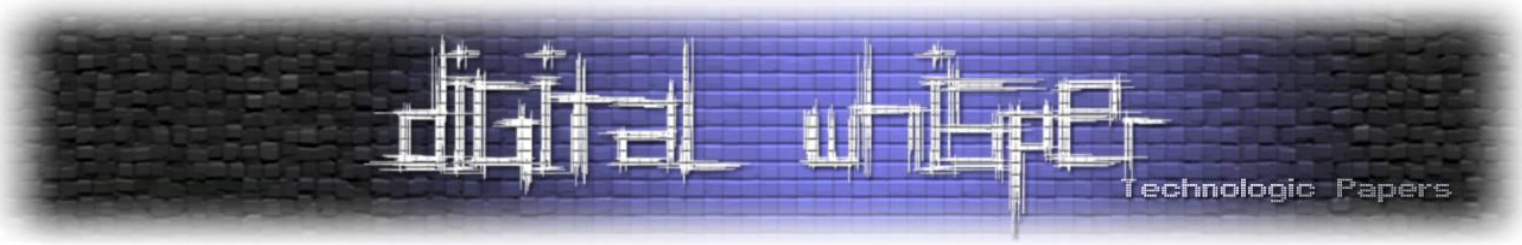


בעיות אבטחה נפוצות במימוש מנגנון Stateless

מאת ישראל חורז'בסקי [Sro], סמנכ"ל טכנולוגיות, AppSec Labs

הקדמה

ב-HTTP כל בקשה (Request) ברמה האפליקטיבית היא בקשה לחוד (גם כאשר מוגדר Connection: keep-alive, הקישור רק ברמת הסוקט ולא ברמה האפליקטיבית). מה שאומר שאם היוזר ביצע בקשת לוגין, קיבל תשובה (Response) ואז ביצע בקשה נוספת לאיזשהו דף, האפליקציה (או האתר) - על פניו - לא יודעת שזה אותו יוזר שרק לפני שניה הזדהה. כדי לעקוב אחרי ה-Flow של היוזר, יש לאפליקציות צורך ב-State ששומר מידע כמו: האם היוזר הזדהה, ואם כן מי הוא. מהן ההרשאות שלו וכד'.

באפליקציות Full state, כל המידע נשמר בצד-שרת, ב-DB או ב-RAM של השרת והמידע של כל יוזר ממופה מאחורי Session ID שנשלח לקליינט (ברב המקרים - דפדפן). הקליינט שולח את ה-Session ID לשרת עם כל בקשה (ב-GET/POST/Header/Cookie parameter) והשרת יודע לשייך את ה-Session ID של המשתמש ל-Data שמוצמד לאותו ID בשרת וכך הוא יודע אם היוזר הזדהה לפני כן וכו'.

ישנם מספר חסרונות בשיטה הזו, לדוגמא, אם יש לנו שרת ששומר את הסשנים ב-RAM ויכול להחזיק 1000 משתמשים ואנחנו רוצים לתמוך ב-3000 משתמשים. לא נוכל פשוט לשכפל את השרת כפול 3 ולבצע Load balancing אקראי ביניהם, כי משתמש שהזדהה מול שרת A ואח"כ יגיע לשרת B, שרת B לא יכיר את ה-Session ID שלו כיוון שאין לו את ה-RAM של שרת A. נצטרך או לחזק את השרת (חסרונות: א. יקר, ב. לא Scale-בילי, ג. מוגבל באיזשהו סף. ד. אי אפשר לפצל אותו לכמה חוות שרתים) או להשתמש בפתרונות עקומים כמו Load balancer עם Sticky Session (חסרונות: אתה מגביל בכמות סשנים פר שרת בלי חישוב אמיתי כמה מהם עדיין פעילים ובאיזו רמה ועוד).

כאן נכנס Stateless שגורס - תפיל הכל על הקליינט. רוצה לשמור מידע על היוזר כמו מה ה-User ID שלו? קח את המידע, תחתום/תצפין אותו ותשלח אותו לקליינט. הקליינט בתורו ישלח אליך את המידע החתום/מוצפן עם כל בקשה. השרת שיקבל את המידע יאמת את החתימה ואז יסמוך על המידע שנמסר שם.

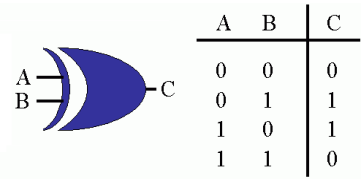
הכן משתמשים ב-Stateless?

- כמעט בכל מנגנון SSO
- נפוץ בעת כתיבת צד שרת ב-NodeJS
- בעת שימוש בארכיטקטורה מבוזרת - microservices

במאמר זה אסקור מספר טעויות נפוצות של מפתחים שבהן נתקלתי בעת ביצוע בדיקה למערכות אשר עושות שימוש בטכנולוגיה זו.

#1 - הצפנה במקום חתימה

טעות נפוצה היא להצפין את ה-State ובכך לנסות לקבל רווח כפול. גם הסתרה של ה-Data והמבנה שלו מהמשתמש וגם המשתמש לא יכול לבצע מניפולציה, כי הרי זה מוצפן. אז זהו, שיש הבדל בין חתימה להצפנה. בחתימה דיגיטלית יש אימות האם כל הבלוק של המידע לא עבר שום שינוי (או שחותמים דיגיטלית את כל המידע, או שמייצרים Hash ואותו חותמים, בהתאם לשיטות החתימה השונות). אימות החתימה מחזיר תשובה בינארית - או שהמידע חתום כראוי או שלא. כיוון שהחתימה היא פעולה קריפטוגרפית על כל המידע יחד והתשובה היא בינארית, הסיכוי לשנות את המידע ולהצליח להשאיר את החתימה נכונה הוא אפסי, כי צריך ליפול על צירוף מאוד מסויים.



A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

לעומת זאת בהצפנה. הפעולה Decrypt על מחרוזת (String) מוצפנת תחזיר תמיד מחרוזת מפוענחת. בואו ניקח דוגמא פשוטה - הצפנה באמצעות XOR, שבה לוקחים בית (Byte) מהמחרוזת המקורית, מבצעים איתה XOR על בית מהמפתח והתוצאה היא בית מוצפן.

אם תוקף יקח את התוצאה וישנה אותה, עדיין נוכל לבצע XOR עם בית מהמפתח ונקבל איזשהו בית בתוצאה. כך שאם נניח באפליקציה שלנו אנחנו מצפינים את המספר של ה-User ID. תוקף יכול לקחת את המחרוזת המוצפנת ולבצע איתה Brute Force מול השרת, שבה הוא כל פעם משנה ביט אחר ומנסה ליפול על מצב שבו הטקסט המפוענח יהיה מספר כלשהו. כיוון שהמספר מייצג User ID מה שחשוב זה שה-User ID יהיה מספר כלשהו שקיים במערכת (זה שונה מ-BF על Session ID, כיוון ששם צריך לבצע את זה על סשן פעיל, וכאן כיוון שזה State less כל יוזר-ID שקיים במערכת - יהיה רלוונטי).

להבדיל מחתימה שבה התוקף יצטרך לגרום לחתימה להיות נכונה - דבר שיש לו כמעט צירוף בודד. בהצפנה המטרה תהיה ליפול על "אחד מתוך". ברגע שהתוקף נפל על ערך אחר, הוא כבר הצליח.

בקיצור, אם אתם רוצים לאמת שהמידע לא עבר שינוי - כדאי להשתמש בחתימה דיגיטלית ולא לבצע Abuse למנגנון הצפנה... למי שרוצה "להחמיר", ניתן להשתמש באלגוריתמים כדוגמת GCM שיודעים לבצע authenticated encryption. שזה גם הצפנה וגם הגנה מפני שינוי.

#2 - חוסר במימוש מנגנון Expiration

שי חן, בכנס OWASP 2013 הדגים את אחת הבעיות ב-Stateless. הוא דיבר על קונטרולרים נסתרים ב-net.. החלק הרלוונטי לנו זה הפרמטר __ViewState של .NET. שלמעשה ממש מנגנון StateLess. .NET תומך במספר אופציות. החל מחתימה של המידע (כן, יש כאלה שגם מבטלים את הוואלידציה על החתימה...), המשך בהצפנה של התוכן (כשזה לא מוצפן, זה רק Base 64), ועד לחתימה פר משתמש (מה שמונע על הדרך CSRF לבקשות שעושות שימוש ב-viewState).

הוא ציין שדרך נפוצה היא להכניס ל-State את ה-Data source כרפרנס של מספר. נניח Data source 3, ואז בשרת לבצע מיפוי. מה שקורה, זה שאם יש בשרת מיפוי ל-Data sources נוספים, אנחנו יכולים למצוא ברשת (archive.org וכד') State חתום שטמון בו Data source אחר, נניח 2. ולהשתמש בסטייט הישן - שעדיין תקף כי החתימה תהיה חוקית לעולמים, כל עוד נשאיר את אותו מפתח ואותו אלגוריתם - ולתשאל Data source אחר לחלוטין.

ההמלצה שלו בהרצאה הייתה - תחליפו מפעם לפעם את המפתח.

פתרון חזק יותר הוא להוסיף ל-State את תאריך היצירה שלו (Creation time) ואז לבדוק שלא עבר ממנו X זמן (נניח שעה), מה שהופך את ה-State ל-Expired אחרי שעה. בשימוש שוטף יחודש את ה-Creation time ב-State כל בקשה, או כל פעם שמתקבלת בקשה שכבר עבר עליה חצי שעה.

#3: אי מימוש מנגנון Invalidation לעצירת מתקפה בזמן אמת

מקרה אמיתי מלקוח: תוקף ביצע פישינג על חשבון מייל מסוים במערכת והשיג את סיסמת המייל של משתמש Back office. למייל הוא לא יכל להתחבר כי היה 2 factor authentication (למרות שעם פישינג טוב יותר ניתן לעקוף גם את זה). מסתבר שהבחור שנפל לפישינג על המייל, השתמש באותה סיסמה למערכת Back office... התוקף הזדהה באמצעות המייל והסיסמה למערכת והתחיל "להשתולל" ולשנות חשבונות של משתמשים במערכת. בתוך דקות ספורות הבינו שמשהו לא טוב מתרחש, היו לוגים ולכן ניתן היה למצוא בקלות את החשבון שנפרץ שממנו מתבצעות הפעולות. אבל אז נתקלו בבעיה - איך "מעיפים"

אותו? אחרי הלוגין, ה-Session נוצר, כל המידע שרלוונטי (שם משתמש, הרשאות וכו') כבר בסשן, וגם אם מוחקים את המשתמש מה-DB, כל עוד הוא Logged-in הוא יכול להמשיך לבצע פעולות. למעשה לא הייתה למתכנתים/לסיסטם דרך לנתק Session ספיציפי.

במקרה ההוא היה מדובר באפליקציית State full, כך שניתן היה ללכת לפתרון ההזוי - לבצע Reset לשרת HTTP וכך למחוק את כל הסשנים וה-Session של התוקף בכללם. במקרה ומדובר באפליקציית State less, אפילו Reset לשרת לא היה עוזר. כי אין סטייט בשרת, והחתימה תהיה חוקית לתמיד...

הפתרון לאפליקציות State less הוא לכתוב קוד שירוץ על כל בקשה (סטייל Spring filter, NodeJS middleware ודומיהן) שבודק אם היוזר נמצא ברשימה שחורה ואם כן עוצר את עיבוד הבקשה. אפשר לבדוק כל פעם מול טבלת המשתמשים (במידה ויש שם שדה של Disabled/Invalidated), או להוריד משם אחת לדקה את המידע ואז לבדוק אותו מול קאש מקומי.

#4 - פגיעות ל-DoS בסיסי של שליחת Blob0-ים

בקריפטוגרפיה אחת הבעיות היא קושי העיבוד / ביצוע הפעולות הקריפטוגרפיות. הקושי מתבטא בזמן העיבוד. זו בעיה כל כך קשה, עד שכל תפיסת האבטחה מתבססת על כך שזה בעייתי, ושלפרוץ הצפנה X אורך Y זמן שהוא ארוך מאוד (נניח 20,000 שנים) ולכן אלגוריתם זה נחשב ל"בלתי פריץ", כי אז המידע כבר לא יהיה רלוונטי ולכן אין בעיה שייחשף.

בהקשר שלנו, חתימה דורשת הרבה כח עיבוד. למרות שהיום השרתים חזקים בהרבה מבעבר, עדיין ניתן להעמיס על CPU של שרת על ידי פעולות קריפטוגרפיות די בקלות. אם המשתמש שולט על המידע שנחתם (לדוג', אם בחתימה נכנסת כתובת ה-IP של המשתמש, וניתן לשלוח לה כל תוכן שהוא באמצעות X-Forwarded-For), ניתן לגרום לשרת לחתום מידע גדול בהרבה מזה שתוכנן וכך ליצור פעולה כבדה. אם אין הגבלה על כמות הבקשות, ניתן לשלוח שוב ושוב את הבקשה לחתימה ולהעמיס עליו עוד יותר.

לגבי התהליך של אימות החתימה, בהנחה ואין מגבלה ברמת הקוד, המגבלה תהיה לפי הקונפיגורציה של השרת, שזה לרוב כמה KB.

#5 - הצפנה סימטרית במקום א-סימטרית

בהצפנה סימטרית משתמשים באותו מפתח בשביל הצפנה ובשביל פיענוח. ההצפנה קלה יותר למימוש, והיא גם מהירה משמעותית מהצפנה א-סימטרית שבה יש מפתח אחד להצפנה ומפתח שני בשביל פענוח.

בארכיטקטורה שבה יש רכיב שחותם ורכיבים אחרים שרק מפענחים / מאמתים (לדוג', מערכת מיילים / SMS-ים ששולחת לינק שבתוכו מוצפן ה-ID של המשתמש, ומערכות אחרות שכשלוחצים על הלינק מגיעים אליהם, או שרת אחד של לוגין ושרתים אחרים של משאבים - נפוץ בארכיטקטורות SSO), שימוש במפתח סימטרי גורם לכך שפריצה לכל אחד מהשרתים שמחזיק את המפתח, תאפשר לחתום לינקים עבור שרתים אחרים. לעומת שימוש בהצפנה א-סימטרית שבה כדי לחתום לינקים יש צורך לפרוץ לשרת החותם. באמצעות שימוש בהצפנה א-סימטרית ממזערים את ה-attack surface למינימום.

#6 - שימוש ב-Hash במקום ב-Hmac

אחת הדרכים להגן מפני שינוי, היא לקחת את המידע המקורי, להוסיף לו מחרוזת שמשמשת כמפתח ואז לבצע עליהם Hash. ולהצמיד למידע את ה-Hash. המשתמש/התוקף שיש להם את המידע המקורי ואת ה-Hash לא יכולים לשנות אותו, כיוון שכדי לחתום כראוי צריך לדעת את המפתח.

מה שהסברתי כעת זה המנגנון הבסיסי, הוא מהיר בהרבה מחתימה דיגיטלית, ולכן קל יותר לשימוש. החסרון שלו זה שהוא סימטרי. כך שכל שרת שאמור לוודא את המידע, מכיל את המפתח ויכול גם לחתום מחדש את המידע.

מימוש הטכניקה כמו שתיארתי, פגיע ל-Length extension attack. לא אפרט אותה כאן כי כבר פירטו אותה מספיק, עיקרו של דבר, שבמקום לבצע רק Hash יש לבצע Hmac, שזה בסגנון:

```
MAC = hash(key + hash(key + message))
```

לינקים לקריאה נוספת ניתן למצוא בסוף המאמר.

מספר מילים לסיום

את ואתה שקורא/ת משהו שאני כותב ולא יצא לנו עדיין להיפגש פרונטלית. אם תראו אותי באיזה כנס או משהו, לא להסס, לגשת להגיד שלום. זהו. Be kind ☺

תודה לאפיק (cP) ולניר אדר (UW) שכבר שש שנים מוציאים לנו גיליונות. אין עליכם! ותודה לכל העוזרים. כרגיל, אפסק מגייסת פנטסטרים, יועצים ומדריכים. כאן חשוב להדגיש שאנחנו מגייסים רק שפיצים (והתנאים בהתאם!), וגם כאלה עם פוטנציאל להיות תותחים. העבודה באפסק היא אינטנסיבית - יש הכשרות פנימיות ומתקדמים מהר. אם את/ה בעלי ידע בתחום אבטחה המידע ובעניין, תוכל/י לשלוח קו"ח ל-jobs@appseclabs.com, או ישירות אלי israel@appsec-labs.com.

ישראל חורז'בסקי

סמנכ"ל טכנולוגיות, AppSec Labs

מקורות לקריאה נוספת

- http://blog.jdriven.com/2014/10/stateless-spring-security-part-1-stateless-csrf-protection/
- http://blog.jdriven.com/2014/10/stateless-spring-security-part-2-stateless-authentication/
- http://sec.cs.ucl.ac.uk/users/smurdoch/papers/protocols08cookies.pdf
- http://appsandsecurity.blogspot.co.il/2011/04/rest-and-stateless-session-ids.html
- https://blog.whitehatsec.com/hash-length-extension-attacks/
- https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks