר ממוצרי WAF מתוך הגישה שכלים אלו מהווים הגנה על המידע, שהוא המוקד החשוב במערכת. מוצרים אלו הם <u>Database Firewalls</u>.

ישנם מוצרים המנטרים את הקלט והפלט בין האפליקציות ל-File system הנקראים מוצרי Netweork Intrusion Detection / Prevension או Host ... - ה-...Host הוא ניסיון להגדיר תבניות לבקשות חוקיות ולא חוקיות מול מערכות הפעלה ומערכות קבצים כמו עבור packets ברשת, אך קשה מאוד להגדיר זאת במערכות אלו לעומת מידע העובר ברשת. לכן קשה לבנות כלים אפקטיביים ל-prevension, אך ישנם כלים ל-detection.

לסיכום, סוגי ה-firewalls שהכרנו עד כה ברמה האפליקטיבית:

- WAF – web app. Firewall
- DBF – database firewall
- Host Intrusion Detection / Prevension

<u>Screen App. Boundaries/Interfaces</u>:

לסיכום – ניתן לנטר כל אחת מהגישות בין האפליקציה למערכות סביבה.

Minimize File-System Premissions:

אלמנט נוסף שיכול להקטין את הסיכון ברמת האפליקציה הוא הרשאות האפליקציה – ככל שהרשאות האפליקציה לבסיס הנתונים או למערכת הקבצים ומערכת ההפעלה יהיה מצומצם יותר, כך הנזק הפוטנציאלי שיכולה לגרום נמוך יותר. למשל, באפליקציית הקטלוג, אין צורך ביותר מ- read access לבסיס הנתונים. אם לאפליקציה זו יש פגיעות המאפשרת גישה עויינת לטבלאות ה-DB, כיוון שלאפליקציה לא תהיה הרשאת כתיבה או הרשאת קריאה של טבלאות לא רלוונטיות, אזי תוקף לא יוכל לנצל זאת למשל באמצעות SQL injection.

באפליקציה של החנות הכותבת פרטי transaction לטבלה, אם נגדיר את הרשאותיה לכתיבה בטבלה מסויימת בלבד ונגדיר את הרשאותיה לכתיבה בלבד, אפילו לא קריאה או מודיפיקציה, אז השימוש בה יוגבל כך שנזקים אפשריים ממוזערים ככל שניתן. באופן דומה כלפי מערכת ההפעלה ניתן להגביל ולחסום כך למשל OS command injection, כי האפליקציה תוגדר עם הרשאות מינימליות למערכת ההפעלה.

פגיעויות רבות קשורות בכך שהאפליקציה עושה שימוש במנועים המבצעים סקריפטים. ככל שמזער שימוש ב-script engines ונעבור לשימוש ב-compiled code כך נקטין את הפגיעויות הללו, שהן יחסית בעלות נזק פוטנציאלי גבוה ויחסית קלות לניצול.

Examine Attack Paths:

בסופו של דבר ה-asset העיקרי הוא המידע ב-DB, אליו בסופו של דבר תהיה גישה מהאפליקציה. מספיק פגיעות באפליקציה כדי לעקוף את כל ההגנות לאורך מסלול ההגנה. המינימום הנדרש הוא להגדיר account נפרד בבסיס הנתונים ולכל משתמש כזה (עבור ה-web site, store, CMS admin, store admin) המייצג אפליקציה יהיו הרשאות לבצע את הפעילות שלו לפי עקרון ה-least privileges. אם לכל משתמש כזה יהיו הרשאות מתאימות מינימליסטיות, אז הנתונים יהיו מוגנים דרך התקפות דרך אותן אפליקציות.

Database Risk Mitigation:

במקום להגדיר משתמשים שונים על אותו DB, נבנה כמה DBs נפרדים, כך שההפרדה תהיה ממש ברמת בסיסי הנתונים. זה דומה ל-DMZ – האם עובדים על firewall אחד ומגדירים חוקים על כל אחד מזוגות הרגליים, מול הגדרת שני firewalls נפרדים.

Additonal Mitigation Activities:

אם לא עובדים במצב של prevention, ניתן לאגור את דיווחי ההתראה וה-monitoring במקום אחד שיכול לבצע הצלבה של כלי ה-detection באמצעות כלים מסויימים החל מהרמה הנמוכה ועד לרמה הגבוהה בה המערכת יכולה לתת התראות בזמן אמת למפעיל. אם אותם כלים ינפיקו התראות ממוקדות, הן יקבלו יחס יותר רציני, ולכן חוזק מערכות אלו נמדד בהתראות האמת שנותנות לעומת התראות השווא.

לכל ארגון גדול יש network operation center אליו מדווחים כל התקלות. כיום מקובל להצמיד אליהן secure operation center שמטרתם לאסוף את האינפורמציה ברמת ה-security לפחות לצורכי תיעוד ובמידת האפשר להוצאת התראות זמן אמת.

עניין נוסף הוא אבטחת ה-backups, שיכול להכיל גיבוי למידע רגיש. צריך להגן על הגיבויים באמצעות הצפנה.הגיבויים חשובים ל- disaster recovery procedures, אך יש להגן עליהם.

The Current Status of Web Application Security:

What is web app. Vulenrability:

פונקציונאליות הקיימת באפליקציה שאף אחד לא התכוון שתהיה שם, שאינה מפריעה לפעילות הנורמלית של האפליקציה. זהו לא באג ב-feature, אלא כזה שלא התכוונו שיהיה קיים. מטרת התוקף היא מציאת אלו ותקיפה דרך ניצולם.

פגיעות זו מהווה business risk שאחראי המערכת צריך להחליט האם להשבית את פעולת המערכת (כמעט ולא קורה), מתקן את אותה פגיעות מהר או אפילו מתעלם כי מוכן לקחת על עצמו את הסיכון. את ה-unintended functionality ה-QA לא יכול בכלל לבדוק, ולכן בדיקות security QA שונים באופי מ-QA רגיל לתוכנה, וזה לא טרוויאלי להגדיר testplan למציאת אלו.

App. Vulnerabilities Categories:

1) Coding errors: בעיות של המפתח בקוד. הבעיה העיקרית במקרים אלו המהווה בסיס להתקפות רבות היא חוסר ב-input validation. כאשר מוצאים בעיה כזו, יחסית קל לתקנה ולאכוף אותה.
2) Bugs: באגים בתוכנה שעלולים לגרום למשל לדריסת זיכרון, בתוך כך נכנסים features לא מדוקמנטים, שבסופו של דבר מתגלים ע"י תוקפים.
3) Design flaws: בעיות אלו קשות יותר לתיקון. כאשר בעיות כאלו מתגלות בשלבי ה-production / deployment, מאוד קשה ליישם תיקון. יישום מנגנוני אותנטיקציה בצורה לא נכונה הדורשת החלפתו – תהליך מסובך. כך גם authorization, access-control הם מנגנונים שמימושם בדיעבד מסובכים, לעומת תיקון coding errors אד-הוק. לעתים יכולות להיות בעיות business logic.

סה"כ שתי הקטגוריות העיקריות הן coding vulnerabilities ו-design flaws.

מדו"ח WhiteHat Q1 2009 בו נבדקו 1031 אתרים נמצאו 17888 פגיעויות. כיום ישנם שני כלים מרכזיים אוטומטיים לגילוי פגיעויות, אך רמת ה- false positive שלהם גבוהה. הנתונים בדו"ח נאספו מ-2006 עד 2009, ונעשו סריקות על האתרים פעם בשבוע. האתרים עליהם מבוססת הסטטיסטיקה הם אתרים רגישים לנושאי ה-security בעלי high-volume transactions, כאלו שמצפים מהם להיות מאובטחים ברמה גבוהה. תוצאות:

- 82% מהאתרים הכילו פגיעויות שהוגדרו כ-high, critical, urgent כאשר האחרון דורש הורדת המערכת.
- 63% הכילו בעת פרסום הדו"ח לפחות באג אחד critical / urgent.
- אחוז תירון הפגיעויות עומד על 60%, כלומר 7157 מתוך 17888 הם unresolved issues שלא תוקנו. כלומר, גם כאשר דווח, לא טרווילי לתקן אותו, שכן התהליך מורכב וארוך – התוצאה הסופית היא שזמן תיקון פגיעות הוא משבועות לחודשים.
- ממוצע הבאגים עם פגיעות גבוהה בסריקה ראשונה הוא 17. ממוצע אלו המתוקנים מתוכם עומד על 7.

המשיך לדבר על הדו"חות לאורך השנים. הנקודה החשובה היא שהרבה אתרים מכילים פגיעויות חמורות, שלאורך זמן תיקונם היה מינורי ואף לא קיים. נקודה לשים לב אליה – באתרי Retail המצב הכי טוב, שכן כל פגיעות מתפרשת ישירות לכסף (לעומת אתרים אחרים).

The Web Application Security Problem:

Multi layer information Security:

בכל רמה עליה דיברנו עד היום ישנם סטנדרטים ופתרונות בונים לאבטחה – User layer, Transport layer, access layer, network layer. ב-application layer אין כלים מפותחים וחזקים לפתרונות אבטחה. אמנם יש WAF ו-DBF או כלי סריקה, אך אלו סובלים מאחוז גבוה של false-positive ו-false negative, כיוון שקשה להתמודד עם כל הוריאנטים השונים בפיתוח אפליקציה. שתי משפחות כלים עיקריות:

- ה-web app. Vul. Scanner מדמה תוקף מבחוץ: מסתכל על האפליקציה כקופסא שחורה ומנסה לתקוף אותה כדי לגלות vulnerability. הבעיות הן הקושי לתכנן בזמן סביר את ההתקפות שרוצים לבדוק, ולא קל להבין ולנתח את תגובת האפליקציה.
- Static code analysis: כלי למפתחים שאמורים לסייע להם לגלות בקוד חריגות מ-best practices, למשל גילוי אי בדיקת קלט לפני שימוש בו. כלים אלו יקרים, לא תמיד יודעים כיצד להשתמש בהם כמו שצריך, ויכולתם מוגבלת.

Application security is a different world:

עולם אבטחת האפליקציה שונה בתכלית מאבטחת הרשת.

בעולם הרשת: אין מוצרים רבים כמו בעולם האפליקציה, ואותם מוצרים נמכרים במספרים גבוהים ובעלי סבירות גבוהה לגילוי פגיעויות. כמו כן קיימים מומחים בתחומם בעולם הרשת. ישנם סטנדרטים בעולם הרשת וניתן לעבוד עם חתימות ו-patch management.

בעולם האפליקציה: ישנה בעיית מודעות ל-Security. web app בקרב הארגונים המפתחים .web app. רוב המפתחים מבינים היטב פונקציונאליות, ביצועים ותחומי תוכנה, ופחות ב-security. חוסר המודעות בקרב מפתחים או מנהלי מוצר מהווה שורש לבעיה. נקודה נוספת היא שאפליקציות רבות מפותחות in-house ומוצאות בעותק יחיד או מעט עותקים, והסיכוי לגילוי פגיעויות נמוך מאוד. בגלל חוסר סטנדרטיזציה של פיתוח קוד, לא ניתן להשתמש בחתימות – לא ניתן לכתוב חתימה להתקפות, כלומר לאפיין אותה ומה חוקי ומה לא.

The Path of least resistance for Hackers:

רוב ההתקפות נעשות על האפליקציה. לתוקף לעומת מפתח אין מגבלות זמן, וזאת בנוסף לכך שמפתח צריך לספק 100% הגנה והתוקף צריך למצוא רק פגיעות אחת.

Security evolution toward security 3.0:

- Security 1.0: ...
- Security 2.0: כאשר נכנס עידן הרשת בו ה-client אינו רק טרמינל אלא גם מחשב שמריץ תוכנה, האיומים נכנסו לרמת ה-client (אנטיוירוס), עולם הרשת – firewall, הצפנות, VPN וכו'. בדור 1.0 אבטחת המידע היתה בתוך המערכת. ברמת האפליקציה ישנו מעבר ל-proactive security, כלומר לבנות את אבטחת המידע בצורה אינטגרטיבית אל תוך האפליקציה כדי למנוע את הפגיעויות. אם יתקנו את הפגיעות בדיעבד זה יקר ומסובך לפיתוח, וקשה להשתמש בפתרונות חיצוניים להגן על הקוד.
- Security 3.0: ...

Characteristics of Web App.:

מה הופך אותו לפגיע כל כך:

- החיבור לרשת האינטרנט מאפשר ברמה העקרונית לכל אחד בעולם לגשת אליו – החשיפה מבחינת הסיכונים גדולה.
- פעמים רבות התוקפים יושבים במקומות בהם גם החוק אינו הגנה במובן שאם תוקף ביצע תקיפה, היא תהיה לא חוקית.
- Client-server interaction בארכיטקטורה מורכבת יותר מאשר client-server קלאסי, והארכיטקטורה הזו הטרוגנית – מורכבת ממוצרים שונים, טכנולוגיות שונות ופרוטוקולים שונים, והמפתח צריך להכיר טכנולוגיות רבות ואת אופן השימוש המאובטח בהם. ברמת המערכת מפתח צריך להבין את הפגיעויות של אותן טכנולוגיות ואת טכניקות ומנגנוני האבטחה. ככל שהמערכת בעלת פונקציונאליות גדולה יותר, כך היא פחות בטוחה.
- אפליקציה מתבססת על פרוטוקול HTTP שהוא stateless, sessionless – טוב לתמיכה בבקשות מרובות בו"ז, אך הצורך לחבר requests למשתמשים יוצר בעיית management ומקשה על אותנטיקציה ו-authorization. חיבור לא נכון למשתמש יבטל את רלוונטיות האותנטיקציה, ואז לא ניתן ליישם גם access control. העבודה מעל HTTP יוצרת מורכבות רבה בניהול sessions המשליכה באופן חמור על authentication, authorization.

What makes web app. (more) vulnerable:

- מעבר לגישה שיש דרך האינטרנט לאפליקציה ברשת, ברשת ישנם כלי תקיפה רבים העומדים לרשות התוקפים. אם בעבר האקרים היו צריכים רקע טכנולוגי משמעותי כדי לפתח כלי תקיפה, כיום ישנם כלים קיימים שעם ידע טכנולוגי מינימלי ניתן לתקוף באמצעותם. כלומר הנגישות עלתה והידע הנדרש ירד.
- בעבר כדי להבין את הקוד האפליקטיבי בשביל לתכנן התקפה מוצלחת צריך לבצע reverse engineering על ה-executable המותקן על מחשב התוקף. פעולה זו היתה מורכבת מאוד שכן הייתה שליטה על התפוצה שלו, וגם מי שהשיג אותו, הבנת הקוד האפליקטיבי מתוכו מסובך מאוד בסדרי גודל של שבועות עד חודשים של עבודה. כיום בעולם ה-.web app ישנה נגישות לקוד, כיוון שבקבלת דף web ניתן לצפות ב-source שלו ולראות בקלות את כל הקוד האפליקטיבי, והבנת קוד HTML או JavaScript אינו מסובך (ב-client side). ממנו ניתן ללמוד על מבנה הפרמטרים וה-requests – ידע רב שיעזור לתכנן את ההתקפה.
- Client side validation: כאן נגמר הסיכום.