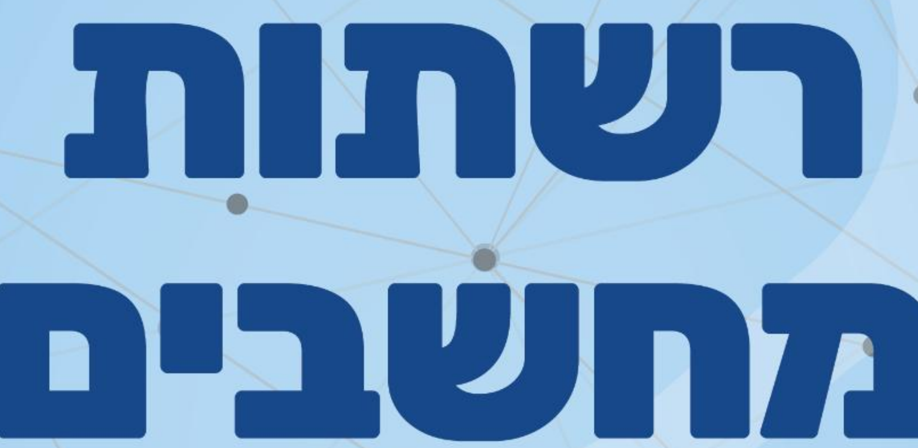


חלק ב'

ברק גונן

רשתות מחשבים חלק ב

גרסה 1.0 מרץ 2026

<u>כתיבה</u>

ברק גונן

<u>עריכה</u>

ד"ר שלומי בוטנרו

אין לשכפל, להעתיק, לצלם, להקליט, לתרגם, לאחסן במאגר מידע, לשדר או לקלוט בכל דרך או אמצעי אלקטרוני, אופטי או מכני או אחר – כל חלק שהוא מהחומר שבספר זה. שימוש מסחרי מכל סוג שהוא בחומר הכלול בספר זה אסור בהחלט, אלא ברשות מפורשת בכתב מהמרכז לחינוך סייבר.

© תשפ"ו, 2026. כל הזכויות שמורות למרכז לחינוך סייבר של קרן רש"י.

הודפס בישראל.

http://www.cyber.org.il

לבוגרי קודקוד סייבר, בהצדעה.

ברק

תוכן עניינים

תוכן עניינים 4
הקדמה 7
פרק 15: מבוא לסוקטים מאובטחים Secure Sockets 9
היסטוריה: מ-SSL ל-TLS 13
סיכום 17
פרק 16: הצפנות סימטריות 18
מבוא – אלפיים שנות הצפנה 18
מהי הצפנה? 18
צופן הזזה 21
צופן קיסר 22
צופן ויז'נר 24
האניגמה 27
הצפנות בלוק – Block Ciphers 28
ECB, CBC, CTR 30
סיכום 34
פרק 17: החלפת מפתחות סימטריים 35
מבוא 35
Diffie Hellman 35
RSA 40
בעיית פירוק מספר לראשוניים 41
אלגוריתם יצירת זוג מפתחות 43
הצפנה ופענוח 44
שימוש ב-RSA לטובת החלפת מפתחות 46
סיכום 47
פרק 18: Integrity ו-Authentication 48
פונקציות Hash 48
HMAC 53
חתימה דיגיטלית 55
סיכום – איך הכל מתחבר (בינתיים) 58
פרק 19: סרטיפיקטים 64
Certificate 64
Certificate Authority 70
חתימה על Root Certificate 73
CSR – Certificate Signing Request 76
סיכום ביניים 79
Certificate Verification 80
סוגי סרטיפיקטים 82
Certificate Chain 85
Basic Constraints 87
Certificate Revocation 89
CRL 90
OCSP 91
OCSP Stapling 94

CT Operators, Auditors, Monitors ... 96
Signed Certificate Timestamp ... 97
צפיה ב-SCT ... 100
סיכום ... 105

פרק 20: TLS 1.2 ... 106
הקדמה ... 106
TLS Records ... 107
TLS Handshake – מטרות ... 110
Handshake – בגרסת DH ... 113
Client Hello ... 113
Server Hello ... 115
Certificates ... 115
Server Key Exchange ... 117
Server Hello Done ... 118
Client Key Exchange ... 119
Change Cipher Spec ... 119
פענוח תקשורת מוצפנת ... 119
Client Finished ... 121
Change Cipher Spec, Server Finished ... 123
Handshake בגרסת RSA ... 124
שלבי יצירת מפתחות ... 126
שדה ה-SNI ... 128
RTT ... 129
Session Resumption ... 132
Session Ticket ... 133
סיכום ... 137

פרק 21: TLS 1.3 ... 138
מבוא – מוטיבציות לגרסה 1.3 ... 138
שינויים ב-Cipher Suites ... 139
שימוש ב-AEAD ... 141
TLS 1.3 Handshake ... 142
Client Hello, Server Hello ... 144
Extensions ... 146
Encrypted Extensions ... 147
Certificate ... 148
Certificate Verify ... 150
סיכום תהליך ה-Handshake ... 152
Session Resumption ... 152
סיכום ... 156

פרק 22: DNS ו-HTTP מעל TLS ... 158
הקדמה ... 158
HTTP/2 ... 158
Stream ID ... 158
HTTP PING ... 162
Packed Header ... 164
Server Push ... 166
DNS Over HTTPS ... 167
הסנפה של DNS רגיל ... 167
הסנפה של DNS מעל HTTP ... 168
צפייה ב-DoH ... 169
DoH עם Base64 ... 171
סיכום ... 177

פרק 23: QUIC 178
הקדמה 178
HTTP/2 – מקומות לשיפור 179
בעיית ה-Head Of Line Blocking 179
TCP Ossification 180
הורדת כמות ה-RTT 182
Connection Migration 184
מבוא ועקרונות QUIC 184
מעבר מ-TCP ל-QUIC 186
QUIC Header 186
QUIC Frames 188
TCP ACK 191
QUIC ACK 195
תהליך ה-Handshake והמעבר למוצפן 198
היררכיה של Connection, Stream, Packet, Frame 201
הצפנת Headerים 202
0-RTT "אמיתי" 203
TLS over UDP 204
DNS over QUIC 205
השוואה בין DoT, DoH, DoQ ו-DNS over DTLS 207
סיכום 209
מה צופן העתיד? 213
חומרי העשרה מומלצים 214

הקדמה

חלקו הראשון של הספר "רשתות מחשבים" הקנה ללומדים וללומדות את הידע הבסיסי לטובת הבנת עולם רשתות המחשבים. מטרת הספר היתה לענות על השאלה "**כיצד מחשבים מעבירים מידע זה לזה?**".

הספר פורסם בשנת 2014, ואיזכור עובדה זו היא משמעותית מכיוון שבזמן שחלף מאז התרחשו שינויים משמעותיים. בזמן שפורסם הספר, אחוז ניכר מהגלישה באינטרנט היתה בפרוטוקול HTTP לא מאובטח. כל מי שלמדו מהספר יכלו לשחזר בקלות את ההסנפות ולצפות בפקטות שעוברות ב-Wireshark בדיוק כפי שמוסבר ומודגם. מאז, לאט-לאט "נעלמו" והשתנו חלקים. תלמידים החלו לשאול "למה אנחנו לא רואים HTTP בהסנפה? ולמה לא רואים DNS? מה זה TLS שמופיע במקום זה?"

ובכן, התשובות לכך נמצאות בספר זה, "רשתות מחשבים חלק 2". הספר עונה על השאלה "**כיצד מחשבים מעבירים מידע זה לזה בצורה מאובטחת?**".

תוך כדי לימוד הפרקים בספר, נסקור כמה מהשינויים המרכזיים שהתרחשו באינטרנט ונחזיר לעצמנו את היכולת לפענח הסנפה עכשווית.

כדי לעשות זאת, נתחיל קודם כל בשאלה "מה צריך לכלול סוקט כדי להפוך אותו למאובטח?". נגלה שיש מספר חלקים לתשובה ונעבור עליהם חלק-חלק.

הצפנה – נסקור את התפתחות ההצפנות לאורך השנים, החל מהצפנות סימטריות פשוטות כגון קיסר, צופן החלפה וצופן ויז'נר, עבור דרך הצפנות סימטריות מודרניות כמו AES, וכלה בהצפנות א-סימטריות.

החלפת מפתחות – נבין שגם ההצפנה החזקה ביותר לא תצליח אם שני הצדדים לא יכולים להחליף ביניהם את מפתח ההצפנה. בעיית החלפת המפתחות הובילה לשני אלגוריתמים שבזכותם האינטרנט קיים – Diffie Hellman ו-RSA.

יכולת לזהות מסר לא מקורי – נבין כיצד עובד אלגוריתם גיבוב Hash ואיך אפשר להשתמש בו כדי לוודא שמסר שהועבר התקבל ללא שינוי.

אימות – איך מוודאים שהשרת שאנו מתקשרים איתו הוא אכן השרת האמיתי ולא מתחזה שנועד להונות אותנו? נבין מהו סרטיפיקט, נלמד אודות ה-Public Key Infrastructure, אותה תשתית יצירת אמון שמאפשרת לנו לבטוח בשרת מרוחק. נבין כיצד חתימה דיגיטלית מאפשרת לנו להבדיל בין שרת מקורי למתחזה.

לאחר שנלמד את הבסיס הזה, נראה איך רעיונות מורכבים מתחברים בכל פעם שאנחנו מבצעים גלישת אינטרנט. נראה איך ה-Handshake המורכב של פרוטוקול TLS (קיצור של Transport Layer Security) מאפשר להשיג את המטרה – להפוך סוקט רגיל לסוקט מאובטח.

נסיים בלימוד של גרסאות פרוטוקולים מתקדמות – HTTP גרסה 2 וגרסה 3, DNS מעל HTTP, ופרוטוקול QUIC המפתיע, שיגרום לנו לחשוב מחדש על חלוקת התפקידים במודל השכבות ועל פרוטוקול UDP.

אלפי שנים של צבירת ידע אנושי פועלות מאחורי הקלעים בכל פעם שאנחנו מבצעים גלישת אינטרנט. עולם ידע חדש מחכה לנו, בואו נצלול אליו.

פרק 15: מבוא לסוקטים מאובטחים Secure Sockets

בחלקו הראשון של ספר הרשתות, עסקנו רבות ביצירת סוקטים בין לקוח ושרת. השאלה שהעסיקה אותנו היתה "איך אפשר לגרום למידע לעבור בין שני צדדים?". תוך כדי הלימוד, סקרנו רכיבי רשת שונים וראינו כי המידע שנשלח בין הצדדים עשוי לדלג בין כמה וכמה ראוטרים בדרכו אל היעד. כל ראוטר כזה יכול, לפי רצונו של הגורם המפעיל אותו, להסניף את המידע שעובר דרכו ולאפשר לצד שלישי גישה למידע שעובר.

בחלק זה נשאל את השאלה – מה צריך לקרות כדי שהתקשורת בין הצדדים תהיה מאובטחת?

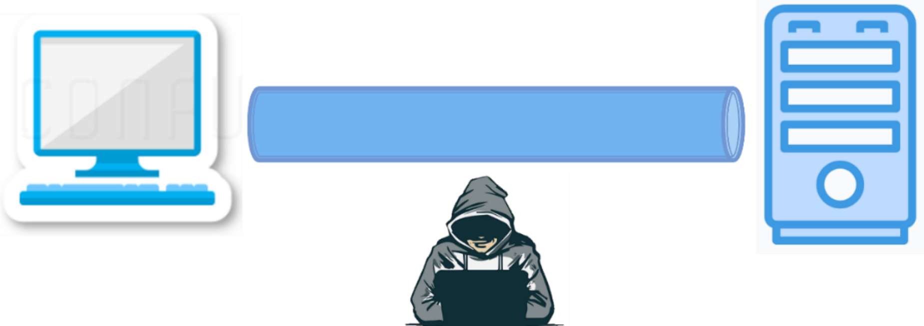
תקשורת מאובטחת מניחה שהערוץ שהיא עוברת עליו אינו פרטי. דמיינו לדוגמה שאתם בכיתה וכתבתם פתק לתלמיד אחר: "נפגשים בשעה שלוש". על הפתק כתבתם את שם התלמיד, קיפלתם אותו, ומסרתם את הפתק למי שיושב לידכם. מי שיושב לידכם יכול להעביר את הפתק הלאה, אך בלי שום מאמץ הוא יכול לפתוח את הפתק, לקרוא מה כתבתם ולדעת שהפגישה בשלוש, גם בלי שהפתק היה מיועד אליו. ה"ערוץ" ביניכם לבין היעד אינו פרטי.

חזרה לעולם הטכנולוגיה. התיישבתם עם לפטופ בבית קפה והתחברתם לרשת ה-WiFi. כעת גלשתם לאתר כלשהו, לדוגמה לבנק שלכם. נוצר סוקט בין הלקוח, הלפטופ שלכם, לבין השרת, שרת הבנק. הסוקט נבנה מעל שכבת הרשת שיש בה כמה וכמה ראוטרים בדרך, שהם כמו התלמידים שמעבירים פתק בכיתה. הם יכולים להעביר את הפקטות שלכם בלי לקרוא אותם, אך אין שום קושי טכנולוגי לקרוא את תוכן המידע ששלחתם. הבעלים של בית הקפה, שיש לו גישה לראוטר, יכול להתקין תוכנת הסנפה על הראוטר וכך כל הפקטות שנשלחות על ידכם לאינטרנט – נשמרות אצלו. סיסמת חשבון הבנק שלכם עברה על גבי אחת הפקטות וכעת היא עלולה להיות בידיו. ולא רק הוא יכול להאזין לכם. כל ראוטר שנמצא ביניכם לבין השרת יכול לקרוא את הפקטות ששלחתם וקבלתם.

אם כך, אנחנו מבינים שכל עוד הסוקט שלנו מעביר מידע בלי לעשות שום דבר שמגן על המידע, ניתן להאזין למידע. אך לא רק זאת – גורם זדוני יכול גם לשנות את המידע. נחזור לדוגמת התלמידים שמעבירים פתקים בכיתה. אחד התלמידים בכיתה רוצה לשבש לכם את התכניות. לא רק שהוא קורא את שעת הפגישה שלכם, הוא מוחק את המילה "שלוש", ומחליף אותה ב"ארבע". החבר אליו מיועד הפתק קיבל הודעה "נפגשים בשעה ארבע", הוא מאשר שבסדר, בלי לדעת שמישהו שינה את התוכן.

כל אחד מהתלמידים בדרך מכם אל החבר שלכם הוא מה שנקרא MITM – קיצור של Man In The Middle. בתור MITM הוא יכול להסתפק בקריאת תוכן ההודעות שלכם, או ממש לשבש ולשנות אותן, מה שנקרא מתקפת MITM.

האיור הבא ממחיש גורם זדוני שיש לו גישה לסוקט (לדוגמה, שולט באחד הראוטרים בדרך).



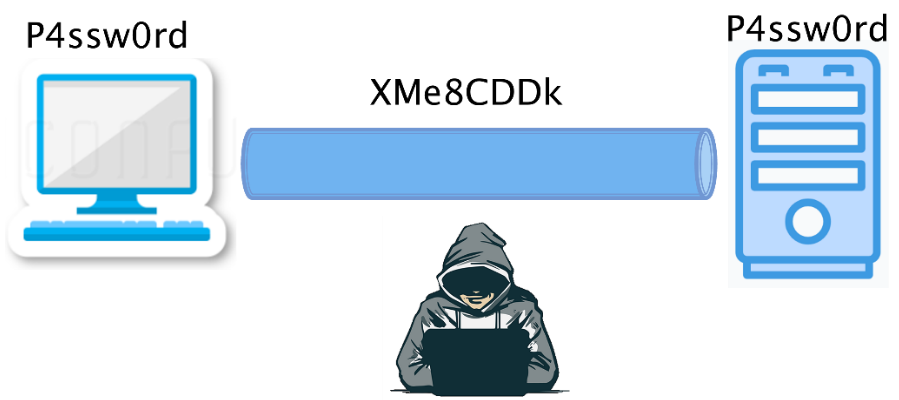
אין לנו יכולת להגן על עצמנו ממצב זה. הנחת העבודה שלנו היא שהסוקט בינינו לבין שרת שנפנה אליו לעולם יעבור דרך ראוטרים שאינם שלנו. הרי לא נוכל להחזיק סיב אופטי פרטי בין המחשב שלנו לכל שרת בעולם. למרות הנחת העבודה הבעייתית, שהתקשורת שלנו תמיד חשופה לגורם זר, נרצה שהתקשורת בינינו לבין השרת תהיה מאובטחת.

סוקט מאובטח, או Secure Socket, צריך לקיים חמש תכונות:

- חיסיון, או באנגלית Confidentiality
- שלמות, או באנגלית Integrity
- זמינות, או באנגלית Availability
- אימות, או באנגלית Authentication
- אי הכחשה, או באנגלית Non Repudiation

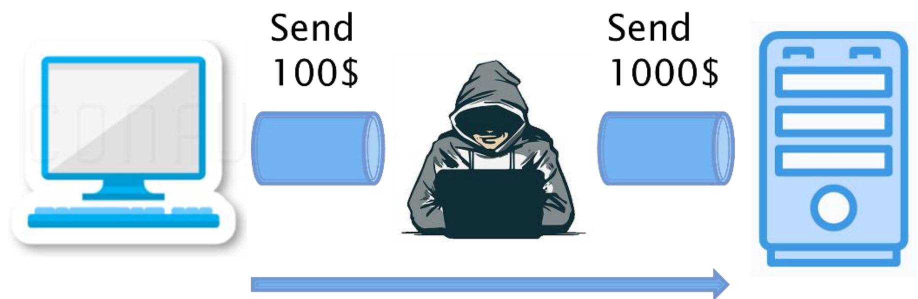
נציין מראש, שנושא הזמינות (Availability) עוסק ביתירות של מערכות, בגיבויים, בעמידות למתקפות מניעת שירות וכדומה. לא נעסוק בו במסגרת ספר זה.

Confidentiality: כדי לשמור על חסיון המידע המועבר מעל הסוקט, נדרש למנוע מההאקר לפענח את המידע שהוא קולט.



כדי למנוע מההאקר להבין מה המסר ששלחנו, נרצה לבצע עליו איזושהי מניפולציה שתהפוך אותו ממסר גלוי למסר מוצפן. המסר המוצפן יהיה בעל תכונה, שמי שמאזין לו אינו מסוגל להבין אותו אך הצד שאליו המסר מיועד מסוגל להפעיל פונקציה כלשהי שהופכת את המניפולציה שמפעיל השולח כך שהמסר המוצפן חוזר להיות מסר גלוי. אם הרעיון של מסר גלוי, מוצפן, ופונקציה שממירה ביניהם אינו מובן לכם כרגע – אל דאגה, נקדיש לכך חלק משמעותי מההמשך.

Integrity: כדי למנוע מההאקר שיושב בתווך לשנות את המידע שעובר דרכו, נצטרך למצוא שיטה שבה כל צד לשיחה יוכל לזהות אם ההודעה שהתקבלה אצלו שונתה בדרך ואינה ההודעה המקורית ששלח הצד השני.



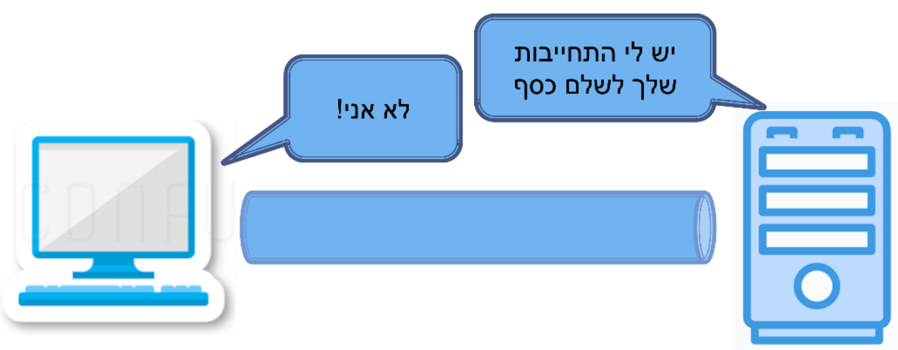
בדוגמה שבתרשים יש הודעה שהועברה במקור מאיתנו אל הבנק שלנו, עם בקשה לשלם למישהו 100$. ההאקר שינה את ההודעה כך שהבנק יבין כי אנו מבקשים להעביר 1000$.

הצפנה אינה פותרת את הבעיה. אם ההודעה שלנו מוצפנת, אמנם להאקר יהיה מסובך לשנות אותה בצורה כה מדוייקת, אולם עדיין אפשר להכניס בה שינויים אקראיים בתקווה לגרום נזק. מי שמקבל פקטה שעברה מעל סוקט צריך להיות בטוח שהמידע שעבר בה נאמן למקור ולא שונה בדרך.

Authentication : כדי למנוע מההאקר להתחזות לצד השני בשיחה, צריך למצוא דרך שתאפשר לנו לזהות ולאמת שהצד השני לשיחה הוא אכן מי שהוא טוען שהוא.

בדוגמה שבתרשים, האקר התחזה לבנק שלנו והפיל אותנו בפח. בכך הוא ניטרל את מנגנון ה-Confidentiality, וכיוון שאנחנו מאמינים שההאקר הוא הצד השני לשיחה, נשלח את המידע בצורה שהוא יוכל לפענח. לאחר שההאקר פענח את ההודעה, הוא עלול לשלוח לבנק האמיתי שלנו הודעה דומה אך עם שינוי, כגון 1000$ במקום 100$, ובכך לעקוף גם את מנגנון ה-Integrity.

Non Repudiation : התבוננו בתרשים הבא. ישנה עסקה, ואחד הצדדים לביצוע העסקה מתכחש שהוא לקח בה חלק. לדוגמה, הבטחנו להעביר כסף תמורת שירות כלשהו שקיבלנו. מגיע מי שמחזיק את ההתחייבות, ומבקש את הכסף. אנחנו מכחישים שביצענו את העסקה וטוענים שמישהו אחר חתם על ההתחייבות בשמנו, או שאנחנו חתמנו על התחייבות לשלם, אבל סכום קטן יותר.



מה אפשר לעשות? צירוף של שתי תכונות שראינו מונע התכחשות. הראשונה היא Authentication. יש לנו דרך לזהות מי היישות ששלחה את המידע, כלומר אם הגנתם על הסוקט ששימש לתקשורת באמצעות מנגנון אימות, אין לצד השני שתקשר אתכם אפשרות להכחיש שהוא שלח את ההודעה.

עדיין, הצד השני יכול לטעון שהוא שלח הודעה, אבל מה שאתם מחזיקים אינו ההודעה המקורית שהוא שלח. תכונת ה-Integrity פותרת את הבעיה הזו. אם השתמשתם במנגנון בדיקת שלמות, אתם יכולים להוכיח שההודעה לא שונתה בדרך.

התוצאה היא שאפשר לענות למי שמתכחש לעסקה: יש לי הוכחה שאתה ביצעת את העסקה, ויש לי הוכחה נוספת שהעסקה שאתה ביצעת לא שונתה על-ידי אף אחד אחר. אם פתרנו את בעיות ה-Integrity וה-Authentication, פתרנו את בעיית ה-Repudiation.

היסטוריה: מ-SSL ל-TLS

השנה – 1991. הגרסה הראשונה של פרוטוקול HTTP, גרסה 0.9, מפורסמת לציבור. יחד עם פרסום הפרוטוקול, הצוות של CERN, שהמציא את הפרוטוקול, מפרסם קריאה לאנשי טכנולוגיה לכתוב "HTTP Client". במילים של היום – Browser, או דפדפן.

Things to be done

There are many of these ...if you have a moment, take your pick! Some of these would make good projects, or would be a way your organization could contribute to the building of the web.

There are also special lists of little things to do for each existing project such as the line mode browser , the NeXT browser , the common library , and the server .

Client side

More clients
: Clients exist for many platforms, but not all. Editors only exist on the NeXT, but will be really useful for sourcing info and group work. (Group editor?)

Search engines
: Now the web of data and indexes exists, some really smart intelligent algorithms ("knowbots?") could run on it. Recursive index and link tracing, Just think...

אתר האינטרנט הראשון, עם הצעה לפתח דפדפן ומנוע חיפוש
מקור: https://info.cern.ch/hypertext/WWW/Bugs.html

את כפפת הדפדפן מרימה חברה בשם Netscape. הדפדפן שלה, Netscape Navigator, מופץ ב-1994 ועד מהרה הופך להיות הכלי הנפוץ ביותר לגלישה באינטרנט, עם יותר מ-90% נתח שוק.

חברת Netscape מחליטה להשקיע בטכנולוגיה של סוקטים מאובטחים, כאלו שיספקו את היכולות שסקרנו. בשנת 1994 ממציאה נטסקייפ את ה-SSL, ראשי תיבות של Secure Socket Layer. גרסה 1.0.

הרעיון המרכזי של SSL, הוא לקחת סוקט TCP רגיל, כלומר סוקט TCP שעבר בסך הכל Three Way Handshake, ודרך העברה של פקטות מסויימות בין השרת והלקוח, לדאוג לעקרונות החשובים של סוקט מאובטח: להצפין את התקשורת, לוודא שכל צד הוא מי שהוא טוען שהוא, לוודא שכל הודעה שמגיעה היא מקורית וללא שינויים.

SSL הוא פרוטוקול, שמגדיר את אותן פקטות מסויימות – בשלב זה נהיה מעט מסתוריים לגביהן אבל הן יובהרו בפרקים הבאים – שיוצרות את האבטחה.

אם כך, לאיזו שכבה ממודל השכבות שייך פרוטוקול SSL? חישבו על כך לרגע לפני שתמשיכו.

ובכן, SSL אינו שייך לשכבת התעבורה. הפקטות שנשלחות ב-SSL אינן משנות שדות של TCP ולמעשה מבחינת TCP אין זה משנה אם יש שימוש ב-SSL או לא. SSL מתייחס ל-TCP כמו ש-TCP מתייחס ל-IP: פרוטוקול שיוצר בסיס, עליו מוסיפים יכולות נוספות.

אך SSL גם אינו שייך לשכבת האפליקציה. אמנם, התרגלנו שכל מה שעובר מעל שכבת התעבורה הוא אפליקציה, אבל במקרה של SSL אין זה כך. מעליו יכול לעבור פרוטוקול של אפליקציה להורדת קבצים, אפליקציה לגלישה באינטרנט, אפליקציית מייל ועוד. כלומר שכבת האפליקציה עוברת מעל SSL. במילים פשוטות, SSL מייצר סוקט מאובטח. יש הקוראים לו "שכבה 4.5", כלומר שכבה שנמצאת בין התעבורה לאפליקציה. זה לא מונח מדוייק אבל עוזר לזכור את המיקום שלו.

בשנים הבאות, 1995 ו-1996, מוציאה נטסקייפ את SSL 2.0 ו – SSL 3.0.

חברת נטסקייפ מחוסלת על-ידי מיקרוסופט, סופית בשנת 2008 אבל מעשית ב-1998. הרעיון של סוקט מאובטח עובר מחזקתה של נטסקייפ לצוות תקינה בינלאומי, שעל הבסיס של SSL יוצר את TLS, קיצור של Transport Layer Protocol. מדי פעם עדיין נתקלים בכינוי SSL; זיכרו ש-SSL הוא פשוט גרסה ישנה של TLS הנפוץ כיום. הגרסה הראשונה של TLS 1.0 יוצאת ב-1999. מאז יצאו הגרסאות הבאות:

- TLS 1.1 בשנת 2006
- TLS 1.2 בשנת 2008
- TLS 1.3 בשנת 2018

מה קרה לנטסקייפ? הסיפור של נטסקייפ אינו נדרש לשם המשך הלימוד, אבל אתם חובבי טכנולוגיה וכסף גדול, אם מעניין אתכם להבין כיצד עולם הטכנולוגיה שלנו עוצב כפי שהוא כיום, ואם אתם חושבים להשפיע על עולם הטכנולוגיה של העתיד, בבקשה.

באמצע שנות התשעים נטסקייפ שולטת בכ-90% משוק הדפדפנים. המודל המסחרי שלה – מכירת רישיונות שימוש. אם אתם רוצים רישיון לשימוש בדפדפן, תשלמו מספר דולרים. בשנת 1995 נטסקייפ מנפיקה מניות בבורסה האמריקאית ונסחרת בשווי של כמעט 3 מיליארד דולר. היום אנחנו רגילים לראות חברות טכנולוגיה מונפקות בסכומים עצומים, אבל ב-1994 זה היה סכום חריג ומפתיע, ובפרט לחברה שהוקמה לפני פחות משנתיים ועדיין מפסידה כסף. ההצלחה המסחררת לא נמשכת זמן רב. מיקרוסופט נכנסת למלחמה עם נטסקייפ על שוק הדפדפנים, ומנצחת אותה בענק. בשנת 1998 נטסקייפ כבר מפטרת רבים מאנשיה, היא נמכרת, נתח השוק של הדפדפן שלה

יורד לאחוז אחד בלבד לפני שהיא נסגרת סופית ב-2008. מה שנותר כיום מנטסקייפ הוא בעיקר הדפדפן Mozilla, שהחל כפרוייקט צד של נטסקייפ, ושפת התכנות JavaScript.

מה עשתה מיקרוסופט כדי להגיע מנתח שוק אפסי בשוק הדפדפנים לכיבוש השוק ולסגירת המתחרה? באותה תקופה, אף יותר מאשר היום, מיקרוסופט שלטה ללא עוררין בשוק מערכות ההפעלה. כל מחשב חדש הגיע עם מערכת ההפעלה החדשה שלה – חלונות. מיקרוסופט מבצעת שני דברים. ראשית, הדפדפן שלה, Internet Explorer, מותקן אוטומטית עם כל התקנה של חלונות. שנית, מחירו של הדפדפן – אפס.

הניצחון המוחץ של מיקרוסופט עורר את רשות ההגבלים העסקיים של ארה"ב. מאחורי השם המשעמם והלא מובן "הגבלים עסקיים" עומדת מטרה, למנוע מעסקים יכולת להתארגן בצורה שתפגע בצרכנים, בין על-ידי הפקעת מחירים ובין על-ידי עצירת חדשנות טכנולוגית. משרד המשפטים האמריקאי נכנס ב-1998 לתביעת ענק נגד מיקרוסופט. הדרישה – את מיקרוסופט צריך לפרק.

ההחלטה לתבוע את מיקרוסופט בשם הצרכן האמריקאי וממשלת ארה"ב לא היתה החלטה פשוטה. ניתן לטעון, שמיקרוסופט ניצחה את שוק הדפדפנים בצורה תחרותית. מה הבעיה בכך שאנשים מעדיפים לקבל מוצר חינמי? ייתכן גם שבשלב מסויים האקספלורר של מיקרוסופט היה דפדפן טוב יותר. מחיר טוב יותר, מוצר טוב יותר, איפה הבעיה? האם המדינה צריכה לקבוע את זהות המנצחים בשוק חופשי?

העניין הוא, שלמיקרוסופט לא היה עניין בשוק הדפדפנים כשלעצמו. מיקרוסופט תכננה לשכפל את המונופול שלה, השליטה שלה במערכות הפעלה, אל כל שוק התוכנות ומשם אל שליטה באינטרנט. נטסקייפ היתה רק אבן ראשונה בדרך אל היעד. זה מה שהבינה רשות ההגבלים ולכן היא תבעה לפרק את מיקרוסופט.

הבסיס שעליו עמדה מיקרוסופט היה שליטה במערכת ההפעלה. באמצעות שליטה זו אפשר לא רק להציע התקנה אוטומטית של תוכנות, אלא גם "התאמה" טובה יותר של מערכת ההפעלה לתוכנות שיצרן מערכת ההפעלה רוצה בקידומן. אפשר, תאורטית, לגרום לכך שדפדפן של מיקרוסופט ירוץ בצורה מהירה וחלקה יותר תחת מערכת הפעלה של מיקרוסופט, מאשר דפדפן מתחרה. אפשר למנוע ממתחרה להכניס שיפורים ושינויים שונים, שדורשים תמיכה של מערכת ההפעלה. אפשר להסתיר את הממשק של מערכת ההפעלה, כך שיצרני תוכנות חיצוניים לא יצליחו לתכנת או לפתור בעיות. שליטה בשוק מערכות ההפעלה מאפשרת לדכא כל מתחרה בשוק התוכנה – דפדפנים, תוכנות משרדיות (כגון open office), משחקים ועוד.

אם כך, למיקרוסופט כמונופול מערכות הפעלה יש עוצמה יוצאת מן הכלל, המאפשרת לה גם להגביל כניסה של מתחרים וגם לפתח מוצרים מתחרים ולמכור אותם באפס כסף. מבחינה עסקית, מיקרוסופט היתה עשויה למצוא את עצמה בנקודה שבה היא מציעה ליצרני תוכנה "הצעה שאי אפשר לסרב לה": עליכם להימכר לנו, או שנהרוג לכם את העסק. ולא רק דפדפנים, אלא כל תוכנה. זו אגב, מהות תביעת ההגבלים העסקיים שהתנהלה נגד פייסבוק.

לפי הנטען פייסבוק השתלטה על מתחרים פוטנציאליים באמצעות הצעות "נקנה אותך או נקבור אותך" וכך השתלטה על ווצאפ ואינסטגרם. התביעה כנגד פייסבוק נדחתה בנימוק שהיא אינה מונופול.

הצעד הבא, לו מיקרוסופט היתה מורשית להמשיך באסטרטגיה שלה, היה ככל הנראה השתלטות על שוק מנועי החיפוש. שליטה בדפדפן ובמערכת ההפעלה עשויה להוביל לכך, ומיקרוסופט אכן ניצלה את שליטתה בדפדפן כדי להציע בתוכו את מנוע החיפוש MSN Search. הצעד הבא, השתלטות על המסחר באינטרנט. מיקרוסופט כבר החלה בכך בסוף שנות התשעים, באמצעות רכש של חברת הנופש Expedia ושל חברת ההשמה Sidewalk.

בסוף 2001, התביעה נגד מיקרוסופט מסתיימת בפשרה. מיקרוסופט לא מפורקת, אך נמנע ממנה להמשיך ולהשתמש במונופול שלה בשוק מערכות ההפעלה כדי להשתלט על שווקים נוספים. עבור נטסקייפ זה כבר מאוחר מדי, אך נוצרת הרתעה – מיקרוסופט לא יכולה להשתמש במונופול שלה על מערכת ההפעלה כדי להשתלט על שוק התוכנה בעולם. מספר חברות שמתחילות לפעול באותה תקופה נהנות מהגנה. ביניהן – גוגל, אמזון, אפל. ייתכן שאלמלא מיקרוסופט היתה מקבלת כרטיס צהוב מרשות ההגבלים של ארה"ב, היא היתה שולטת היום במרבית כלכלת האינטרנט, השווי שלה היה גדול יותר מאשר השווי של כלכלות גרמניה ויפן ביחד. הגודל והכח הכלכלי האדיר היה יוצר הרתעה הפוכה – במקום שחברת ענק תחשוש ממדינות, מדינות יחששו מחברת הענק. דבר זה היה מעניק חסינות, למעשה, מפיקוח של הגבלים עסקיים, מה שהיה מגביר חיסול של תחרות בכל מקום שבו היתה החברה רוצה לפעול.

15.1 תרגיל – הסנפת TLS

בהסנפת אינטרנט תוכלו למצוא גרסאות 1.2 ו-1.3. כל הגרסאות האחרות, בין ש-SSL ובין ש-TLS מגרסאות ישנות יותר, נחשבות לא בטוחות. כלומר, או שיש להן פרצות ידועות, או שיש חשד שהן ייפרצו בקרוב. נמחיש זאת עכשיו.

בצעו הסנפה כלשהי, גלשו למספר אתרים כרצונכם. השתמשו בפילטר TLS.

תוכלו לראות פקטות משלושה סוגים: TLS 1.2, TLS 1.3 וייתכן שגם פקטות של פרוטוקול נוסף שנקרא QUIC, שעושה שימוש ב-TLS. על כל הפרוטוקולים הללו נרחיב בהמשך.

להלן דוגמה של הסנפה מפולטרת TLS. הפקטות הירוקות הן כאלו שההצפנה שלהן נפתחה. בהמשך נלמד לעשות גם את זה. הסיבה שנבחרה הסנפה עם הצפנה פתוחה, היא כדי שתוכלו להתרשם ולראות שכיום פרוטוקול HTTP עובר מעל TLS.

tls

No.	Source	Destination	Protocol	Length	info
7003	35.212.192.196	192.168.1.208	TLSv1.3	341	New Session Ticket
7004	35.212.192.196	192.168.1.208	TLSv1.3	341	New Session Ticket
7005	35.212.192.196	192.168.1.208	HTTP2	116	SETTINGS[0], WINDOW_UPDATE[0]
7006	35.212.192.196	192.168.1.208	HTTP2	85	SETTINGS[0]
7008	192.168.1.208	35.212.192.196	HTTP2	85	SETTINGS[0]
7010	13.226.2.17	192.168.1.208	TLSv1.2	831	Application Data
7011	13.226.2.17	192.168.1.208	TLSv1.2	85	Application Data
7022	74.125.21.113	192.168.1.208	HTTP2	1036	SETTINGS[0], WINDOW_UPDATE[0]
7023	192.168.1.208	74.125.21.113	HTTP2	85	SETTINGS[0]
7024	74.125.21.113	192.168.1.208	HTTP2	85	SETTINGS[0]
7025	74.125.21.113	192.168.1.208	HTTP2	1036	SETTINGS[0], WINDOW_UPDATE[0]
7028	74.125.21.113	192.168.1.208	HTTP2	585	HEADERS[1]: 200 OK
7030	74.125.21.113	192.168.1.208	HTTP2	1013	DATA[1] (JPEG JFIF image)
7031	74.125.21.113	192.168.1.208	HTTP2	93	PING[0]
7033	192.168.1.208	74.125.21.113	HTTP2	93	PING[0]
7034	74.125.21.113	192.168.1.208	HTTP2	183	HEADERS[5]: 200 OK
7035	74.125.21.113	192.168.1.208	TLSv1.3	1414	[TLS segment of a reassembled PDU]
7037	142.250.105.132	192.168.1.208	HTTP2	1034	SETTINGS[0], WINDOW_UPDATE[0]
7038	192.168.1.208	142.250.105.132	HTTP2	85	SETTINGS[0]
7039	142.250.105.132	192.168.1.208	HTTP2	85	SETTINGS[0]
7040	74.125.21.113	192.168.1.208	HTTP2	308	DATA[5] (JPEG JFIF image)
7041	74.125.21.113	192.168.1.208	HTTP2	184	HEADERS[3]: 200 OK
7043	74.125.21.113	192.168.1.208	HTTP2	573	DATA[3] (PNG)
7044	142.250.105.132	192.168.1.208	HTTP2	361	HEADERS[1]: 200 OK
7046	142.250.105.132	192.168.1.208	TLSv1.3	1414	[TLS segment of a reassembled PDU]
7049	142.250.105.132	192.168.1.208	TLSv1.3	1414	[TLS segment of a reassembled PDU]
7054	142.250.105.132	192.168.1.208	TLSv1.3	1414	[TLS segment of a reassembled PDU]
7055	142.250.105.132	192.168.1.208	HTTP2	775	DATA[1] (JPEG JFIF image)
7057	142.250.105.132	192.168.1.208	HTTP2	93	PING[0]
7058	192.168.1.208	142.250.105.132	HTTP2	93	PING[0]
7063	162.125.21.2	192.168.1.208	TLSv1.2	99	Application Data
7102	192.168.1.208	142.250.75.33	QUIC	1292	Initial, DCID=6de85e26da6adbe8, PKN
7110	142.250.75.33	192.168.1.208	QUIC	1292	Initial, SCID=ede85e26da6adbe8, PKN
7115	142.250.75.33	192.168.1.208	QUIC	1292	Handshake, SCID=ede85e26da6adbe8, P

סיכום

סקרנו את עקרונות התקשורת המאובטחת. תמיד נניח שהערוץ שאנחנו מדברים עליו אינו פרטי, ולמרות זאת נצטרך להשיג אבטחה. עקרונות האבטחה הם Confidentiality, Integrity, Availability, Authentication ו- Non Repudiation. כעת, לאחר שהבנו את העקרונות, נוכל להתחיל בדרך המורכבת לעבר הפתרון. הצעד הראשון, בו נדון בפרק הבא: הצפנה.

פרק 16: הצפנות סימטריות

מבוא – אלפיים שנות הצפנה

בשנות ילדותי, כנראה בהשפעת סדרת הטלויזיה "מנהרת הזמן", חלמתי שאני נוסע בזמן ומגיע אל מקומות שונים ומרתקים. עשרות שנים אחר כך, אני מודה, אני עדיין משתעשע לפעמים בשאלת המסע בזמן אבל מהיבט אחר: נניח שהיו לוקחים היום אדם כמוני או כמוכם הקוראים, על כלל הידע והכישורים שצברנו בחיים, ומעבירים אותנו לתקופה אחרת בהיסטוריה. האם יש לנו כישורים מועילים כלשהם שיהיו רלבנטיים לתקופות אחרות? האם הכישורים האלה בתוספת הידע שצברנו על מאורעות היסטוריים שונים, היו יכולים לאפשר לנו לשנות את ההיסטוריה? או שיותר סביר שעם כישורים כמו שלנו, שרובם טכנולוגיים, היינו נפטרים תוך זמן קצר מרעב ומחלה?

נניח לדוגמה שמכונת זמן הקפיצה אתכם לשנת 66 לספירה, מרד היהודים ברומאים. אתם רוצים לשנות את תוצאות המרד ובמקום לסיים בתוצאה שבית שני נחרב, להפוך את ההיסטוריה כך שממלכת יהודה מביסה את הרומאים. זה אפשרי? איזו תועלת יש לידע הטכנולוגי שלכם כ-2000 שנה לפני המצאת המחשב? מבחינות רבות אתם חסרי תועלת, תוכלו אולי לזרוק חנית או לירות בקשת, מה שבטח לא ישנה את תוצאות המערכה. אבל בכל זאת יש כישור אחד שבו הייתם יכולים לשנות את ההיסטוריה כמעט בכל תקופה שהייתם מגיעים אליה, לשנות תוצאות של קרבות וגורלות של מדינות ואנשים: קריאת הודעות מוצפנות של צבא האויב. החל מהצבא הרומאי ועד היום, מדינות וצבאות משתמשים בצפנים. ואת רוב ההצפנות יש לנו כיום ידע לפצח, ואפילו די בקלות.

אנחנו יוצאים עכשיו למסע מרתק שייקח אותנו כאלפיים שנה אחורה, מתקופת הרומאים ועד מלחמת העולם השניה. וכשנסיים לעשות את זה, כל מה שתצטרכו לעשות כדי לשנות את ההיסטוריה הוא לייצר מכונת זמן.

מהי הצפנה?

ראשית, נגדיר בצורה מסודרת מהי הצפנה, ומה ההבדל בין הצפנה לבין קידוד או דחיסת מידע.

הצפנה היא תהליך שבו אנחנו משנים מידע גלוי, כלומר שניתן להבין אותו, למידע שאינו גלוי – קרי מוצפן.

כדי ששינוי של מידע ייחשב הצפנה, צריכים להיות לו כמה מאפיינים. המאפיין הראשון הוא אלגוריתם, כלומר פעולה שיטתית כלשהי שמבוצעת על המידע. בהמשך הפרק נסקור אוסף של שיטות שפותחו עם השנים, אך כרגע נתמקד ברעיון שנדרשת שיטה. נניח שהיינו רוצים להצפין את המסר הגלוי "רשתות מחשבים" אבל בלי שיש לנו שיטה כלשהי. היינו מחליפים כל תו באופן אקראי בתו אחר ומקבלים לדוגמה "אחתשפ מגאאשל". אין דרך לחזור חזרה אל המסר הגלוי. אם ההחלפה היא אקראית, המסר הגלוי יכול להיות כל דבר, גם "חביתת עגבניה". צריך להיות אלגוריתם, שיודע להפוך כל מסר גלוי למסר מוצפן אחד – ואחד בלבד.

המאפיין השני – התהליך צריך להיות הפיך. כלומר, בהצפנה אין אפשרות שמסר מוצפן יתקבל כתוצאה של כמה אפשרויות של מידע גלוי. לדוגמה, שיטה שבה סוכמים את ערכי הגימטריה של משפט כלשהו (לדוגמה – "רשתות מחשבים" = 1706) אינה הצפנה, כיוון שיכולים להיות כמה מסרים שיסתכמו באותו ערך, ואין לדעת מה מהם המידע הגלוי (1706 שווה גם "אחר צהרים של פורענות").

המאפיין השלישי – צריך סוד. הסוד, או מפתח ההצפנה, ידוע רק לשני הצדדים לתקשורת. בואו נבין מה זה אומר מבחינה מעשית. כאשר שני צדדים יבחרו להצפין מידע, הם יבחרו שני דברים: שיטת הצפנה ומפתח הצפנה. שיטת ההצפנה יכולה להית גלויה לכולם, כולל למי שמנסה להאזין להם. מפתח ההצפנה הוא סוד שנעשה בו שימוש בשיטת ההצפנה כחלק מתהליך ההפיכה של המידע הגלוי למידע מוצפן. מי שאין לו את מפתח ההצפנה לא אמור להיות מסוגל לפענח את המידע הגלוי מתוך המידע המוצפן, למרות שהוא יודע מה אלגוריתם ההצפנה. אחד מהיסודות של תורת ההצפנה קובע, שהסודיות של מערכת הצפנה לא צריכה לנבוע מכך ששיטת ההצפנה סודית, אלא מכך שמפתח ההצפנה סודי (Kerckhoff's principle).

כדוגמת פתיחה ניקח צופן פשוט מאד, צופן אתבש המוכר ביהדות. בצופן זה כל אות מוחלפת באות אחרת, שהיא תמונת מראה שלה לפי האלף בית. האות אל"ף מוחלפת עם תי"ו, בי"ת עם שי"ן וכו'. כך לדוגמה מופיע בספר ירמיהו "מלך ששך". ממלכה בשם "ששך" אינה מוכרת ויש הטוענים שהשם "ששך" הוא למעשה צופן של ממלכת "בבל". באותו אופן "יושבי לב קמי" הם תושבי העיר "כשדים".

האלגוריתם, החלק שלא נשמר בסודיות, הוא שכל אות מוחלפת באות אחרת. מפתח ההצפנה הוא הסדר המדויק שבו מוחלפות האותיות. מי שיודעים שהיה חילוף אותיות אך לא איזו אות התחלפה עם איזו אות, אמורים להתקשות לפענח את המסר המקורי. זיכרו – צופן שנראה היום ילדותי, ייתכן שלא היה כזה לפני אלפי שנים, תקופה שבה אחוז קטן מאד מבני האדם בכלל ידע לקרוא.

כעת נברר מה ההבדל, אם כך, בין הצפנה, דחיסה וקידוד.

בתהליך דחיסה של מידע, לוקחים מידע ומפעילים עליו תהליך שגורם לכך שנדרש פחות מקום כדי לשמור אותו. לדוגמה, כאשר אנחנו שולחים קבצי תמונה באפליקציית ווטסאפ, מופעל אלגוריתם דחיסה שמוריד את איכות התמונה. נניח שהתמונה המקורית כוללת לכל פיקסל את שלושת ערכי ה-RGB – אדום, ירוק, כחול. לכל אחד משלושת הערכים נשמר מספר של 32 ביט שקובע את הערך המדויק של כמות האדום, הירוק והכחול. אם כך, כל פיקסל בתמונה המקורית יתפוס 4x3 = 12 בתים. אלגוריתם דחיסה פשוט עשוי להיות משהו כגון "קח פיקסל מהתמונה המקורית, בדוק לאיזה צבע מתוך 256 צבעים הוא הכי קרוב, וכתוב את מספר הצבע שמצאת". כך, כל פיקסל בתמונה המקורית, שלפני כן תפס 12 בתים, הופך להיות מספר בגודל בית אחד בלבד. גודל התמונה יהיה קטן בערך פי 12. חישבו: מדוע זו לא גם הצפנה?

זו אינה הצפנה משתי סיבות. הראשונה, שאין סוד משותף שאלגוריתם הדחיסה עושה בו שימוש. השניה, לא ניתן לשחזר את המידע המקורי. חלק מאיכות התמונה אבד לנצח. מסיבות אלו, דחיסה אינה הצפנה.

מה אם כן נוכל לבצע שחזור מלא של המידע המקורי? לדוגמה, אם ניקח את האות האנגלית "a" ונראה איך היא שמורה בזיכרון המחשב, נמצא שם את רצף הביטים "01100001". חישבו: האם זו הצפנה? ואם אתם חושבים שלא – מדוע?

התשובה היא שלפי קוד ascii, הערך שניתן לתו "a" הוא 61 (בבסיס הקסדצימלי), מה שמתבטא ברצף האחדות והאפסים, הייצוג הבינארי. אם כך, יש לנו אלגוריתם החלפה, שלוקח כל תו ומחליף אותו ברצף של אחדות ואפסים. מצד שני, אלגוריתם ההחלפה עדיין חסר סוד משותף. אם אני אכתוב את התו "a" ואתם תכתבו את התו "a", הביטים שיווצרו יהיו זהים. לכן אפשר לסכם כי ascii הינו קידוד, לא הצפנה.

נסכם את התהליך:

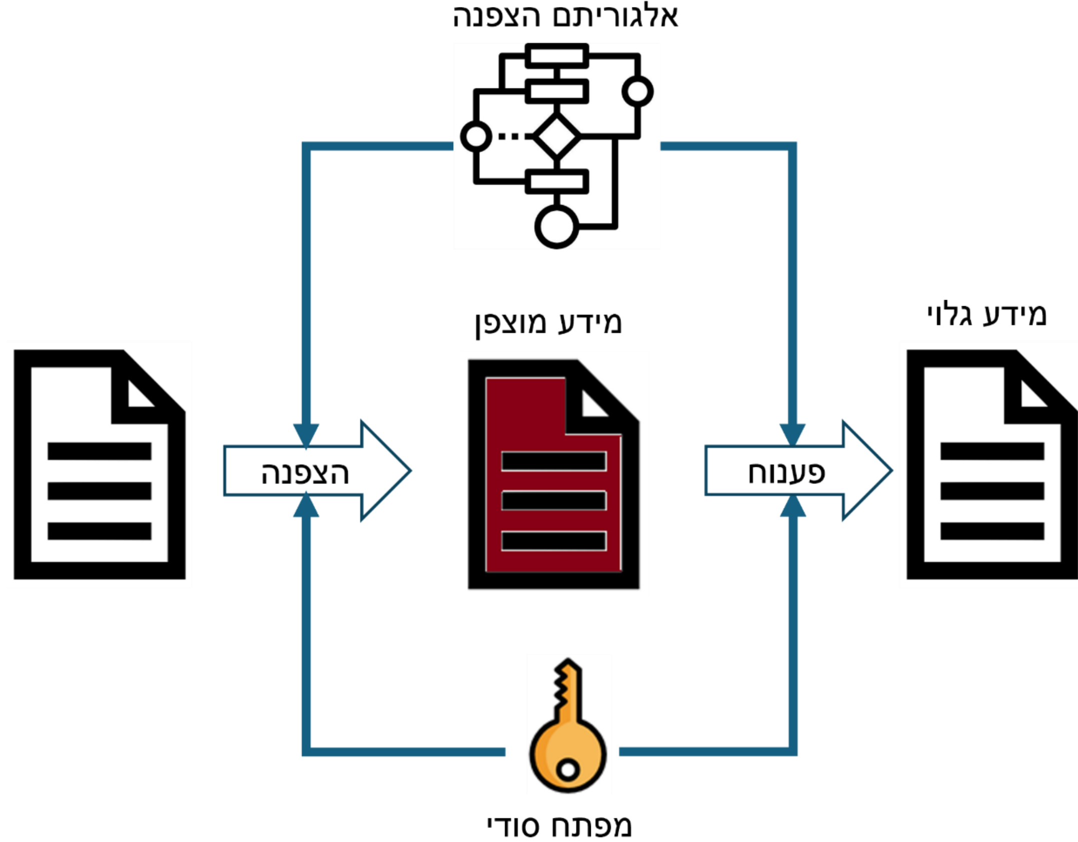
-מידע גלוי הוא מידע שניתן לקרוא אותו ישירות.

-הצפנה מבוססת על אלגוריתם הצפנה, שהינו גלוי וידוע, ועל מפתח, שהוא סוד משותף רק לצדדים לתקשורת.

-הפעלת האלגוריתם ומפתח ההצפנה על המידע הגלוי יוצרת מידע מוצפן.

-באמצעות אלגוריתם פענוח ומפתח פענוח על המידע המוצפן, ניתן לשחזר את המידע הגלוי.

מדוע שם הפרק שלנו הוא "הצפנה סימטרית", ולא פשוט "הצפנה"? בהצפנה סימטרית, אותו מפתח משמש גם להצפנה וגם לפענוח. נקרא לו פשוט "מפתח סודי". התהליך המומחש באיור הבא הוא הצפנה סימטרית מסוג נפוץ, שבה אלגוריתם ההצפנה משמש גם לפענוח:



האם ישנן סוגים אחרים של הצפנות? הצפנות בהן יש מפתח אחד להצפנה ומפתח אחר לפענוח? על כך נרחיב בפרק אחר, עד אז נתחיל בלכסות את הידע שהיה קיים מידי קדם עד 1970 בערך...

צופן הזזה

צופן הזזה הוא צופן פשוט שבו נכתוב את אותיות האל"ף-בי"ת במעגל, ונבחר כמפתח הצפנה מספר כלשהו N, שהוא ההזזה. כל אות בטקסט הגלוי תזוז N מקומות. לדוגמה, אם נבחר את N בתור 2, אז האות A תהפוך להיות C, האות B תהפוך להיות D, ואילו האותיות Y ו-Z יהפכו להיות האותיות A ו-B, בהתאמה.

כדי לשבור את ההצפנה, צריך למצוא את N. למרות ש-N יכול להיות תיאורטית מספר עצום, בפועל N יהיה מספר הקטן מכמות האותיות האל"ף-בי"ת, עקב עקרון המעגליות. כמו כן N לא יהיה 0, כיוון שבמקרה זה כל אות "תזוז" לעצמה והטקסט המוצפן יהיה זהה לטקסט הגלוי. אם כך, נותרנו עם אפשרויות כמספר האותיות באל"ף-בי"ת, פחות אחת. ננסה את כל האפשרויות עד שנקבל מסר בעל משמעות.

האם תוכלו לפענח את המסר הבא?

IRGVCTXIH GSQTYXIV RIXASVOW EVI GSSP

צופן קיסר

כמו בצופן הזזה, גם בצופן קיסר נחליף כל אות באות אחרת, אך בלי שחייב להתקיים שהמרחק בין האות הגלויה למוצפנת הוא קבוע. לדוגמה, A יכול להפוך ל-K ואילו B יכול להפוך ל-D. כמובן שלכל אות מוצפנת יש רק אות אחת גלויה שיכולה להפוך אליה. לא יכול להיות לדוגמה שגם A וגם L יהפכו להיות K. צופן הזזה הוא למעשה מקרה פרטי של צופן קיסר.

צופן קיסר נקרא כך על שם הקיסר הרומי יוליוס קיסר, שחי במאה הראשונה לפני הספירה. ההיסטוריון הרומי סווטוניוס (Suetonius) סיפר בכתביו שקיסר עשה שימוש בצופן זה.

הבעיה שמתמודדים איתה מי שרוצים להשתמש בצופן קיסר, היא שמורכב יחסית לתאם את ההחלפות. שני הצדדים לתקשורת צריכים להחליף טבלה של חילופי האותיות. סביר שהצדדים גם יצטרכו להחזיק כלשהו עותק של הטבלה, כיוון שקשה לזכור לכל אות באיזו אות היא מוחלפת. השמירה של עותק של הטבלה היא בעצמה חולשה בצופן. לכן נדרשת שיטה פשוטה ליצור צופן החלפה זכיר בקלות.

השיטה שנבחרה היא זו: ניקח מילה או ביטוי, נוריד את הרווחים ואת האותיות הכפולות, ונרשום את התוצאה מתחת לאותיות האלפבית. לדוגמה, אם המילה שבחרנו היא PUMPKIN, נצמצם אותה לאחר הורדת הכפילויות ל-PUMKIN. ראשית נכתוב את התוצאה מתחת לאותיות האל"ף-בי"ת, כך ש-P תיכתב מתחת ל-A, וכו'. המילה PUMKIN מסתיימת באות N, לכן נמשיך את האל"ף-בי"ת מהאות הזו, תוך כדי שאנחנו מדלגים על אותיות שכבר הופיעו במילת הצופן PUMKIN:

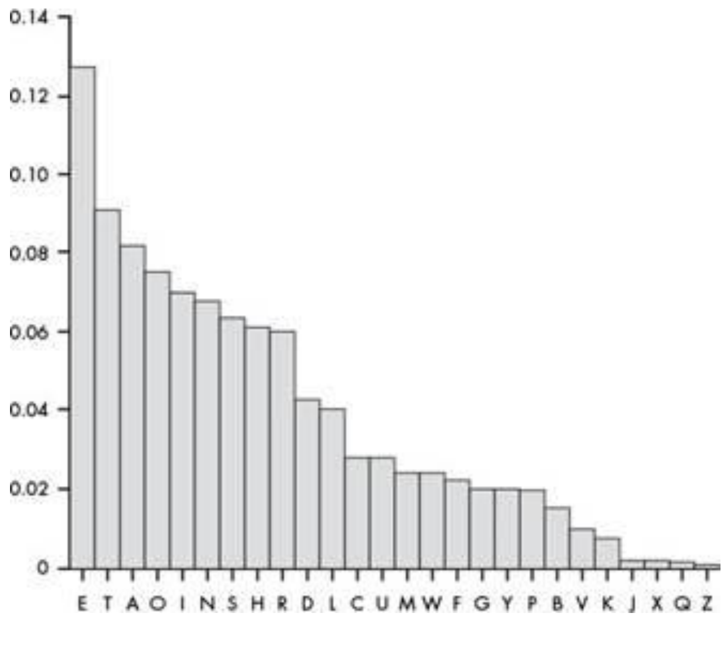
a b c d e f g h I j k l m n o p q r s t u v w x y z
P U M K I N O Q R S T V W X Y Z A B C D E F G H J L

שיטה זו היא בעלת יתרונות רבים, כיוון שמאפשרת הצפנה פשוטה באמצעות מילת קוד שקל לזכור. כמו כן, באופן תיאורטי, מספר האפשרויות השונות לצופן הוא כמעט אינסופי – האות A יכולה להפוך להיות 26 אותיות, האות B יכולה להפוך לאחת מ-25 אותיות וכך הלאה. תיאורטית, יש 26 עצרת אפשרויות, מה שהופך את הצופן לבלתי שביר.

ואכן הצופן היה בלתי שביר משך מאות שנים, עד ימי האימפריה הערבית. האימפריה הערבית התקיימה משך כ-600 שנה ובשיא כוחה שלטה בשטח עצום, מאיראן ועד ספרד. בזמן הזה התקיים תור הזהב של האיסלאם,

התקיימה פריחה מדעית ותרבותית, וגם היהודים נהנו מפריחה תרבותית. המלומדים הערבים החזיקו בשילוב ידע הכרחי לטובת פתרון הבעיה של צופן קיסר: ידיעת שפות וידיעת מתמטיקה. במאה התשיעית לספירה המלומדים שמו לב לתופעה מעניינת. אם ניקח טקסט כלשהו, אחוז הפעמים שכל אות מופיעה בתוכו הוא פחות או יותר קבוע וניתן לחישוב מראש, על סמך טקסטים אחרים באותה השפה. לדוגמה, בעברית האותיות א-ה-ו-י יופיעו באחוז גבוה יותר מאשר האותיות צ-ט-ז, וכדומה. השלב הבא הוא להכין ניתוח תדירויות – טבלה של ההסתברות של כל אות בשפה להופיע בטקסט. כאשר ניגש לפענח צופן החלפה כמו צופן קיסר, נספור את כמות האותיות מכל סוג ונתאים אותן לפי מה שצפוי בהתאם לניתוח התדירויות שהכנו. חשוב לציין שטקסט לא תמיד מתנהג באופן זהה לניתוח התדירויות, במיוחד אם הוא קצר. לכן נתחיל באותיות הנפוצות ומשם נתקדם, כשאנחנו מנסים כמה אותיות שהתדירות שלהן יצאה דומה לניתוח התדירויות התיאורטי שהכנו מראש.

לדוגמה, להלן ניתוח תדירויות בשפה האנגלית:



מקור: inventwithpython.com

אפשר לראות כי האות E מופיעה כ-12% מהפעמים, האות T כ-9%, וכו'.

להלן טקסט מוצפן, האם תצליחו לפענח אותו?

https:\\data.cyber.org.il\networks\sub_cipher_text

נסו לפענח בעצמכם, לפני שתקראו את הטיפ הבא, שכולל הסבר על שלבי הפענוח. כמו כן, נסו לא להשתמש בכלי AI לטובת פתרון הבעיה הזו או הבעיות הבאות :)

טיפ: כיתבו סקריפט פייתון שיבצע את הפעולות הבאות:

- יספור ויציג לכם את התדירויות של כל אות
- יקבל מכם השערה איזו אות התחלפה עם איזו אות, ויכניס את ההחלפה למילון (המפתח במילון יהיה האות המוצפנת, הערך יהיה האות הגלויה)
- יציג את הטקסט עם הפענוח לפי הערכים הקיימים במילון
- העזרו בידע שלכם באנגלית כדי לבדוק אפשרויות שונות לאותיות שהתדירויות שלהן קרובות, חפשו מילים נפוצות באנגלית (לדוגמה "The").

צופן ויז'נר

כפי שראינו, החולשה של צופן קיסר הוא ניתוח תדירויות. נעשו ניסיונות לפתח צפנים שבהם הוחלפו צמדי אותיות בצמדים אחרים, אבל גם ניסיונות אלו כשלו, כיוון שגם לצירופים של שתי אותיות יש ניתוח תדירויות. בסופו של דבר, הבעיה היסודית של ניתוח תדירויות סטטיסטי לא השתנתה והיה צריך רעיון יצירתי שימנע ניתוח תדירויות.

ויז'נר (Blaise de Vigenere) המציא את הצופן הקרוי על שמו ב-1586. הרעיון הוא פשוט – ניצור מצב שבו מאחורי אות מוצפנת לא עומדת תמיד אותה אות גלויה, אלא אחת מתוך מגוון אותיות. לדוגמה, האות המוצפנת K עשויה להיות מדי פעם האות הגלויה E ומדי פעם האות הגלויה V. התדירות של E היא בערך 12% ואילו התדירות של V היא בערך 1%. מי שיספור את התדירות של האות K בטקסט המוצפן יקבל ממוצע שאי אפשר להיעזר בו לניתוח תדירויות.

איך מייצרים את זה? באמצעות אוסף של צפני הזזה פשוטים. כדי להבין את השיטה, ניקח טקסט גלוי ונצפין אותו. לדוגמה lets encrypt this. כעת נבחר מילת צופן, נניח לדוגמה PHONE.

נכתוב את הטקסט הגלוי ומתחתיו את מילת הצופן באופן מחזורי:

l e t s e n c r y p t t h i s
PHONE PHONEPHONE

ניקח את "ריבוע ויז'נר" ונשתמש בו באופן הבא:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

כדי להצפין את האות הראשונה L, נלך לעמודה L ולשורה P. נקבל את האות המוצפנת A.

כדי להצפין את האות השנייה E, נלך לעמודה E ולשורה H. נקבל את האות המוצפנת L.

וכך הלאה. שימו לב שהאות הגלויה T מופיעה בטקסט הקצר שלנו שלוש פעמים:

- פעם היא תוצפן על-ידי האות O ותהפוך ל-H
- פעם תוצפן על-ידי P ותהפוך ל-I
- פעם תוצפן על-ידי H ותהפוך ל-A

אנחנו רואים כי התדירות הגבוהה יחסית של האות T מתפזרת ומתמצעת על מספר אותיות. והיופי – כל מה שהיינו צריכים לתאם בין שני הצדדים לתקשורת היא רק מילת קוד אחת, שקל לזכור.

משך מאות שנים צופן ויז'נר נחשב בלתי שביר וזכה לכינוי "Le chifre indechifre", בצרפתית "הצופן שאינו ניתן לשבירה". עד שבשנת 1853, צ'רלס באבאג' (Charles Babage) הצליח. באבאג' היה איש אשכולות בעל דמיון

מפותח. הוא נודע בכך שהמציא את האב-טיפוס למחשב מודרני, שהקדים את זמנו בכמאה שנה. באבאג' המציא מחשב מכני שנקרא "מכונת ההפרשים" ושהיה ניתן לתכנות.

באבאג' שם לב לחולשה של צופן ויז'נר – אם אנחנו יודעים מה אורך מילת הצופן, אנחנו מקבלים צופן הזזה פשוט שקל לפענח אותו. בדוגמה שלנו, אורך מילת הצופן PHONE הוא חמש. נעשה ניתוח תדירויות כאשר אנחנו לוקחים בחשבון רק כל אות חמישית. נכתוב את כל האותיות במקומות ה-1, 6, 11, 16 וכו', ונעשה להן ניתוח תדירויות. האות הכי נפוצה היא בוודאות גבוהה האות שבמקור היתה E. נשתמש בריבוע ויז'נר כדי לדעת באיזו שורה אנחנו נמצאים עבור כל האותיות שהמיקום שלהן מתחלק בחמש עם שארית אחד. כעת נכתוב את כל האותיות במקומות 2, 7, 12, 17 וכו'. גם כאן נמצא את האות עם התדירות הכי גבוהה, נלך לטור של E ונראה מה השורה המתאימה כדי להגיע לאות המוצפנת. וכך הלאה. בצורה זו נפענח בקלות את מילת הצופן.

הבעיה שנותרה לנו היא לדעת מה אורכה של מילת הצופן. גם את זה אפשר לפתור בשיטה מתימטית. ננחש אורך כלשהו, לדוגמה 4. אז ניקח כל אות רביעית (מקומות 1, 5, 9 וכו') ונספור את התפלגות התדירויות של האותיות שקיבלנו. אם ההתפלגות דומה לתיאוריה – לדוגמה, יש אות אחת נפוצה בערך ב-12% וארבע אותיות נדירות שמופיעות פחות מאחוז אחד, אז מצאנו את האורך.

נסו לפתור את הקוד הבא. שימו לב שהוא ארוך למדי כדי לאפשר ניתוח תדירויות בדילוגים.
https:\\data.cyber.org.il\networks\vigenere_cipher_text

טיפ: כדי להעריך האם ההתפלגות שקבלנו "טובה", הגדירו פונקציה שמחשבת את המרחק בינה לבין ההתפלגות התיאורטית.

להלן טבלה ממויינת של ההתפלגויות:

FREQ_TABLE = [12.02, 9.1, 8.12, 7.68, 7.31, 6.95, 6.28, 6.02, 5.92, 4.32, 3.98, 2.88, 2.71, 2.61, 2.3, 2.11, 2.09, 2.03, 1.82, 1.49, 1.11, 0.69, 0.17, 0.11, 0.1, 0.07]

המרחק של כל אות בנפרד יהיה התדירות שקיבלנו פחות התדירות התיאורטית, בריבוע (כדי שהמרחק יהיה תמיד חיובי). הפונקציה תחזיר ערך שהוא סכום המרחקים של כל האותיות מהתדירות התיאורטית שלהן. האורך הנכון הוא זה שהפונקציה של המרחקים נמצאת עליו במינימום.

לאחר שמצאתם את האורך הנכון, כל מה שצריך לעשות הוא לספור את התדירות של האות הנפוצה ביותר בדילוגים. סביר מאד שזו האות E. באמצעות ריבוע ויז'נר תוכלו לבדוק מה היתה ההזזה שיצרה אותה וכך תגלו אות מתוך מילת הצופן.

האניגמה

החולשה של צופן ויז'נר היא החזרתיות שלו, שנובעת מכך שמילת הצופן הקצרה למדי מועתקת שוב ושוב. אם מצאנו את האורך של מילת הצופן, סיימנו. אבל מה אם היתה לנו מילת צופן באורך שהוא למעשה אינסופי? את האתגר הזה מימשה מכונת האניגמה, ששימשה את הגרמנים במלחמת העולם השנייה.

העיקרון של מכונת האניגמה הוא צופן החלפה בין אות לאות. נדמיין גלגל שיניים מסתובב, כאשר על כל אחת מהשיניים שלו רשומה אות. נגיד שבמצב ההתחלתי האות הגלויה A תוחלף באות D. עם ההקשה על תו כלשהו, גלגל השיניים זז שן אחת. כעת אם נקיש על A היא תוחלף באות E. הקשה נוספת על A תוציא את האות F וכו'. כלומר, אם נכתוב את האות A עשרים ושש פעמים, יודפסו לנו אותיות ה-ABC בזו אחר זו, החל מהאות D ועד חזרה מעגלית אל האות C. יצרנו בעצם צופן הזזה עם מילת קוד באורך 26.

בואו נסבך את העניינים: נוסיף עוד גלגל שיניים. נניח שהקשנו על התו A, גלגל השיניים הראשון המיר אותה לאות D, גלגל השיניים השני המיר אותה לאות S. הקשנו 26 פעמים על האות A, גלגל השיניים הראשון סיים סיבוב מלא והנה החלק החשוב – גלגל השיניים השני זז תו אחד. בפעם הבאה שנקיש על A, גלגל השיניים הראשון שוב ימיר אותה לאות D, אבל גלגל השיניים השני ימיר אותה לאות T. הופ! יש לנו מילת קוד באורך 26 בריבוע.

הגרמנים השתמשו במערכת של בין 3 ל-5 גלגלי שיניים. ככל שהתקדמה המלחמה הם הוסיפו גלגלי שיניים, וצי הצוללות הגרמני השתמש תמיד בגרסה עם יותר גלגלי שיניים. אם כך, ניתן לומר שהאלגוריתם של האניגמה מבוסס על אוסף של גלגלי שיניים המחוברים זה לזה. כפי שאתם ודאי זוכרים, הצפנה דורשת גם אלגוריתם וגם מפתח סודי. מהו המפתח הסודי הידוע רק לשני הצדדים לתקשורת האניגמה?

המפתח הסודי הוא בחירת הגלגלים והמיקום ההתחלתי שלהם. לכל גלגל יש 26 אותיות שמהן הוא יכול להתחיל להסתובב. לדוגמה, באניגמה פשוטה של שלושה גלגלים יש 26 בחזקת 3 אפשרויות סידור התחלתיות לגלגלים, או 17,576. מעבר לכך, הגלגלים שונים זה מזה בסידור האותיות, כך שיש משמעות לסדר הגלגלים במערכת. יש שש דרכים לסדר שלושה גלגלים, לכן מספר האפשרויות ההתחלתי באניגמה הפשוטה שלנו הוא 105,456.

המפתח הסודי אם כך עשוי להראות כך: JAK 132. כלומר סדר את הגלגלים לפי הסדר 1-3-2, את גלגל 1 שים על האות J, גלגל 2 על האות K, גלגל 3 על האות A.

למען הדיוק ההיסטורי חשוב לציין שהיו מנגנונים נוספים שהגדילו את כמות האפשרויות למצב התחלתי:

- אחד-עשר כבלים שהחליפו בין צמדי אותיות
- כמות גלגלים גדולה יותר לבחירה מתוכם, לדוגמה בחירה של 3 גלגלים מתוך 5
- כמות גלגלים גדולה יותר בשימוש, לדוגמה צוללת גרמנית השתמשה ב-5 גלגלים מתוך 8

כאשר מבצעים את המכפלות האמיתיות, מגיעים למספרים עצומים בהרבה מהדוגמה הפשוטה שהסתכמה בכמאה אלף.

כיצד שני הצדדים לתקשורת היו יכולים לתאם ביניהם את מפתח ההצפנה? הגרמנים הדפיסו ספרי מפתחות שהציגו לכל יום את המפתח היומי. בכל יום התחלף הקוד ופעם בזמן מה הוחלפו ספרי המפתחות. כאן טמונה חולשה משמעותית של האניגמה. אם ספר המפתחות נפל בידי בעלות הברית הן יכלו לפענח את כל תשדורות הגרמנים עד החלפת ספר המפתחות. אפשר לקחת הפסקה מהספר ולצפות בסרט המעולה U-571.

בעיית החלפת המפתחות בין צדדים למערכת צופן סימטרית אינה ייחודית לאניגמה. עוד נגיע בהמשך לבעיה זו ולפתרון שנמצא, בינתיים, לבעיה זו.

פיצוח האניגמה התבסס על מה שנקרא "עריסות". כלומר, ננחש טקסט כלשהו שאמור להופיע בהודעה ונריץ את כל האפשרויות לסידורים התחלתיים עד שהטקסט הזה יופיע בטקסט המפוענח. הצוללות הגרמניות היו שולחות מדי בוקר עדכון מזג אויר שכלל את המילה הגרמנית wetter, מזג אוויר. זו היתה עריסה מוצלחת לחיפוש. כדי להצליח לבצע את סריקת האפשרויות בצורה יעילה, נדרש ציוד שיודע לקבל קלט, לבצע חישובים בצורה מהירה ולעצור אם תנאי עצירה מתקיים (הופעת העריסה wetter בטקסט המפוענח). ציוד זה הוא המחשב, שהומצא על-ידי המתמטיקאי אלן טיורינג בזמן שעבד על פענוח האניגמה הגרמנית.

ההיסטוריונים חלוקים בנוגע לתרומתו של טיורינג לניצחון בעלות הברית על הנאצים. יש מי שאומרים שללא טיורינג, בעלות הברית היו מפסידות לנאצים. מי שחולקים עליהם אומרים שככל הנראה בעלות הברית היו מנצחות, אבל טיורינג קיצר את המלחמה בשנתיים. כך או כך, ייתכן שטיורינג הציל יותר יהודים מכל אדם אחר בהיסטוריה.

הצפנות בלוק – Block Ciphers

פיתוח המחשב היה כלי שובר שוויון בעולם ההצפנה. 90 שנה אחרי המצאת האניגמה, לפטופ רגיל יכול לשבור צופן אניגמה בתוך דקות. נדרש למצוא פתרון שמחשב לא יוכל להתגבר עליו, או לפחות לא יוכל להתגבר עליו בזמן סביר. השלב הבא בהתפתחות הצפנים היה ליצור צופן שיש לו אפשרויות כה רבות, עד שלא מעשי למחשב למצוא את

מפתח ההצפנה באמצעות ניסוי וטעייה. הפתרון שנבחר על-ידי מכון התקנים האמריקאי – הצפנות בלוק. הצפנות אלו הן סוג ספציפי של הצפנות סימטריות.

הצפנות בלוק התפתחו עם השנים, וככל שנמצא שהן שבירות למתקפות – האלגוריתמים שלהן נהיו מורכבים יותר. בשנות ה-70 פותח אלגוריתם בשם DES, קיצור של Data Encryption Standard. אלגוריתם ה-DES שביר היום, כמו גם שכלול שלו – Triple-DES. בראשית שנות ה-2000 אומץ אלגוריתם ה-AES, קיצור של Advanced Encryption Standard. אלגוריתם זה נפוץ בשימוש גם כיום.

הרעיון של הצפנת בלוק הוא כזה: ניקח מידע בינארי (אם המידע שלנו אינו בינארי, נהפוך אותו לכזה) ונחלק את המידע הבינארי לביטים בקבוצות בגודל קבוע – בלוקים. לדוגמה כל 128 ביט יהיו בלוק. על כל בלוק נבצע פעולות כגון XOR, החלפה והזזה. תוך כדי ביצוע הפעולות נשתמש גם בביטים של המידע וגם במפתח סודי כלשהו.

ה"קסם" של הצפנת בלוק הוא שקל להגדיל את כמות האפשרויות. אם באניגמה היה צורך לטרוח ולהוסיף גלגל שיניים פיזי, בצופן בלוק פשוט נבחר כמות גדולה יותר של ביטים בשביל המפתח הסודי. בחירה של 128 ביט, לדוגמה, תיתן לנו שתיים בחזקת 128, בערך 3 עם שלושים ושמונה אפסים אחריו. לא מספיק טוב? קחו 256 ביט...

הנה תיאור כללי של אופן פעולת צופן AES מבוסס מפתח הצפנה בן 128 ביט. למעוניינים בהסבר מקיף יותר, להלן סרטון מומלץ: AES Explained (Advanced Encryption Standard) -Computerphile

ניקח 16 בתים, 128 ביט, של מידע גלוי ונכתוב אותם בצורת בלוק של 4x4 בתים. בכל בית יש ערך של 8 ביט, שיכול לקבל כל ערך בין 0 ל-255.

על השלבים הבאים נחזור מספר פעמים:

נעשה פעולת XOR עם מפתח ההצפנה שלנו. בנקודה זו חשוב לציין שפעולת ה-XOR היא פעולה סימטרית. אם תקחו מספר בינארי כלשהו, תעשו לו XOR עם מפתח הצפנה ואז את התוצאה תעבירו XOR עם אותו מפתח – תקבלו את המספר הבינארי המקורי.

נחליף כל ערך בערך אחר, לפי טבלת המרה שיש לנו בזיכרון. למעשה, זהו צופן החלפה – אך בין מספרים. לדוגמה, היכן שהערך הוא 55 נחליף ב-71, את 56 נחליף ב-125 וכו'. אין שני מספרים שמוחלפים להיות אותו מספר.

נערבב את השורות. נחליף, לדוגמה, את שורה 1 בשורה 3.

נערבב את העמודות.

נחזור על השלבים הללו מספר פעמים כרצוננו. שימו לב שהפעולות שביצענו הן פעולות של XOR והחלפה, פעולות שמאוד קל למעבד לבצע. מי מכם שלמד אסמבלי יכול להעריך כי ביצוע של כל פעולה כזו עשוי לקחת מספר קטן של פעולות בשפת מכונה.

קיבלנו באופן די פשוט ערבוב טוב של הביטים. הכוונה בערבוב "טוב" הוא שהתוצאה הסופית שקיבלנו היא שבתוך כל בית מוצפן יש קשר למספר בתים של מידע מקורי, ושהבתים של המידע המקורי אינם קרובים אחד לשני. זהו עיקרון מועיל, כי אם היתה לנו איזושהי השערה לגבי "עריסה", או מידע שאמור להיות במקור, הוא מעורבב היטב במידע המוצפן וקשה להשתמש בו.

וכמובן, חוזקו של הצופן נובע מהכמות הרבה של האפשרויות שנצטרך להשתמש בהן כדי לשבור את ההצפנה אם נרצה לעבוד בשיטת Brute-Force, קרי לעבור אחת-אחת על כל האפשרויות.

CTR ,CBC ,ECB

ובכל זאת, לצפני בלוק, כפי שתארנו אותם עד כה, עלולה להיות בעיה משמעותית: עבור אותו מידע, התוצר של ההצפנה זהה. ניקח לדוגמה 256 ביטים של אפסים. נחלק אותם לשני בלוקים של 128 ביט, ונעביר כל אחד מהבלוקים הצפנת בלוק בנפרד. התוצאה תהיה זהה. הרי שום דבר לא משתנה – האלגוריתם נותר זהה, מפתח הצופן נותר זהה, אז אם גם הבלוקים שמכילים את המידע הגלוי זהים – אין שום הבדל.

צופן ה-ECB, קיצור של Electronic Code Book, הוא דוגמה לצופן מסוג בלוק שמתקיימת בו בעיה זו.

כדי להמחיש את הבעיה של צופן ECB, ניקח תמונה, שיש בה אזורים דומים זה לזה. חלקם לבנים לגמרי, חלקם שחורים לגמרי. במקרה זה, תמונה שווה יותר מאלף הסברים, הנה כך נראית התמונה הגלויה, ואתה תמונה לאחר שהיא מחולקת לבלוקים וכל בלוק מוצפן. כפי שרואים, גם לאחר הצפנה, אפשר לזהות את התמונה:

מקור: ויקיפדיה, block cipher

אם כך, כדי למנוע את תופעת החזרתיות של המידע המוצפן, נצטרך להכניס גורם כלשהו שייצור שוני. כיוון שהאלגוריתם קבוע והמידע קבוע, המשחק שלנו יכול להיות עם מפתח ההצפנה. נסקור שתי שיטות שמשנות את מפתח ההצפנה.

CBC, קיצור של Code Block Chaining, אומרת כך: לאחר שסיימנו להצפין בלוק, ניקח חלק כלשהו מהתוצאה, לדוגמה 8 הביטים האחרונים, ונשתמש בהם כחלק ממפתח ההצפנה של הבלוק הבא. התוצאה היא שכל בלוק יוצפן באופן שונה, גם אם המידע הגלוי זהה.

החסרון של CBC הוא שאי אפשר לעבוד במקביל. לא בהצפנה ולא בפענוח. עד שלא סיימנו הצפנה של הבלוק הראשון לא נוכל להתחיל להצפין את הבלוק השני, וכן הלאה. עד שלא סיימנו לפענח את הבלוק הראשון, לא נוכל להתחיל לפענח את הבלוק השני, וכן הלאה.

CTR, קיצור של Counter, מציע פתרון אחר לבעיה: כל בלוק יוצפן על-ידי המפתח המקורי, בתוספת מונה. הבלוק הראשון יוצפן על-ידי המפתח המקורי פלוס 0, הבלוק השני על-ידי המפתח פלוס 1 וכן הלאה. כך, מצד אחד מקבלים שוני בין תוצרי ההצפנה, ומצד שני ניתן לשמור על הצפנה ופענוח מקבילים. אפשר לעבוד עם בלוק כלשהו גם אם הקודמים לו טרם סיימו הצפנה או פענוח.

ל-CTR יש וריאציה ספציפית, הנקראת GCM, קיצור של Galois/Counter Mode. זוהי וריאציה פופולרית לשימוש יחד עם AES. הבעיה ש-GCM רוצה לפתור היא שתוקף עלול לקחת את המסר המוצפן, וגם מבלי לדעת לפענח אותו פשוט להפוך בו כמה ביטים ולקוות שהתוצאה תהיה בעלת משמעות לצד הקולט. כלומר, היינו רוצים יחד עם ההצפנה גם דרך לדעת שהמסר שלנו לא שונה בדרך. GCM מספק יכולת לבדוק בצד הקולט שהמסר הגיע ללא שינוי. למעוניינים להרחיב, מומלץ לצפות בסרטון הבא:

AES GCM (Advanced Encryption Standard in Galois Counter Mode) -Computerphile

ישנם צפני בלוק שמשתמשים בשיטות נוספות, כגון CFB (Cipher Feedback), ו-OFB (Output Feedback). התמקדנו ב-CBC ו-CTR כיוון שסוקטים מוצפנים משתמשים בהן.

תוצר לוואי חשוב של CBC ו-CTR הוא מענה לבעיית ה-Anti Reply שתוארה בפרק המבוא לסוקטים מוצפנים. נזכיר, כי הבעיה נובעת מכך שגורם כלשהו יכול לקחת פקטות אמיתיות שיצרנו, מוצפנות יפה, לשכפל אותן ולהעביר אותן הלאה. אם יצרתם תקשורת מוצפנת מול הבנק שלכם ונתתם פקודת העברה של 1000 ש"ח לזכותי, אני יכול תאורטית להקליט את פקודת ההעברה, לשכפל אותה ולשדר אותה לבנק שוב ושוב. כיצד הבנק יכול לדעת שמדובר בבקשה משוכפלת? בשיטות CBC ו-CTR, מפתח הפענוח של כל פקטה הוא שונה. אך כאשר שכפלתי את הפקטה

ששלחתם, לא שיניתי את מפתח ההצפנה, אלא העברתי את הביטים כפי שהיו במקור. בין שנעשה שימוש בביטים האחרונים בפקטה הקודמת ובין שנעשה שימוש במונה, מפתח ההצפנה השתנה. הבנק ינסה לפענח את הפקטות המשוכפלות באמצעות מפתחות לא נכונים, ומתקפת השכפול תכשל.

למעוניינים להרחיב את ההבנה על AES CBC, מומלץ המאמר הבא על מתקפת ה-BEAST: [תיאור מתקפת ה-BEAST על פרוטוקול SSL]

16.1 תרגיל – צ'אט מוצפן בצופן בלוק

בתרגיל זה נשתמש בהצפנת בלוק פשוטה, המדמה את העקרונות של AES אך הרבה יותר פשוטה למימוש. ההצפנה שלנו תהיה יעילה מול חובבנים שיסניפו את התעבורה בין השרת והלקוח, אבל לא בפני מפצחי צפנים. בעולם האמיתי תמיד כדאי להשתמש בצפנים מוכרים, שעברו בדיקות וניסיונות שבירה.
מפתח ההצפנה שלנו יהיה בגודל 16 ביטים בלבד. לא מורכב לפיצוח באמצעות בדיקת כל האפשרויות, אך ממחיש את העיקרון ולא קשה למימוש. בשלב זה בחרו מפתח הצפנה כרצונכם.

בצד המשדר, כל 4 בתים שעומדים להישלח בסוקט יוצפנו בתור בלוק. אם כמות הבתים שצריכה להישלח אינה מתחלקת ב-4, הוסיפו ריפוד באפסים לגודל של 4 בתים.

א. בשלב הראשון יבוצע XOR של כל שני בתים עם מפתח ההצפנה.

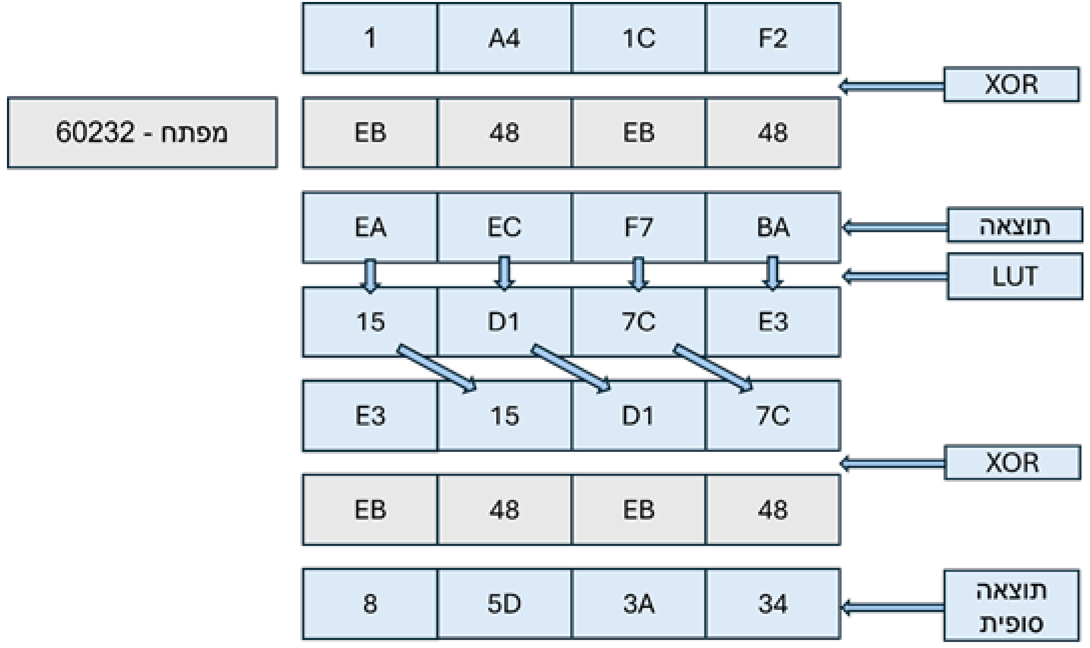
ב. בשלב השני, באמצעות מילון, יוחלף כל ערך בגודל בית, בערך אחר שגודלו בית. לדוגמה בית שערכו 0 יהפוך להיות 14, בית שערכו 1 יהפוך להיות 251, וכן הלאה. מילון שכל ערך בו מומר לערך אחר נקרא גם Look Up Table או בקיצור LUT.

ג. בשלב השלישי, יוחלף סדר הבתים בבלוק של הרביעייה, בצורה מעגלית. כך הבית ה-0 יהפוך להיות ה-1, ה-1 יהפוך להיות ה-2, ה-3 יחזור להיות ה-0.

ד. יבוצע XOR נוסף של כל שני בתים עם מפתח ההצפנה וזה הבלוק שישודר.

בצד הקולט: כל ארבעה בתים שמתקבלים מהסוקט יעברו פענוח לאחר הקליטה. הפענוח הוא הפעולה ההפוכה להצפנה: ביצוע XOR עם המפתח, החלפת סדר הבתים בצורה מעגלית (לכיוון השני), ביצוע Look Up Table (עם מילון שהערכים והמפתחות בו הפוכים יחסית למילון ששימש להצפנה, לדוגמה 251 הופך להיות 1), ביצוע XOR נוסף.

האיור הבא ממחיש את שלבי העבודה:



בשלב הראשון, נלקחים ארבעה בתים של מידע. נבחר מפתח – לדוגמה 60232 (ערך שנכנס ב-16 ביט). ערך ההקס' של המפתח הוא EB48.

בשלב השני, כל שני בתים יעברו XOR עם המפתח.

בשלב השלישי, כל אחד מהבתים יעבור המרה דרך LUT לערך אחר.

בשלב הרביעי, כל רביעיית הבתים "מסובבת" בית אחד.

בשלב החמישי והאחרון, כל בית עובר שם פעם XOR עם המפתח.

שלב הפענוח הוא פשוט חזרה על השלבים הללו בסדר הפוך, תוך שימוש ב-LUT ההופכי (מילון שבו ה-keys וה-values הפוכים ביחס למילון ששימש להצפנה).

וכעת, לאחר שמימשתם את ההצפנה, כיתבו צ'אט מוצפן: כל צד ישלח לצד השני הודעות מוצפנות, והצד הקולט יציג אותן למשתמש כאשר הן מפוענחות.

סיכום

בפרק זה צללנו לעולם הצפנים הסימטריים, החל מימי קדם ועד ימינו אנו ממש. למדנו מה היו החולשות השונות בכל צופן, כיצד הן שימשו את מפצחי הצפנים לשבירת ההצפנה, וכיצד דור חדש של צפנים התגבר – במידה כזו או אחרת – על החולשה.

צופן ההזזה ניתן לפענוח Brute Force על-ידי ניסוי וטעייה, כיוון שיש לו מספר אפשרויות מוגבל.

צופן קיסר הגדיל את מספר האפשרויות להיות עצרת של מספר האותיות באל"ף-בי"ת, אבל היה שביר עם מתקפת תדירות אותיות.

צופן ויז'נר פירק את הבסיס של מתקפת תדירות האותיות, אבל היה שביר עקב האורך המוגבל של מפתח ההצפנה, מה שגרם לחזרתיות הצופן.

צופן האניגמה פירק את הבסיס של מתקפת החזרתיות, בכך שאם המכונה נמצאת במצב מסויים, הפעם הבאה שהיא תגיע לאותו המצב עשוי להיות בעוד מאות אלפי או מליוני אותיות. עם זאת, הצופן היה שביר לטכנולוגיה חדשה ששינתה את העולם – המחשב.

צפני בלוק מנטרלים את הצד שמשתמש במחשב, בכך שהם פשוט הופכים את עבודת החישוב לאינסופית בזמן. כל ביט שמוסיפים לבלוק מכפיל את הזמן הפענוח פי שניים.

עם זאת, לכל הצפנים הסימטריים יש בעיה בולטת, שאם קראתם את הפרק בעיון גיליתם אותה. על כך בפרק הבא.

רמז לבעיה של צפנים סימטריים. מקור: הסרט U-571

פרק 17: החלפת מפתחות סימטריים

מבוא

סיימנו את הפרק הקודם כאשר אנחנו מבינים ששיטת ההצפנה הנוכחית מבטיחה כיום עמידות טובה בפני מפצחי צפנים המצוידים במחשבים חזקים. עם זאת, התנאי הבסיסי להצפנות סימטריות נותר קבוע משחר ההיסטוריה – שני הצדדים צריכים לתאם ביניהם מפתח הצפנה.

תיאום של מפתח הצפנה הוא משימה מורכבת מאוד כשמדובר בצוללת שצריכה ליצור קשר עם המפקדה. ספר של מפתחות ההצפנה צריך להיות מועבר לצוללת, והספר עלול ליפול בידי הצד שרוצה להאזין לתשדורות, ושיהיה מוכן להשקיע מאמצים אדירים כדי להשיג את ספר הצפנים.

תיאום של מפתח הצפנה הופך להיות משימה בלתי אפשרית כשמדובר במרחב האינטרנט. נסו לדמיין שהבנק שלכם, כדי שתוכלו ליצור איתו קשר מוצפן באינטרנט, היה מדפיס לכם ספר עם סיסמאות (ספר שהיה שונה כמובן מכל ספר של לקוח אחר). דמיינו שלא רק הבנק היה צריך לעשות זאת אלא גם כל רשת חברתית שהייתם חברים בה, כל אתר קניות, כל שירות ממשלתי. סביר שמרבית זמן הגלישה שלכם באינטרנט היה מוקדש לדפדוף בספרים ולהזנת סיסמאות...

אנחנו מבינים שכדי לקיים תקשורת מוצפנת, נדרש להכניס למערכת ההצפנה שלנו גורם נוסף, שבלעדיו קיום של תקשורת מוצפנת הוא פשוט לא מעשי – מנגנון החלפת מפתחות סימטריים.

Diffie Hellman

פקטת האינטרנט הראשונה הועברה ב-29 באוקטובר 1969. באמצע שנות השבעים, זמן רב לפני שהאינטרנט הפך נפוץ או שמישהו חשב על כך שיבוא יום ואנשים ישלחו מעליו תעבורה מוצפנת, שני חוקרים חזו את הנולד. ויטפילד דיפי ומרטין הלמן הבינו כי הקיום של האינטרנט תלוי במציאת שיטה להחלפת מפתחות הצפנה סימטריים.

נפרט מהי הבעיה שדיפי והלמן ניסו לפתור.

יש שני צדדים, נקרא להם אליס ובוב. אליס ובוב מרוחקים זה מזה והם לא נפגשו פיזית בעבר, כך שנמנע מהם לתאם סודות ולהעביר ספרי מפתחות.

אליס ובוב מקושרים זה לזה באמצעות ערוץ שאינו פרטי שלהם. כלומר, הנחת העבודה היא שהמידע שעובר בערוץ חשוף גם לעיניים זרות, נקרא לה איב. באנגלית השם איב מסתדר טוב יותר, כיוון שאיב נשמעת כמו קיצור של המילה האנגלית Eavesdropping. מקור מילה זו הוא השטח הצמוד לבית, שבזמן שיורד גשם, הטיפות מטפטפות עליו מן הקצה של גג הבית (Eaves). אנשים שעמדו באזור הזה, שקרוב מאוד לבית, וניסו לשמוע מה קורה בתוכו – נקראו Eavesdroppers. בעברית הפעולה נקראת האזנת סתר, או בקיצור ציתות, והמקור שלה הוא מהמילה הארמית צות – שמיעה. קצת פחות ציורי מהגרסה האנגלית.

אליס ובוב מעוניינים לתאם ביניהם מפתח סודי להצפנה הסימטרית, מפתח שרק שניהם יכירו. אולם כיוון שאיב יכולה להאזין גם לשלב שבו הם מתאמים את המפתח, זה חסר טעם. איב לא תצטרך לשבור את הצופן, הצופן יהיה בידיה.

כיצד, אם כן, יוכלו אליס ובוב לתאם מפתח הצפנה?

הפתרון של דיפי והלמן היה להתבסס על פונקציה מתמטית שקשה מאוד להפוך אותה, כלומר קשה מאוד לפתור אותה שלא בניסוי וטעייה. ניקח לדוגמה את הפונקציה $y = 2x + 3$. אם, לדוגמה, נאמר לכם שערכו של y הוא 7, יהיה קל מאוד לחלץ את x.

מצד שני, ניקח את הפונקציה שדיפי הלמן בחרו – g בחזקת x מודולו p:

$g^x \ mod \ p$

נניח שהייתם יודעים, כי 5 בחזקת x מודולו 23 הם 9. האם תוכלו לחשב את x? סביר שמה שתאלצו לבצע הוא ניסוי וטעייה. בדיוק מה שהיינו רוצים שאיב תצטרך לעשות, כיוון שאם נגדיל מספיק את המספרים, אפשר יהיה להניח שגם אם איב מצויידת במחשב משוכלל היא לא תספיק לפתור את הבעיה בזמן סביר.

לאחר שהבנו מדוע הרעיון המתימטי הזה מקשה על איב, נראה כיצד אליס ובוב משתמשים בו בצורה מוצלחת.

ראשית, אליס ובוב יחליפו ביניהם שני מספרים, הקרויים p ו-g. המספר p יהיה מספר ראשוני, והמספר g יהיה primitive root modulo p. נקדיש דקה להבין את המושג הזה באמצעות שתי דוגמאות.

נניח ש-p הוא 5.

נבדוק מה קורה כאשר לוקחים את המספר 2, מעלים אותו כל פעם בחזקה אחרת ומבצעים לתוצאה מודולו עם 5.

$2^1 \ mod \ 5 = 2$

$2^2 \ mod \ 5 = 4$

$2^3 \ mod \ 5 = 3$

$2^4 \ mod \ 5 = 1$

תוצאה: עבור העלאה של 2 בחזקות מ-1 ועד 4 (p מינוס 1), קבלנו את כל המספרים מ-1 ועד 4 (p מינוס 1).

נקרא ל-2 "primitive root" של 5.

ניקח את המספר 4. האם הוא primitive root של 5?

$4^1 \ mod \ 5 = 4$

$4^2 \ mod \ 5 = 1$

$4^3 \ mod \ 5 = 4$

$4^4 \ mod \ 5 = 1$

לא קיבלנו את כל המספרים. לכן 4 אינו primitive root של 5.

מהתוצאה הזו אפשר לקבל הבנה למה g צריך להיות primitive root של p.

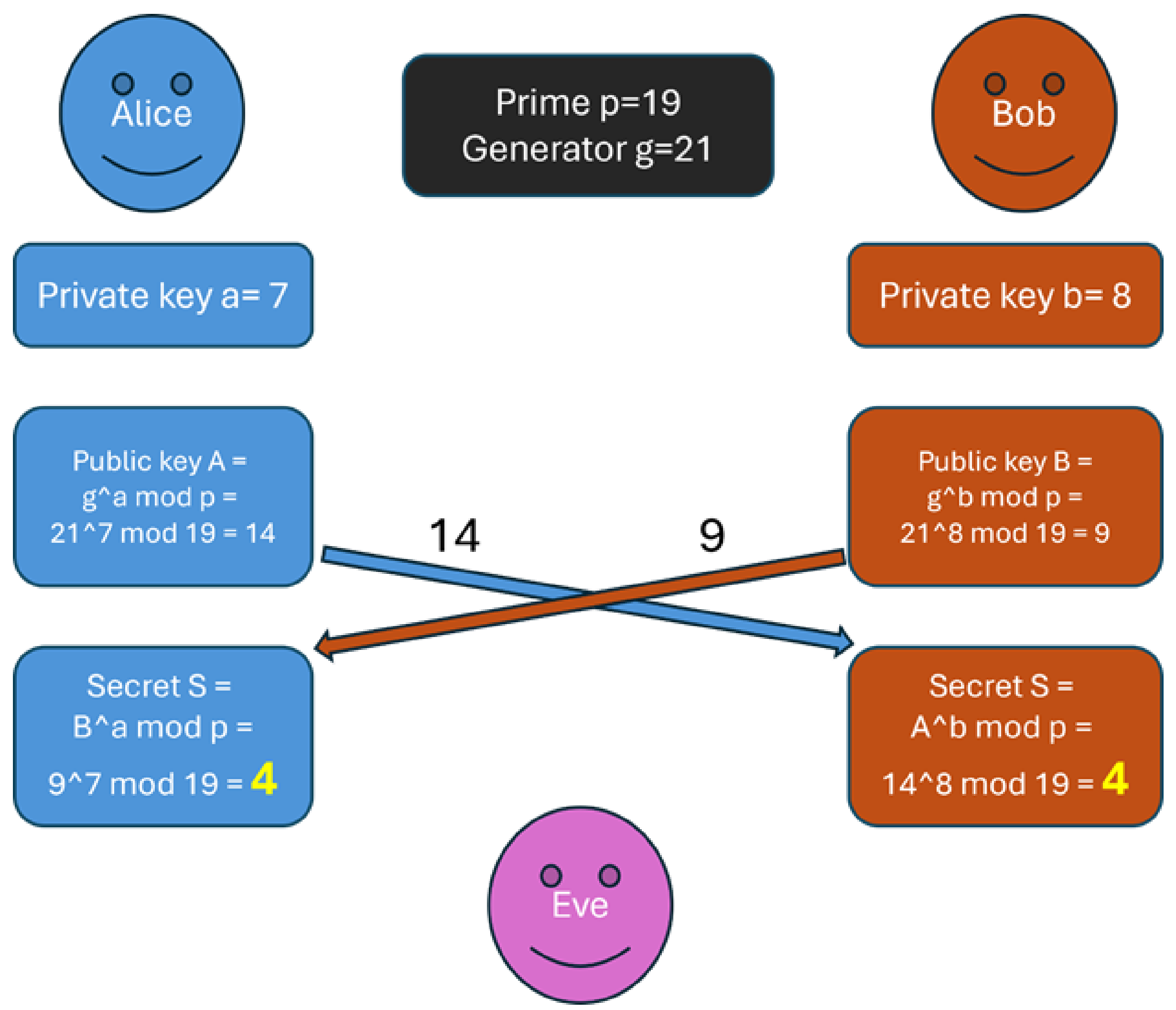
נניח שהיו מוסרים לכם שהעלו את 2 בחזקת x, ביצעו לתוצאה מודולו 5 וקיבלו 1. מהו x? הייתם נדרשים בניסוי וטעייה לעבור על ארבע אפשרויות. לעומת זאת, אילו היו משנים את השאלה כך שהעלו את 4 בחזקת x, ביצעו לתוצאה מודולו 5 וקיבלתי 1, הייתם צריכים לעבור על מחצית האפשרויות כדי למצוא את x.

השימוש ב-primitive root מגדיל את כמות האפשרויות שאיב צריכה לבדוק. אם g לא היה תלוי בצורה כזו ב-p, אפשר היה למצוא מהר יחסית באמצעות ניסוי וטעייה את הפתרון לבעיה של דיפי הלמן.

אם כך, אליס ובוב בחרו p ו-g. ומה הם עושים איתם? מודיעים זה לזה, וכן – גם איב יכולה להאזין ולדעת אותם. אנחנו כבר מבינים שזה לא הולך לעזור לה הרבה. בשלב זה חשוב לציין, שלמרות שבדוגמה השתמשנו במספרים קטנים למדי, אליס ובוב יבחרו במציאות מספרים גדולים מאד. לדוגמה, המספר p יהיה מספר ראשוני שנדרשים 2048 ביטים, או 617 ספרות דצימליות כדי לכתוב אותו:

FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1 29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245 E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE65381 FFFFFFFF FFFFFFFF

ואילו המספר g יהיה פשוט – 2.



לצורך הפשטות נניח כי p הוא 19 וכי g הוא 21. תוכלו לבדוק ולהווכח כי 21 הוא אכן primitive root של 19. בשלב הזה אליס בוחרת מספר סודי כלשהו, נקרא לו a, הקטן מ-p. נניח a הינו 7. אליס מבצעת את החישוב הבא:

$$21^7 \; mod \; 19$$

את התשובה, 14, שנסמן באות A, אליס מעבירה לבוב. גם איב יכולה לרשום אותה.

במקביל, גם בוב בוחר מספר סודי b, נניח 8. בוב מבצע את החישוב הבא:

$$21^8 \; mod \; 19$$

ושולח את התשובה, 9, לאליס. נסמן את התשובה באות B.

כעת מגיע השלב ה"קסום".

אליס לוקחת את מה ששלח בוב, 9, ומכניסה אותו לפונקציה של דיפי הלמן:

$9^7 \, mod \, 19$

התוצאה שאליס קיבלה היא 4.

בוב לוקח את מה שאליס שלחה, 14, ומכניס אותו לפונקציה של דיפי הלמן:

$14^8 \, mod \, 19$

התוצאה גם היא 4 😊

הסבר קצר על הקסם המתימטי. החישוב שאליס ביצעה בשלב האחרון הוא

$B^a \, mod \, p$

נציב בחישוב את B:

$\left(g^b \, mod \, p\right)^a \, mod \, p$

הביצוע של פעולת מודולו פעמיים ניתן לצמצום לפעם יחידה:

$\left(g^b\right)^a \, mod \, p$

ולפי חוקי החזקות נקבל:

$(g)^{ba} \, mod \, p$

באותו אופן אם נבדוק את החישוב של בוב, הוא יבצע:

$(g)^{ba} \, mod \, p$

שזהה לחישוב של אליס. כלומר, בעצם החישוב של דיפי הלמן הוא כזה: ניקח את g, נעלה אותו בחזקת המכפלה של המספרים שבחרו אליס ובוב, נבצע מודולו p.

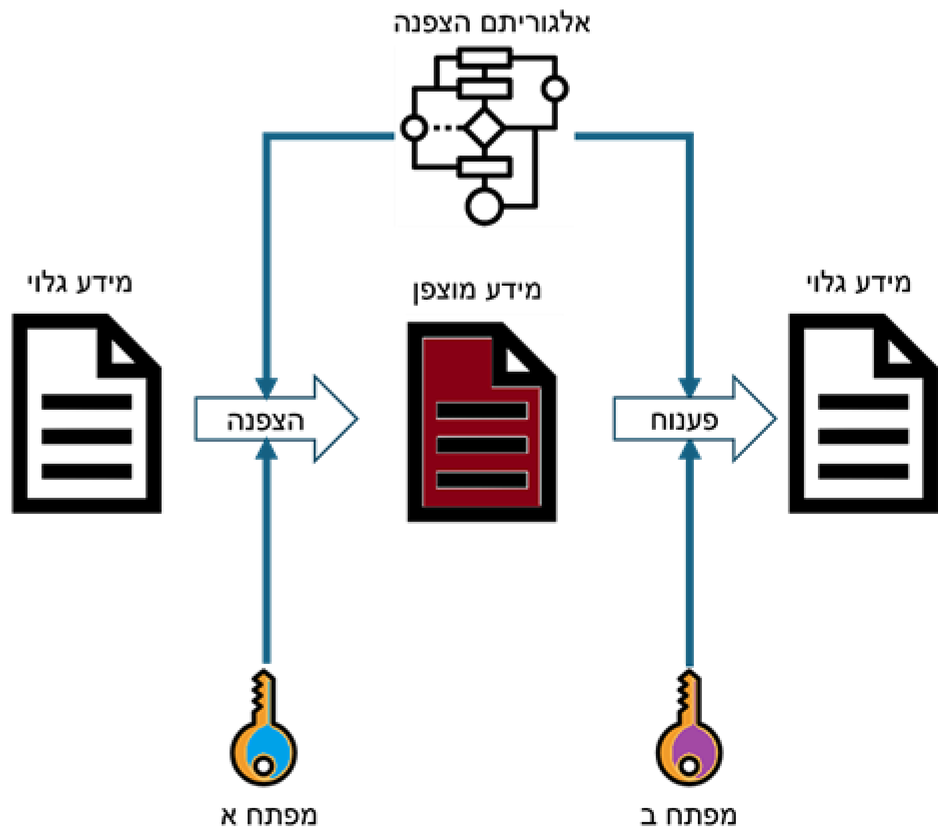
בדוגמה שלנו, אליס ובוב תיאמו ביניהם את הסוד המשותף 4, בלי שיהיה צורך שהם יעבירו אותו זה לזו באופן גלוי, ובלי שאיב יכולה להסיק מהו. כדי למצוא את הסוד המשותף של אליס ובוב, איב צריכה לעבור על כל האפשרויות בין 1 ל-18 (p פחות אחד). אם p הוא בגודל 2048 ביט, זו משימה לא פשוטה למחשב לא קוואנטי.

RSA

המדענים Rivest, Shamir ו-Adleman הציעו ב-1977 פתרון אחר לבעיית החלפת המפתחות. אגב, הזווית הישראלית – עדי שמיר הוא פרופ' במכון ויצמן. אלגוריתם RSA הקרוי על שמם הוצע כאלגוריתם ההצפנה הא-סימטרי הראשון בעולם. עם זאת, חשיבותו הרבה היא דווקא בהיבטים אחרים: החלפת מפתחות סימטריים, וחתימה (נושא שנבין בקרוב). לפני הכל, אנחנו צריכים להבין כיצד RSA הוא צופן א-סימטרי. מהו בכלל צופן א-סימטרי?

ניזכר ברעיון של הצפנה סימטרית. יש לנו אלגוריתם, ומפתח הצפנה ששני הצדדים משתמשים בו. הצד המצפין והצד המפענח שניהם זקוקים לאותו המפתח.

ב-RSA, נייצר זוג מפתחות שיש ביניהם קשר מתימטי. מפתח אחד – לא משנה איזה מהם – ישמש להצפנה, והמפתח השני ישמש לפענוח.



בעיית פירוק מספר לראשוניים

ממציאי ה-RSA חיפשו בעיה מתימטית שקשה מאוד למחשבים לפתור. RSA הסתכלו על מספרים שהם Semi Prime, כלומר מספרים שאינם ראשוניים אך הם כפל של שני מספרים ראשוניים. פירוק של מספר Semi Prime לשני מספרים ראשוניים נחשבת בעיה קשה. לדוגמה – פירוק של 91 למספרים הראשוניים 7, 13. או לדוגמה, פירוק של המספר 17,467,727 למספרים 2357 ו-7411.

המחשבה הראשונה שחולפת כנראה בראשכם, היא שאפשר לקחת רשימה של כל המספרים הראשוניים ופשוט לעבור עליהם. כלומר, כדי למצוא את הפירוק של 91 למספרים ראשוניים, ניקח את רשימת כל המספרים שקטנים מהשורש הריבועי של 91 והם ראשוניים. אם כך נצטרך לבדוק את 2, 3, 5 ו-7. אין צורך לבדוק את 4, 6, ו-8 שאינם ראשוניים. אם מצאנו ש-7 הוא מחלק של 91, אז באופן מיידי מצאנו גם את 13.

זו אכן שיטה מוצלחת כשמדובר במספרים קטנים יחסית, אבל הבעיה היא שכשעולים למספרים גבוהים, יש הרבה מספרים ראשוניים:

- אם ניקח מספר דצימלי שניתן לייצג אותו ב-512 ביט (מספר דצימלי בן 155 ספרות), יש עשר בחזקת 152 מספרים ראשוניים הקטנים ממנו.
- אם נעלה למספרים בני 1024 ביט, יש עשר בחזקת 304 ראשוניים שניתן לייצג בתחום הזה.
- בין המספרים שניתן לייצג באמצעות 2048 ביט, יש עשר בחזקת 617 מספרים ראשוניים. אין היתכנות מעשית לבדוק את כולם, ואפילו לא לשמור את כולם בזיכרון של מחשב.

מדוע פירוק מספר Semi Prime נחשב בעיה קשה? מכיוון שלא מוכר אלגוריתם שפותר את הבעיה הזו בסיבוכיות פולינומיאלית (פונקציה פולינומית היא פונקציה שהיא סכום של חזקות, לדוגמה $x^2 + 3x$. כדוגמה לבעיה שיש לה פתרון בסיבוכיות פולינומיאלית, אם תבקשו למיין מערך, הזמן שייקח למיין את המערך תלוי באורך המערך, שנסמן אותו ב-n. התלות היא מסדר גודל של האורך כפול לוג של האורך

$$n \log(n)$$

מה יקרה אם ניקח מערך ארוך פי ארבע?

זמן המיון של המערך החדש יהיה מסדר גודל של האורך החדש:

$$4n \log(4n)$$

שהינו:

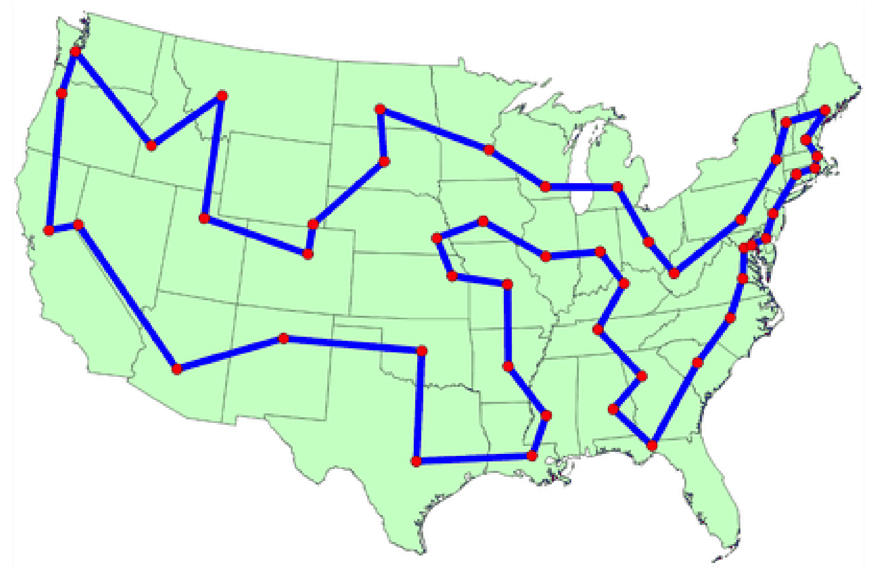
$4n\ (\log(4) + \log\ (n))$

היחס בין זמן המיון החדש לזמן המיון הקודם יהיה

$4n\ (\log(4) + \log(n))/n\ \log(n)\ = 4\ (\log(4) + \log(n))/\log(n)$

זמן המיון יתארך בין פי ארבע לפי חמש, תלוי ב-n. מחשבים יכולים לחיות עם זה. ניקח מחשב חזק פי חמש ונפתור את הבעיה, הרי מהירות המעבדים ממילא מכפילה את עצמה בערך פעם בשנה וחצי (חוק מור).

לעומת זאת, אם תתנו למחשב למצוא את המסלול הקצר ביותר שעובר בין כמות כלשהי של ערים (בעיית הסוכן הנוסע), הסיבוכיות של האלגוריתם היא אקספוננציאלית (פונקציה אקספוננציאלית היא פונקציה שבה המשתנה משפיע על החזקה, לדוגמה שתיים בחזקת x). כלומר, אם תכפילו את כמות הערים פי ארבע, משך החישוב יעלה בחזקת ארבע. וזה כבר משהו בעייתי. כי קצב ההתקדמות של מחשבים לא יוכל להשיג אלגוריתם אקספוננציאלי. אם המחשבים רק יתחילו להתקרב לפתרון הבעיה בזמן סביר, נוסיף כמה ערים לחישוב.



בעיית הסוכן הנוסע. קרדיט - https://optimization.mccormick.northwestern.edu/

בעיות שניתן לפתור בזמן פולינומיאלי נקראות בעיות P ובעיות שלא ניתן לפתור בזמן פולינומיאלי נקראות בעיות NP. האם יכול להיות שיש בעיות שאנחנו חושבים שהן NP, פשוט מכיוון שטרם מצאנו אלגוריתם פולינומיאלי בשבילן? יכול להיות שיש כאלה, אבל השאלה שמטרידה מדעני מחשב היא האם **בהכרח** לכל בעיה יש פתרון

פולינומיאלי. כלומר, שכל הבעיות NP הן בעצם P, ורק צריך למצוא את האלגוריתם המבריק. מי שיצליחו להוכיח P=NP יזכו בפרס של מיליון דולר, בפרסום עולמי, ובידיעה שהרסו את האפשרות ליצור אינטרנט מאובטח.

ובכל זאת, בחירה לא מושכלת של מספרים ראשוניים עלולה לאפשר שבירה מהירה יחסית של RSA.

התבוננו במספר הבא, האם הוא ראשוני? נסו למצוא את הפתרון בלי מחשבון ובלי לבדוק כל אחת מהאפשרויות.

4891

נסו לפתור את השאלה לבד לפני שתקראו את התשובה.

ובכן, אפשר לפרק את המספר לחיסור ריבועים:

$$4891 = 4900 - 9 = 70^2 - 3^2 = (70+3)(70-3) = 73 * 67$$

אם נכליל את המסקנה שעולה מהתרגיל, אם בחרנו שני מספרים ראשוניים שהם קרובים יחסית לשורש המספר ה-Semi Prime, אפשר יחסית במהירות למצוא את הפירוק.

אלגוריתם פשוט יהיה:

-נתון מספר N שנבקש לפרק. ידוע שהוא Semi Prime.

-כמספר התחלתי a נקבע את שורש N. נעגל למספר השלם התחתון.

-נחשב את $N - a^2$. לתוצאה נקרא b.

-נבדוק האם ל-b יש שורש שהוא מספר שלם. אם כן – סיימנו. אם לא – נקטין את a באחד.

פרקו את המספר הבא, שגודלו 100 ביט (נתון בבסיס 16):

0x221d49b02a45b172de232b09b

טיפ: התקינו את חבילת numpy של פייתון, פונקציית ה-sqrt שלה רצה בצורה מהירה. כדי לבדוק אם מספר הוא שלם, השוו את ה-ceil שלו ל-floor שלו.

אלגוריתם יצירת זוג מפתחות

עד כאן הדילוג הקטן שעשינו אל הרקע המתימטי שמאחרי הבעיה שבחרו RSA. וכעת, לתהליך יצירת המפתחות. נתחיל מבחירת שני מספרים ראשוניים, p ו-q. למתעניינים, אפשר לקרוא על מבחן מילר-רבין, המשמש לבדיקה שהמספרים אכן ראשוניים.

נחשב את המכפלה (p-1)(q-1) ונקרא לה T, קיצור של Totient.

נבחר מספר כלשהו, שנקרא לו E. המספר E צריך לקיים שלושה תנאים:

א. ראשוני

ב. קטן מ-T

ג. החלוקה של T ב-E נותנת שארית שאינה אפס. כלומר E אינו מחלק של T.

אם עמדנו בתנאים האלה יש לנו את המפתח הראשון.

המפתח השני, נקרא לו D, הוא מספר שיקיים את התנאי הבא:

$$D * E \ mod \ T = 1$$

כדי למצוא אותו, נעבור על כל האפשרויות.

הניחו P=17, Q=23. הניחו שבחרנו E=113. באמצעות סקריפט פייתון, מיצאו את המפתח השני. תוך כדי, בידקו שהבחירה של E עומדת בשלושת התנאים.

פתרון: 81.

הצפנה ופענוח

לאחר שיש בידינו את N, E, D נוכל לבצע את תהליך ההצפנה והפענוח.

תהליך ההצפנה-ניקח מסר כלשהו ונהפוך אותו למספר. לדוגמה, התו "a" יכול להיות מיוצג באמצעות קוד ה-ascii שלו, 97. את המספר הגלוי, נקרא לו m, נעלה בחזקת המפתח E ולתוצאה נבצע מודולו N:

$$Cipher = m^E \ mod \ N$$

או בכתיב פייתוני:

```
Cipher = m**E % N
```

זהו, יש בידינו מסר מוצפן.

תהליך הפענוח עושה שימוש בחלקו השני של צמד המפתחות, שקראנו לו D:

$$Plain = Cipher^D \ mod \ N$$

או בכתיב פייתוני:

Plain = cipher**D % N

להלן דוגמה. אנו נשתמש במספרים של התרגיל הקודם – 17, 23 יוצרים לנו את N=391. נבחר E=113 ולכן D=81, כפי שחישבנו מראש.

כעת נצפין את התו "a", 97, שהופך להיות 148.

הפענוח של 148 מחזיר אותנו ל-97. הידד! הקסם עובד.

```
>>> P = 17
>>> Q = 23
>>> N = P*Q
>>> N
391
>>> E = 113
>>> D = 81
>>> m = ord("a")
>>> m
97
>>> cipher = m**E % N
>>> cipher
148
>>> plain = cipher**D % N
>>> plain
97
```

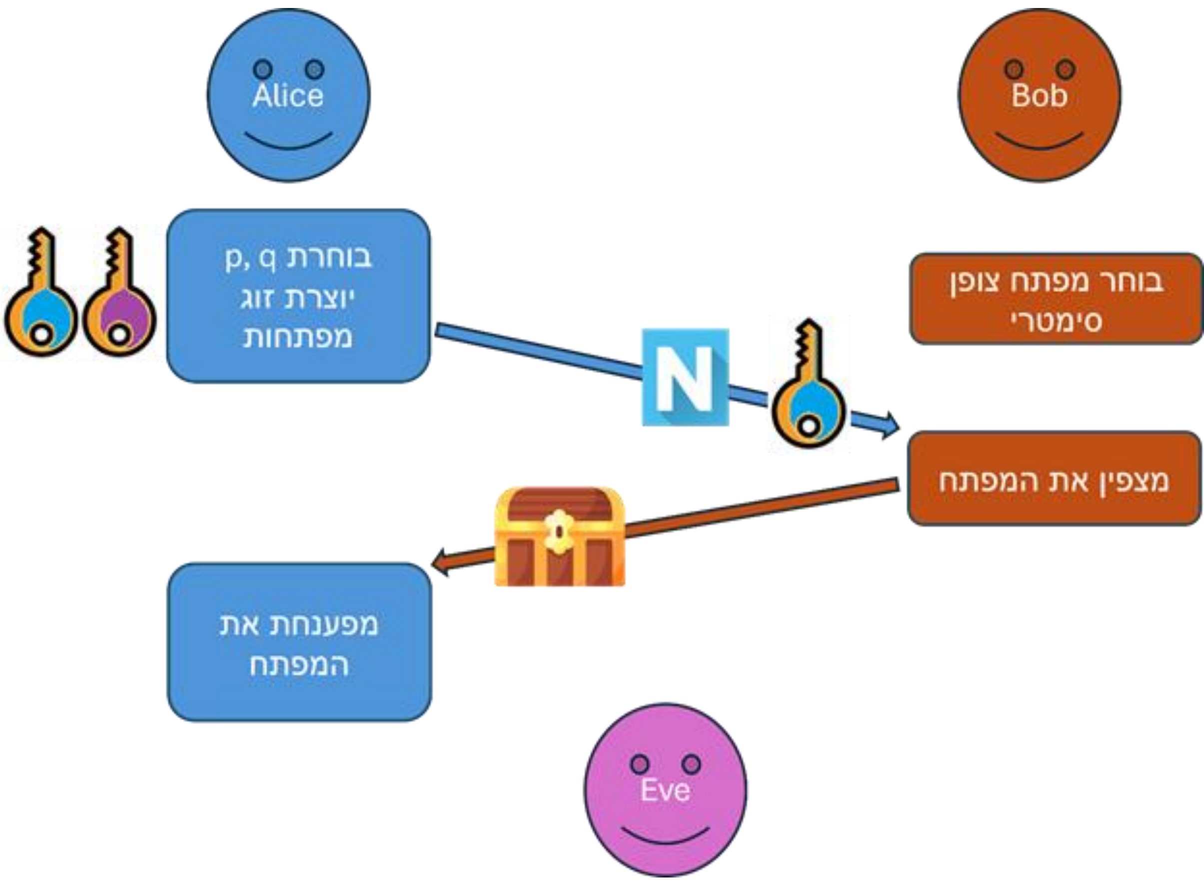
זו דוגמה להצפנה א-סימטרית, שמשתמשת בצמד מפתחות.

הבעיה היא, שביצוע ההצפנה הזו יקר חישובית. במקרה שלנו, ביצענו העלאה של מספר בחזקת 113. וזאת כאשר בחרנו P, Q קטנים יחסית. בעולם האמיתי, כדי למנוע פירוק של N, נצטרך לבחור מספרים גדולים, כך ש-N יהיה בגודל 2048 ביט. העלאה בחזקה תהיה יקרה בזמן. ויתר על כן, יחסית לצפנים סימטריים, שכפי שראינו מורכבים מפעולות ביטיות זולות חישובית. ובכן, מה נעשה?

שימוש ב-RSA לטובת החלפת מפתחות

הטכניקה הנפוצה היא להשתמש בצופן RSA כדי לתאם מפתח סימטרי, ולאחר שהמפתח תואם בין הצדדים עוברים להצפנה סימטרית. בצורה זו נהנים משני העולמות: הצדדים נהנים מהצפנה פשוטה לחישוב, תוך החלפת מפתחות הצפנה בצורה שגם אם הערוץ חשוף להאזנות לא ניתן לחשב את מפתח הצופן, ועם כל זאת – הפעולה היקרה של RSA מתבצעת רק פעם אחת.

למעשה, החלפת מפתח הצפנה ומעבר להצפנה סימטרית זה בדיוק מה שעשינו עם Diffie Hellman. שני צדדים שרוצים להחליף מפתחות סימטריים יכולים לבחור אם להשתמש באלגוריתם RSA או ב-DH.



החלפת המפתחות מתבצעת כך:

- אליס תבחר שני מספרים ראשוניים p, q ותחשב את המכפלה N. נניח ש-N הוא 391, בהתאם לדוגמה שסקרנו לפני כן.
- אליס תייצר זוג מפתחות. נניח שהמפתחות הם 113 ו-81, שוב בהתאם לדוגמה שסקרנו.

- אליס תעביר לבוב את המפתח 113. מעתה המפתח הזה ייקרא "אקספוננט", או E, מכיוון שבפעולת ההצפנה מעלים בחזקת המפתח הזה.
- אליס תעביר לבוב גם את N. מעתה נקרא לו "מודולוס", מכיוון שמשתמשים בו בפעולת המודולו.
- בוב בוחר מפתח הצפנה סימטרי. נניח שבוב בחר את המפתח 97.
- בוב משתמש באקספוננט ובמודולוס כדי להצפין את המפתח שבחר. במקרה שלנו, הוא יחשב את התוצאה – 148. בוב מעביר את התוצאה לאליס.
- אליס משתמשת במפתח השני, שנשאר רק ברשותה, כדי לפענח את המסר המוצפן שבוב שלח לה, ומוצאת כי המסר הוא 97.

כעת אליס ובוב חולקים את אותו מפתח הצפנה סימטרי.

ומה עם איב? איב, שהאזינה לכל התקשורת ביניהם, לא יודעת מה המפתח. הדרך היחידה של איב למצוא את המפתח שבוב בחר, היא לפרק את N, המבוסס על שני המספרים הראשוניים שאליס בחרה. הפירוק של N יאפשר לאיב לחשב את המפתח שאליס השאירה רק ברשותה ולפענח את המסר המוצפן שבוב שלח. אם אליס בחרה p, q גדולים ולא קרובים זה לזה, איב תצטרך לפתור בעיה שמחשבים קלאסיים לא מסוגלים לפתור בזמן סביר.

סיכום

בפרק זה סקרנו שתי שיטות להתמודדות עם האתגר של החלפת מפתחות סימטריים בערוץ תקשורת שנתון להאזנות. שתי השיטות משתמשות בבעיות מתמטיות שקשה למחשב קלאסי לפתור, לכן הן נחשבות בטוחות.

אלגוריתם דיפי הלמן מבוסס על כך שכל צד בשיחה תורם את חלקו ליצירת מפתח הצפנה סימטרי משותף.

אלגוריתם RSA מבוסס על כך שצד אחד יוצר זוג מפתחות ואילו צד אחר יוצר מפתח הצפנה סימטרי. אלגוריתם RSA יכול לשמש גם כאלגוריתם הצפנה, אולם עקב המורכבות החישובית הרבה שלו, הוא אינו נפוץ בשימוש כזה.

דיפי הלמן וגם RSA ישמשו להעברת מפתחות בין הצדדים, ולאחר מכן נשתמש באלגוריתם הצפנה סימטרי. בכך, נראה שהתקדמנו מאוד בפתרון בעיית ההצפנה. אולם עדיין לא פתרנו את כל הבעיות שהוצגו בפרק הפתיחה בנוגע לסוקטים מוצפנים. כיצד אפשר לדעת אם מישהו מנסה להתחזות למישהו אחר? או משנה את המידע שמישהו שלח? על כך – בפרק הבא.

פרק 18: Integrity ו-Authentication

בפרקים הקודמים סקרנו את בעיית ההצפנה ואת הבעיה הנלווית אליה, בעיית החלפת המפתחות.

פתרון בעיות אלה, כפי שנכתב בפרק המבוא לסוקטים מוצפנים, אמנם מבטיח Confidentiality, אך אינו מבטיח מעבר לכך. האקר זדוני שנמצא בין השרת והלקוח יכול לשנות ביטים גם בפקטה מוצפנת, וההצפנה גם אינה מבטיחה לבוב שהוא מדבר עם אליס. בוב צריך דרך לענות על השאלות הבאות:

-האם אליס שלחה לי את הפקטה?

-האם הפקטה הגיעה אלי ללא שינויים?

כדי להבין כיצד מתגברים על בעיות אלו, נלמד כיצד מייצרים דבר הנקרא HMAC. זהו קיצור של Hash-based Message Authentication Code.

הנה הסבר כללי על הרעיון, ומיד ניגש לפרטים. כדי לוודא שאף אחד לא משנה פקטה שהיא שלחה, אליס מוסיפה למידע שהיא שולחת לבוב תוצאה של חישוב מתימטי מיוחד, שרק בוב יכול לבדוק אותו. אם בוב מצליח לשחזר את התוצאה שאליס שלחה, הוא יכול לוודא שאף אחד לא שינה את המידע שהיא שלחה. אם האקר ינסה לשנות את המידע או את התוצאה של החישוב המיוחד, החישוב שבוב יעשה יוביל לתוצאה שונה והוא יידע שהמידע שהוא קיבל אינו מקורי מאליס.

פונקציות Hash

פונקציית Hash, או בעברית "גיבוב", היא פונקציה מתימטית חד-כיוונית. מה זה אומר? קל לחשב כיוון אחד, בלתי אפשרי (או כמעט בלתי אפשרי) לחשב את הכיוון ההפוך. כדי להמחיש את הרעיון, נתחיל מפונקציה מתימטית שאינה חד-כיוונית. לדוגמה, הזזה של ארבעה מספרים, מודולו עשר. קל לחשב שעבור המקור 2 התוצאה היא 6, עבור המקור 7 התוצאה היא 1. הפונקציה הזו גם הפיכה בקלות: אם ידוע לנו שהתוצאה היא 1, קל לחשב שהמקור היה 7.

לעומת זאת, הנה דוגמה לפונקציה חד-כיוונית: סכום הספרות במספר. קל לחשב שעבור המקור 123459 התוצאה היא 24, אבל אם נתונה לכם התוצאה 24 אי אפשר להפוך אותה חזרה למספר המקורי.

ועדיין, סכום הספרות במספר אינו פונקציית Hash "טובה". הנה כמה תבחינים שפונקציות Hash צריכות לקיים כדי להיות "טובות":

1. התוצאה היא באורך קבוע.
2. שינוי קטן במקור יוצר שינוי עצום בתוצאה.
3. אין שני מקורות שנותנים את אותה תוצאה.

פונקציית Hash שעונה על התבחינים האלה יכולה לשמש בתור Cryptographic Hash Function, כלומר להיות חלק מתהליך שבו הפונקציה מסייעת באבטחה.

מדוע התבחינים האלה חשובים? חשוב שלא יהיו כמה מקורות שנותנים את אותה תוצאה, מכיוון שאם אפשר למצוא מקור שמגיע לתוצאה נתונה של Hash, אז אפשר להחליף את המסר במסר שונה. לדוגמה, אם עבור Hash מסויים התוצאה של "ניפגש בשעה שמונה" זהה לתוצאה של "כרובית מטוגנת" אז תוקף זדוני יכול להחליף בין המסרים. וככל שיש יותר מקורות שמגיעים לאותה תוצאה, כך המלאכה של התוקף, למצוא עוד מסר שנותן את אותו ה-Hash, יותר קלה. דבר זה נקרא Hash Collision והוא תופעה לא רצויה ב-Hash.

בנוסף, חשוב ששינוי קטן במקור יוביל לשינוי עצום בתוצאה כדי להקשות על התוקף "להנדס" Hash Collision. אם שינויים קטנים במקור יקרבו את התוצאה לאט לאט לתוצאה הרצויה, אפשר יהיה באמצעות ניסוי וטעייה להצליח למצוא מקור שנותן Hash נתון.

כמובן שאין פונקציית Hash אידאלית, ואפשר רק לנסות ולהתקרב לדרישות הללו. הדבר נכון במיוחד לגבי בעיית המקורות השונים שעלולים להגיע לאותה תוצאה.

הפונקציה הראשונה שסקרנו, סכום הספרות, גרועה מכל שלוש הבחינות הללו. התוצאה אינה באורך קבוע, שינוי קטן במקור יוביל, בדרך כלל, רק לשינוי קטן בתוצאה, ויש מקורות רבים שנותנים את אותה התוצאה.
הנה פונקציית Hash משופרת מעט:

- כפלו את המקור ב-1234
- העלו את התוצאה בריבוע
- בחרו את הספרות במיקומים 4–9 (הספרה הכי ימנית היא מיקום 1)
- העלו את התוצאה בריבוע
- בחרו את הספרות במיקומים 4–8

מקור	תוצאה
99	57569
100	83553
101	64178

כפי שאפשר לראות, הפונקציה המשופרת עונה יפה על שני התנאים הראשונים: לתוצאה יש אורך קבוע, ושינוי קטן במקור מוביל לשינוי גדול בתוצאה. מה דעתכם, האם יכול להיות מצב שבו שני מקורות מובילים לאותה תוצאה? חישבו על כך.

התשובה היא שבהכרח כן. כיוון שהתוצאה היא בגודל קבוע, ומקבלת ערכים בין 0 ו-99999, יכולות להיות לכל היותר מאה אלף תוצאות שונות. כלומר, אם תבחרו מאה אלף ואחד מקורות, מובטח כי תהיה לפחות תוצאה אחת שתחזור על עצמה פעמיים. זהו "עקרון שובך היונים", או "עקרון דיריכלה", למעוניינים לקרוא על כך.

אין בעיה, במקום לבחור פונקציה שגודל התוצאה שלה הוא מספר קטן, נבחר פונקציה שגודל התוצאה בה הוא לדוגמה 160 ביט, כמו פונקציות Hash מודרניות. יש לה באופן תיאורטי 2 בחזקת 160 תוצאות שונות, אך חלק מהתוצאות יהיו זהות, כלומר יהיו Hash Collisions.

נקדיש פסקה קצרה להסבר תיאורטי על הסיכוי להתנגשויות.

עקרון יום ההולדת קובע, שאם יש בחדר 23 אנשים שנולדו בתאריכים אקראיים, אז יש סיכוי של 50% שלשניים מהם יש יום הולדת באותו תאריך. בגלל עקרון יום ההולדת עלולים להיות Hash Collisions הרבה לפני שבדקנו 2 בחזקת 160 אפשרויות. לפי עיקרון זה, עבור Hash שהתוצאה שלו בגודל 160 ביט, בסבירות של 50% יהיה Hash Collision בין שני מקורות כבר אחרי 2 בחזקת 80 מקורות שונים. ניסיון למצוא התנגשויות בשיטה זו נקרא "מתקפת יום הולדת".

אך גם מתקפת יום ההולדת החביבה אינה מבטיחה הצלחה, במיוחד אם במקום 160 ביט נבחר Hash בעל כמות אף גדולה יותר של ביטים. 256, 384 או 512 ביט, הגדלת מספר הביטים של Hash היא שיטה פשוטה כדי להגדיל את כמות האפשרויות ולצמצם את הסיכוי ל-Hash Collisions.

תרגיל 18.1 המצאת Hash

כיתבו פונקציית Hash כיד הדמיון הטובה עליכן. הפונקציה תיקח מסר כלשהו, תבצע עליו פעולות מתמטיות שונות ותהפוך אותו למספר בן 16 ביט.

אלגוריתמי Hash מודרניים כוללים בין היתר:

MD5 – קיצור של Message Digest Algorithm 5. פונקציה שהתוצאה שלה בת 128 ביט. הוכרז לא בטוח לשימוש ב-2008. המשמעות של "לא בטוח לשימוש" היא שאפשר לקחת תוצאות של הפונקציה, ולמצוא מקורות שיוצרים אותן.

SHA1 – קיצור של Secure Hash Algorithm, בן 160 ביט. הוכרז לא בטוח לשימוש ב-2017.

SHA2 – בעל גרסאות של 256, 384 ו-512 ביט. נמצא בשימוש נפוץ היום.

תרגיל 18.2 – Hash מודרניים

צרו שני קבצי טקסט, שיכילו מידע כמעט זהה. לדוגמה, הקובץ הראשון יכיל את הטקסט "hello cool cyber students", והקובץ השני יכיל את אותו טקסט אך עם נקודה בסוף.

היכנסו ל-Powershell והקלידו את הפקודה הבאה:

Get-FileHash filename -algorithm SHA256

צפו בהבדלים בין התוצאות.

חיזרו על החישוב עם SHA512.

```
Windows PowerShell
PS C:\Users\BARAK> Get-FileHash c:\networks\work\file1.txt -algorithm SHA256

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          E40F77138340A10D6BD2440DC029FF49FFC349232678024A6E78A256FF95391F       C:\networks\work\file1.txt


PS C:\Users\BARAK> Get-FileHash c:\networks\work\file2.txt -algorithm SHA256

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          27B8C1C660CB280B8EDCB8A59D2068EFA6657E56B7F123C0305F28926221F29F       C:\networks\work\file2.txt
```

כמו שרואים, שינוי קטנטן במקור גורם לשינוי עצום בתוצאה.

איך פונקציית ה-Hash יכולה לעזור לבוב לדעת שההודעה של אליס לא שונתה בדרך?

נניח שמה שאליס עשתה הוא לשלוח לבוב את ההודעה hello cool cyber students, ויחד איתה, בסוף ההודעה, היא צירפה את ה-SHA256 של המסר.

Hash(Message)	Message

כעת, בוב יכול לוודא שהמסר המקורי לא שונה, באמצעות ביצוע SHA256 על המסר hello cool cyber students. בוב יקבל את תוצאת ה-Hash שמתחילה בספרות E40, כפי שחישבנו בתרגיל, וישווה אותו לחישוב שאליס חישבה ושלחה לו.

מהן האפשרויות השונות לתוצאה ומה בוב יכול ללמוד מהן?

אם תוצאת החישוב שונה, המסקנה היא אחת – זה אינו המסר המקורי שאליס שלחה. אבל אם תוצאת החישוב היא זהה, זה עדיין משאיר את בוב עם מספר אפשרויות:

א. זהו המסר המקורי של אליס, ללא שינוי.

ב. האקר שינה גם את המסר המקורי של אליס וגם את ה-Hash שאליס שלחה. לדוגמה, ההאקר הוסיף למסר המקורי נקודה, וגם שינה את ה-Hash כך שיתחיל ב-27B וכו', כפי שביצענו בתרגיל.

ג. ההאקר לא שינה את ה-Hash, אבל החליף את המסר המקורי במסר אחר, שה-Hash שלו מסתבר להיות אותו Hash של המסר המקורי של אליס.

אפשרות ג' אינה צריכה להטריד מאוד את בוב. בוב סומך על כך ש-SHA256 אמור להיות עמיד בפני תעלולים כאלה; עם 256 ביט הסיכוי ל-Hash Collision הוא זעיר, גם עם מתקפת יום הולדת. קשה מאוד למצוא מקור שיתאים לתוצאת חישוב שברשותנו.

אפשרות ב', לעומת זאת, היא מטרידה מאוד. היא פשוטה למימוש.

אם כך, נצטרך להוסיף לאליס ובוב יכולת מתמטית נוספת, קסם אחרון שהם יוכלו להשתמש בו כדי להבטיח שההאקר החרוץ שבתווך לא יוכל לשבש את ההודעה שאליס שלחה לבוב.

HMAC

אליס ובוב הבינו שההאקר היושב בתווך ביניהם עלול לשנות בקלות את ההודעות שהם שולחים זה לזו, גם אם הם יוסיפו חישוב מבוסס Hash. טיכסו עצה והגיעו לרעיון הבא: אליס ובוב יתאמו ביניהם מפתח סודי מעל התווך המשותף. כשסקרנו את דיפי-הלמן ואת RSA, ראינו שאפשר לתאם מפתח סודי גם אם גורם זדוני מאזין בתווך. חשוב להדגיש שזהו מפתח נוסף על מפתח ההצפנה, שיש לו שימוש אחר לחלוטין. כדי להבדיל בין מפתח ההצפנה הסימטרית לבין מפתח ההצפנה של HMAC – לעיתים שניהם נקראים פשוט "secret key" וזה עלול לבלבל – למפתח הראשון נקרא "מפתח הצפנה" או Encryption Key ולמפתח השני נקרא "מפתח גיבוב" או Hashing Key.

כעת, אליס תשדרג את החישוב שלה. אם בחישוב הקודם היא השתמשה רק ב-Hash על ההודעה שלה, כעת היא תכניס לתוך החישוב גם את ה-Hashing Key. פונקציה שמקבלת גם Hash וגם Hashing Key נקראת Hash-based Message Authentication Code, או בקיצור HMAC.

Message	Hash(Hashing Key, Message)

לטובת פשטות ההסבר, נניח HMAC פשוט מאד. אליס לוקחת את ה-Hashing Key, כותבת אותו מצד שמאל, ולימינו כותבת את המסר שברצונה להעביר. על המחרוזת הזו היא מבצעת Hash. לדוגמה, נניח שהמסר נותר hello cool cyber students וה-Hashing Key שאליס ובוב החליפו הוא SECRET12. אליס כותבת אותם זה לצד זה:

SECRET12 hello cool cyber students

ולמחרוזת הזו מבצעת SHA256. בעקבות הוספת ה-Hash Key הסודי, ה-HMAC השתנה והוא כעת

64C8A06704D35BBA94E0246A1208981A9C052DE21D07AFFB094C86583EDC7293

(בידקו בעצמכם!)

אליס שולחת את המסר המקורי, יחד עם ה-HMAC. חשוב מאד להדגיש – אליס אינה שולחת את ה- Hashing Key! היא רק משתמשת בו לחישוב ה-HMAC. המפתח חייב להישאר רק ברשותה וברשותו של בוב.

מה שאליס תשלח הוא:

hello cool cyber students 64C8A0[…]

כאשר בוב מקבל הודעה מאליס, הוא לוקח את ה-Hashing Key שהם תיאמו – "12SECRET" – ומוסיף אותו למסר. הוא מבצע SHA256, ואם הוא מקבל את אותה התוצאה שהיתה מצורפת בתור HMAC, הוא יודע שזו הודעה מקורית של אליס.

כיצד התוספת של ה-Hashing Key פותרת את הבעיה?

היזכרו ממה אליס ובוב חששו: האקר שמשנה גם את ההודעה המקורית וגם את ה-Hash שלה, בצורה שתגרום לבוב להאמין שזו הודעה מקורית של אליס שהגיעה אליו ללא שינוי.

אם האקר שינה משהו, ה-HMAC לא יתאים למסר שהוא מגן עליו. ההאקר לא יוכל לשנות גם את המסר וגם את ה-HMAC בצורה שהם יתאימו זה לזה. לדוגמה, אם ההאקר מנסה לשנות את המסר כך שיהיה goodbye cool cyber students, הוא יודע רק חלק מהתווים שנכנסים ל-SHA256. חסר לו המפתח הסודי שאליס ובוב תיאמו ביניהם.

בפועל, פרטי המימוש של HMAC הם מעט יותר מורכבים מאשר לכתוב את ה-Hashing Key לפני המסר ולבצע Hash. תקן RFC 2104 מגדיר את התהליך. למתעניינים בפרטים:

א. בצעו XOR בין ה-Hashing Key לבין מחרוזת המוגדרת בתקן.
ב. שרשרו למחרוזת שקיבלתם את ההודעה שברצונכם לשלוח.
ג. בצעו Hash על המחרוזת המשורשרת, התוצאה של סעיף ב'.
ד. בצעו XOR בין ה-Hashing Key לבין מחרוזת נוספת המוגדרת בתקן.
ה. שרשרו את התוצאה של סעיף ד' עם התוצאה של סעיף ג'.
ו. בצעו Hash לתוצאה של סעיף ה'.

רגע אחד. האם השימוש ב-HMAC בעצם מאפשר להשתמש באלגוריתמי Hash פחות טובים?

נפרט את השאלה. כשלמדנו אלגוריתמי Hash, תיארנו שאלגוריתם טוב הוא כזה שיש בו כמה שפחות Hash Collisions, כלומר קשה למצוא מספר מקורות שיוצרים את אותה תוצאה. כדי שלדוגמה אי אפשר יהיה להחליף את ההודעה "ניפגש בשעה שמונה" בהודעה "כרובית מטוגנת", אם יש להודעות הללו במקרה את אותו ה-Hash.

זו אינה בעיה תאורטית. אלגוריתם SHA1 "נשבר". כלומר, מומחים מסוגלים למצוא עבור Hash נתון מקור שנותן אותו. למעוניינים בהסבר, מומלץ הסרטון הבא:

How we created the first SHA-1 collision and what it means for hash security -Defcon 25

אבל התוספת של Hashing Key ו-HMAC משנה את כל המשחק. להודעה "ניפגש בשעה שמונה" מצורף לא סתם Hash אלא HMAC. יש מפתח סודי, שאינו ידוע לאף אחד חוץ מאשר שני הצדדים לתקשורת. כעת, מי שרוצה לשנות את תוכן ההודעה צריך למצוא הודעה אחרת, שצירוף שלה למפתח סודי יוצר Hash מסוים. המפתח הסודי הופך את המשחק להרבה יותר מסובך עבור מי שרוצה לשנות את ההודעה.

אז אם כך, לשם מה צריך עדיין להקפיד על אלגוריתמי Hash "טובים"?

התשובה היא שה-Hashing Key אמנם מאוד מקשה על שינוי תוכן ההודעה, אבל לאלגוריתמי Hash יש שימוש נוסף, והשימוש הזה רגיש מאד ל-Collisions. שימוש זה הוא חתימה דיגיטלית.

חתימה דיגיטלית

זוכרים את בעיית הכחשת העסקה שדנו בה?

בוב מחזיק מסר שכתוב בו "אני אליס ואני חייבת לך 100$". למסר הזה נוסף HMAC שמגן עליו, על פי המפתח שבוב ואליס החליפו. בוב אומר לאליס – תני לי את הכסף. אליס משיבה – "רק רגע בוב! יש עוד מישהו שיכול היה לייצר את המסר הזה... אתה!"

ואכן, אליס צודקת. לבוב יש את ה-Hashing Key שאליס משתמשת בו, והוא יכול תיאורטית לזייף כל הודעה ולטעון שהיא של אליס.

בוב גם עלול להעביר את המפתח לגורם אחר. נניח שבוב מעוניין למכור לצד שלישי את ההתחייבות של אליס. לדוגמה, בוב פונה לחברו צ'רלי ואומר לו, "צ'רלי, תלווה לי 90$ וכערבון אתן לך את ההתחייבות של אליס לשלם לי 100$". צ'רלי משיב לבוב, "תמורת ההתחייבות של אליס אני מוכן לתת לך כסף, אבל תצטרך לתת לי את ה-Hashing Key שתאמת איתה, כי אחרת אני לא יכול לוודא שזוהי אכן התחייבות מקורית של אליס". בוב מוסר לצ'רלי את ה-Hashing Key המדובר, צ'רלי משנה את המסר שבהודעה, מחשב את ה-HMAC הנכון ופונה לאליס "היי אליס, יש לי התחייבות חתומה שלך לשלם 1000$"...

אמנם השגנו Integrity ו-Authentication, אבל טרם השגנו Non Repudiation. כלומר, מי שחתם על עסקה עלול להכחיש אותה, מהסיבות שסקרנו כאן.

לאחר שהבנו את הבעיה, נוכל לדון בפתרונות.

הפתרון הראשון הוא אימות של המשתמש. אם התחברתם לבנק שלכם או לאתר קניות, נדרשתם לספק סיסמת כניסה הידועה רק לכם. לעיתים גם תקבלו קוד דרך הטלפון הסלולרי או המייל. מידע שידוע רק למשתמש הוא הדרך

שבוחרים נותני שירות דיגיטלי בדרך כלל כדי להבטיח שהמשתמש הוא אכן מי שהוא נחזה להיות. בקיצור, כאשר שרת רוצה לאמת לקוח, הוא יבקש ממנו סיסמה.

אך מה כאשר הלקוח רוצה לאמת את השרת? התחברתם לאתר של הבנק שלכם – איך תוכלו לוודא שזהו אכן הבנק שלכם? הרי לא הגיוני שנבקש מהבנק שלנו, או מכל שירות אחר שאנחנו מתחברים אליו, שיזינו סיסמת כניסה כדי לאמת את עצמם בפנינו...

הפתרון הנפוץ לאימות צד השרת הוא חתימה דיגיטלית.

הרעיון של חתימה דיגיטלית הוא שלכל צד לתקשורת יהיה מזהה כלשהו שהוא אינו חולק עם אף אחד אחר, והמזהה הזה ישמש להוכחת הזהות שלו. איך נעשה את זה?

ההצפנה הא-סימטרית באה לעזרתנו. היזכרו במה שלמדנו – ישנם שני מפתחות, אחד משמש להצפנה והשני לפענוח.

בתיאור הבא, אליס היא דוגמה ל"בנק", או ספק שירותים דיגיטלי כלשהו, ואילו בוב הוא הלקוח, שמבקש לאמת את זהות הבנק.

אליס תיצור זוג מפתחות א-סימטריים. לאחד היא תקרא "מפתח פרטי" ולשני "מפתח ציבורי". אין זה משנה לאיזה מהם היא תקרא פרטי ולאיזה ציבורי, כל עוד היא תקפיד שלא לחלוק עם אף אחד את המפתח הפרטי שלה. כעת אליס מוכנה להתחיל: היא תיקח את המסר שהיא מעוניינת להעביר לבוב ותבצע עליו Hash. את התוצאה של ה-Hash היא תצפין באמצעות המפתח הפרטי שלה.

אליס תעביר לבוב ארבעה דברים:

-את המסר שברצונה לשלוח

-את התוצאה של ה-Hash כאשר הוא מוצפן באמצעות המפתח הפרטי שלה, כמו באיור הבא:

Message	Encrypt(Hash(Message) , Private Key)

-את המפתח הציבורי שלה

-את N (המודולוס, מכפלת שני המספרים הראשוניים שאליס בחרה)

מה יעשה בוב?

בוב לוקח את המסר שאליס שלחה, מחשב את ה-Hash שלו ומניח אותו בצד.

בוב לוקח את תוצאת ה-Hash שאליס חישבה והצפינה, ומשתמש במפתח הציבורי ובמודולוס של אליס כדי לפענח אותו. התוצאה אמורה להיות חישוב ה-Hash המקורי, לפני שאליס הצפינה אותו. אם החישוב הזה זהה ל-Hash שבוב חישב בעצמו, על סמך המסר שאליס שלחה, הרי שזה מסר מקורי של אליס. גורם שאינו מחזיק במפתח הפרטי של אליס לא היה יכול לבצע את החישוב הזה. לכן, בוב יודע גם שאליס שלחה את ההודעה, וגם שההודעה לא שונתה בדרך.

רגע אחד, אם אליס משתמשת במפתח הפרטי שלה, בשביל מה היא צריכה לבצע גם Hash על תוכן ההודעה? אליס יכלה לשלוח את תוכן ההודעה המקורי ולצדו את תוכן ההודעה כשהוא מוצפן עם המפתח הפרטי שלה. אם ההודעה המפוענחת אצל בוב מתאימה לתוכן ההודעה המקורית – זוהי הודעה של אליס.

ובכן, ההודעה המקורית של אליס עשויה להיות ארוכה מאד, וכזכור, חישוב של הצפנה א-סימטרית דורש משאבי עיבוד רבים, לעומת Hash, שמורכב מפעולות מתמטיות פשוטות יחסית. לאחר ה-Hash נקבל מסר קצר יחסית, שקל להעביר הצפנה א-סימטרית.

סיבה נוספת נובעת מעיקרון בתורת ההצפנה, שמאוד לא רצוי להציג מסר גלוי לצד אותו המסר בצורה מוצפנת. הדבר עלול להקל על מפצחי צפנים למצוא את מפתח ההצפנה.

כעת, נמחיש באמצעות דוגמה את הצעדים של אליס ובוב.

נניח כאמור שהמסר של אליס הוא "I owe you $100".

נניח שאליס בחרה N=391, וייצרה צמד מפתחות: פרטי – 113, ציבורי – 81.

נניח Hash פשוט, שהפך בדרך כזו או אחרת את המסר של אליס למספר 98. אפשר היה להשתמש ב-SHA256, אבל לטובת ההמחשה נסתפק במספר קטן.

I owe you 100$	Encrypt(98, 113)

אליס מצפינה את ה-Hash עם המפתח הפרטי שלה:

$$Cipher = 98^{113}\ mod\ 391 = 285$$

לכן אליס תשלח לבוב את ההודעה הבאה:

I owe you 100$	**285**

בנוסף, אליס תשלח לבוב את המספרים 81 (מפתח ציבורי), 391 (מודולוס).

בוב קיבל את ההודעה וצריך לוודא שהיא לא שונתה.
בשלב הראשון, בוב יבצע Hash למסר "I owe you $100". כיוון שבוב ישתמש באותו אלגוריתם Hash של אליס, התוצאה שלו תהיה זהה – 98.

בשלב השני, בוב יבדוק אם החתימה מעידה שהמסר הוא של אליס ועבר ללא שינוי. בוב יפתח את החתימה הדיגיטלית באמצעות המפתח הפומבי:

$Plain = 285^{81}\ mod\ 391 = 98$

בוב קיבל גם כאן 98, ומכאן שהכל תקין. אילו מישהו היה משנה משהו בהודעה, החתימה לא היתה מתאימה. אי אפשר לשנות את החתימה בלי לדעת את המפתח הפרטי של אליס.

...אי אפשר לשנות, אלא אם כן ה-Hash ששימש אותנו בתהליך אינו מספיק טוב, ויש לתוקף אפשרות למצוא Collision. במקרה כזה, תוקף אינו צריך לשנות את החתימה. להיפך, הוא ישתמש בה. התוקף יחליף את המסר במסר אחר שיש לו את אותו ה-Hash. החתימה הדיגיטלית, שפועלת רק על התוצאה של ה-Hash, תישאר אותו דבר. התוצאה תהיה שאליס חתמה כביכול באמצעות המפתח הפרטי שלה על הודעה שונה מאשר ההודעה שהיא באמת שלחה.

סיכום – איך הכל מתחבר (בינתיים)

התחלנו את הפרק עם הבנה שהצפנה לבדה אינה פותרת את כל בעיות האבטחה של סוקטים. לטובת Integrity ו-Authentication יש צורך במנגנונים נוספים.

בפרק זה, למדנו על שלושה מנגנונים מתמטיים – Hash, HMAC וחתימה דיגיטלית. כעת, נחבר אותם כדי להבין טוב יותר את התפקיד של כל אחד מהם ואיך הם מאפשרים Integrity ו-Authentication.

ה-Hash אינו עומד בפני עצמו. הוא משולב או עם Hashing Key ליצירת HMAC, או עם מפתח פרטי של הצפנה א-סימטרית ליצירת חתימה דיגיטלית. מה שחשוב הוא שה-Hash יהיה מתאים לשימושים קריפטוגרפיים. כלומר, שלא יהיה קל למצוא Collisions – מספר מקורות שה-Hash שלהם זהה.

בתחילת ההתקשרות, אליס (שהיא כזכור השרת, לדוגמה שרת של בנק) תצטרך להוכיח את הזהות שלה מול בוב. אליס תשתמש בחתימה דיגיטלית, כלומר מפתח פרטי ו-Hash, כדי להוכיח לבוב שהיא אכן היא. רק מי שיש לו את המפתח הפרטי של אליס יוכל לעשות זאת. כלומר, בוב השיג Authentication מול אליס. אליס בדרך כלל לא תבקש מבוב חתימה דיגיטלית משלו, אלא תוודא את הזהות שלו, הלקוח, באמצעות סיסמה שהמשתמש יזין.

אם בוב וידא בהצלחה שהצד השני היא אליס, כבר אין צורך שאליס תחתום על כל הודעה עם חתימה דיגיטלית. במקום זה, אליס ובוב ישתמשו ב-Hashing Key הידוע רק לשניהם ויבצעו באמצעותו HMAC על ההודעות שלהם. ה-HMAC מאפשר להם לוודא שני דברים בו זמנית. ראשית, שהצד השני הוא זה ששלח להם את ההודעה (ולא מתחזה כלשהו), ובכך נמשך ה-Authentication גם להודעות הבאות שאינן חתומות דיגיטלית. שנית, שההודעה ששלח הצד השני לא שונתה על-ידי אף אחד אחר בדרך. בכך משיגים גם Integrity.

התוצאה של Integrity יחד עם Authentication היא Non Repudiation, או אי יכולת הכחשה. אם אפשר להוכיח שהודעה נשלחה על-ידי אליס, ושההודעה לא שונתה על-ידי אחד, אז אליס אינה יכולה להכחיש שהיא שלחה את מה שכתוב בהודעה.

אם בוב מחזיק בהודעה חתומה דיגיטלית והוא הולך לבית משפט וטוען "ראו, אליס חייבת לי 100$, הנה מסר חתום על ידה", בית המשפט חייב לפסוק לטובתו.

או שאולי בעצם, לאליס כן יכולה להיות טענה כלשהי שתחלץ אותה? חישבו על כך.

אליס יכולה לטעון רק דבר אחד. "בוב היקר, המפתח שאתה חושב שהוא המפתח הציבורי שלי, כלל אינו המפתח הציבורי שלי. מישהו אחר העביר לך מפתח ציבורי ומודולוס, סיפר לך שהם שלי, אבל לא כך. זו אינה החתימה שלי על החוב שלך. אולי חברך צ'רלי עקץ אותך. תמיד חשדתי שהוא מפוקפק".

מה בוב יכול לעשות כדי למנוע את הבעיה הזו? על כך בפרק הבא – סרטיפיקטים.

18.3 תרגיל מסכם: סוקטים מאובטחים

בתרגיל זה נממש את העקרונות של TLS Handshake בין שרת ולקוח. בהמשך נלמד את התהליך הזה בפירוט, והתרגיל שנבצע כאן יעזור בהבנת החומר. המימוש שלנו לא יהיה זהה כמובן למה שמתבצע בפועל, אך הוא ימחיש חלקים מרכזיים מהתהליך.

כחלק מהתרגיל נממש את ארבעת המנגנונים הנדרשים ליצירת סוקט מוצפן:

1. הצפנה סימטרית המבוססת על מפתח שידוע רק לשני הצדדים
2. מנגנון קביעת מפתח הצפנה משותף
3. פונקציית גיבוב (Hash)
4. יצירת חתימה דיגיטלית באמצעות מפתח פרטי

התרגיל ידריך כיצד לבנות את המנגנונים. להלן תיאור של התהליך, ולאחריו חלוקה מומלצת לפונקציות.

<u>שלבי התקשורת על גבי הסוקט</u>

1. טרם תחילת ההתקשרות, השרת ייצור מפתחות RSA ציבורי E ופרטי D, נוסף כמובן למודולו N.
2. הלקוח יוזם את התקשורת עם השרת עם הודעה שמכילה את הטקסט "Hello".
3. השרת שולח ללקוח את המפתח הציבורי שלו ואת המודולו. כעת נניח שהחלפת המפתחות מתבצעת באמצעות אלגוריתם Diffie Hellman.
4. השרת שולח ללקוח את p, g של אלגוריתם DH
5. הלקוח בוחר ערך DH סודי a, מחשב את הערך הציבורי לפי DH –

$$A = (g^a)\ mod\ p$$

ושולח את A לשרת.

6. השרת בוחר ערך DH סודי b, מחשב את הסוד המשותף על סמך הערך הציבורי שהלקוח שלח לו :

$$Secret = (A^b)\ mod\ p$$

7. השרת מחשב את הערך הציבורי שלו לפי DH –

$$B = (g^b)\ mod\ p$$

אך הוא עדיין לא שולח את B ללקוח!

8. השרת חותם על B באמצעות מפתח ה-RSA הפרטי שלו D:

$$Signature = (B^D)\ mod\ N$$

ושולח את החתימה ללקוח.

9. הלקוח מחלץ מהחתימה את B:

$$B = (Signature^E)\ mod\ N$$

10. הלקוח מחשב את הסוד המשותף שלו עם השרת:

$$Secret = (B^a)\ mod\ p$$

11. בשלב זה, אמור להיות בידי השרת והלקוח סוד משותף זהה. הדפיסו אותו למסך, לטובת חיווי שהכל תקין עד כה.

12. השרת והלקוח ישתמשו בסוד המשותף ויגזרו ממנו Encryption Key ו-Hashing Key. פירוט בתיאור הפונקציה.

13. הלקוח ישלח לשרת מסר כלשהו. שלבי הכנת המסר:
    א. הודעה כלשהי
    ב. חישוב HMAC
    ג. הצפנה של ההודעה וה-HMAC בעזרת הצפנה סימטרית פשוטה (XOR).

14. השרת יפתח את ההצפנה ויבדוק האם ה-HMAC מתאים למסר. אם כן, המסר יודפס למסך המשתמש והתוכנית תסיים את הריצה.

חלוקה מומלצת לפונקציות

<u>צרו פונקציה שיוצרת מפתחות RSA (רק לשרת)</u>

מומלץ לקרוא שוב את שלבי יצירת המפתחות באלגוריתם RSA.

א. בחרו מספרים ראשוניים P, Q שהמכפלה שלהם P*Q גדולה מ-2 בחזקת 16. ניתן לחפש בגוגל מספרים ראשוניים בטווח רצוי. הסיבה לכך תתברר בקרוב: המכפלה תשמש אותנו בפעולת מודולו, ונרצה שהשארית שנקבל תהיה בתחום של 16 ביט.

ב. בחרו מפתח ציבורי.

ג. חשבו את המפתח הפרטי המתאים למפתח הציבורי שבחרתם – המפתח הציבורי כפול המפתח הפרטי מודולו ה-Totient צריך להיות שווה 1.

כעת ברשותכם מפתח ציבורי E, מפתח פרטי D ומודולו – P*Q, הקרוי N.

הפונקציה הזו תופעל רק בשרת. הלקוח מקבל את N ואת E מהשרת.

<u>צרו פונקציה שיוצרת סוד משותף בין הצדדים (קוד שונה לשרת וללקוח)</u>

קביעת המפתח, הסוד המשותף, תתבצע באמצעות אלגוריתם Diffie Hellman. הסוד המשותף צריך להיות בגודל 16 ביטים, ובקרוב יתברר מדוע.

כמו כן, כפי שהוסבר בפתיח, צד הלקוח מבצע אימות לשרת, באמצעות זה שהשרת חותם עם מפתח RSA פרטי על הערך שהוא שולח ללקוח.

שלב א: בחרו שני מספרים p, g הקטנים מ-65535 (הערך המקסימלי של 16 ביט). p צריך להיות ראשוני. לטובת התרגיל בחרו g כלשהו בלי להתחשב בתנאים של האלגוריתם (התנאי באלגוריתם הוא ש-g צריך להיות primitive root של p).

שלב ב: כל צד יבחר ערך סודי.

שלב ג: הלקוח מחשב את ערך ה-Diffie Hellman הציבורי ושולח לשרת.

שלב ד: השרת מחשב את הסוד המשותף באמצעות הערך הסודי שלו והערך הציבורי של הלקוח.

שלב ה: השרת מחשב את ערך ה-Diffie Hellman הציבורי, אך עדיין לא שולח אותו.

שלב ו: השרת חותם על הערך שחישב באמצעות מפתח ה-RSA הפרטי שלו.

ניתן להשתמש בפונקציה pow, לדוגמה:

```
signature = pow(server_public_DH_num, server_private_RSA_key, N)
```

השרת שולח את התוצאה ללקוח.

שלב ז: הלקוח משתמש במפתח RSA הציבורי של השרת, הידוע לו מראש, כדי לפתוח את החתימה של השרת ולחלץ את הערך הציבורי של השרת.

```
Server_public_DH_num = pow(signature, server_public_RSA_key, N)
```

שלב ח: כעת הלקוח יכול לחשב את הסוד המשותף באמצעות הערך הסודי שלו והערך הציבורי של השרת.

אם ביצעתם את השלבים נכון, השרת והלקוח הגיעו לאותו סוד משותף. הדפיסו אותו לטובת חיווי למשתמש.

צרו פונקציה שיוצרת מפתח הצפנה ו-Hashing key (קוד זהה לשרת וללקוח)

חלקו את הסוד המשותף בן 16 הביט לשני חלקים:

8 ביט ראשונים יהיו מפתח הצפנה סימטרי

8 ביט אחרונים יהיו Hashing key

צרו פונקציית הצפנה/פענוח סימטרית (קוד זהה לשרת וללקוח)

הפונקציה תקבל קלט ומפתח הצפנה סימטרי בגודל 8 ביט ותחזיר XOR ביניהם.

צרו פונקציית Hash (קוד זהה לשרת וללקוח)

כיתבו פונקציית Hash כיד הדמיון הטובה עליכן. הפונקציה תיקח את המסר המוצפן כולו, תבצע עליו פעולות מתמטיות שונות ותהפוך אותו למספר בן 16 ביט.

לחילופין אפשר לייבא פונקציית Hash כגון SHA או MD5.

צרו פונקציית HMAC (קוד זהה לשרת וללקוח)

כיתבו פונקציה שמחשבת HMAC לכל מסר. ראשית, ה-Hashing key יחובר למסר. על החיבור יחושב Hash. התוצאה היא HMAC.

כעת הפעילו את הפונקציות לפי הסדר המתאים, כך שיבצעו את הפעולות המתוארות בפתיחה – "שלבי התקשורת על גבי הסוקט".

פרק 19: סרטיפיקטים

את הפרק הקודם סיימנו עם בעיה. נראה, שלא משנה מה עשינו, לא הצלחנו לפתור לגמרי את בעיית אימות הזהות, Authentication.

תחילה ניסינו לפתור את הבעיה באמצעות HMAC. שילבנו ב-Hash מפתח, שתואם בין שני הצדדים לתקשורת. ראינו כי זהו כלי שמאפשר לנו לבטוח בכך שהודעה שהגיעה אלינו, נשלחה על-ידי הצד השני לתקשורת וגם הגיעה ללא שינוי. מצד שני, מי מבטיח שהצד השני לתקשורת הוא מי שאנחנו חושבים שהוא? אם החלפנו את מפתח האימות עם גורם זדוני, אז הגורם הזה יוכל להמשיך לשטות בנו לכל אורך ההתקשרות.

החתימה הדיגיטלית עבדה בגישה אחרת – נצפין את ה-Hash באמצעות מפתח א-סימטרי, פרטי. גישה זו מקדמת אותנו אך אינה פותרת לגמרי את הבעיה. מצד אחד, אם יש לנו מפתח ציבורי של מישהו, אנחנו יכולים לבטוח בכך שהודעה שקיבלנו נשלחה על-ידו. מצד שני, האמון שלנו מבוסס על כך שהמפתח הציבורי שיש לנו אכן שייך למי שאנחנו חושבים שהוא שייך לו.

איך נפתור את בעיית האמון? איך בוב יכול לדעת שכאשר הוא יוצר קשר עם השרת של אליס שלנו, או עם כל שרת אחר, הוא אכן מי שהוא נחזה להיות, ולא מתחזה?

כדי לפתור את הבעיה נציג שני רכיבים חדשים בהתקשרות: סרטיפיקט, ומודל בשם PKI.

Certificate

הסרטיפיקט הוא בבסיסו, לפני שניכנס לפרטים, הודעה פשוטה. שרת שולח ללקוח את הפרטים הבאים: הנה שם הדומיין שאני מייצג, הנה המפתח הציבורי שלו והנה המודולוס שלו.

נזכיר כי הבסיס של Authentication הוא חתימה דיגיטלית. שרת חותם על הודעה עם מפתח פרטי, והלקוח מוודא שהחתימה נכונה באמצעות המפתח הציבורי של השרת, מפתח אשר אמור להיות ידוע לכולם. כדי שהלקוח יוכל לבדוק את החתימה הדיגיטלית, הוא חייב לדעת מה המפתח הציבורי של השרת וגם מהי מכפלת המספרים הראשוניים שבחר השרת. המכפלה נקראת מודולוס, כיוון שאיתה מבצעים את פעולות המודולו. זו הסיבה שהקדשנו זמן ללימוד הרעיון של אלגוריתם RSA וכיצד עובדת חתימה דיגיטלית.

אם כך, סרטיפיקט חייב לכלול לכל הפחות את הפרטים הללו. כמו בדוגמה הבאה, הממחישה סרטיפיקט שאליס מעבירה לבוב כדי שיוכל לוודא שהיא באמת היא ולא מתחזה:

כפי שאפשר לראות, הוא כולל את שם הדומיין של אליס, את המפתח הציבורי ואת המודולוס. הסרטיפיקט חתום על-ידי דבר מה מסתורי המתכנה Certificate Authority, ונדון בו כמובן בהמשך הפרק.

אך לפני כן, בואו נצפה בפרטים הללו בסרטיפיקט אמיתי!

תרגיל מודרך: צפיה בסרטיפיקט – WSL

אנחנו הולכים לפנות לשרת, לבקש ממנו את הסרטיפיקט שלו, ולהציג אותו. נעשה זאת תוך שימוש בחבילת תוכנה שפותחה לטובת זה, openssl.

ראשית, התקינו WSL, Windows Subsystem for Linux. זוהי תוכנה של מיקרוסופט שתאפשר לכם להריץ פקודות לינוקס.

https://learn.microsoft.com/en-us/windows/wsl/install

כעת התקינו את openssl:

```
sudo apt install openssl
```

אנחנו הולכים לבקש את הסרטיפיקט של cyber.org.il. הריצו את הפקודה הבאה, פקודת bash, שתוסבר מיד לאחר שנסקור את הסרטיפיקט:

openssl s_client -connect cyber.org.il:443 | openssl x509 -text -noout

```
barak@DESKTOP-91SIDPQ: ~
barak@DESKTOP-91SIDPQ:~$ openssl s_client -connect cyber.org.il:443 | openssl x509 -text -noout
depth=2 C = US, O = Internet Security Research Group, CN = ISRG Root X1
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = R11
verify return:1
depth=0 CN = cyber.org.il
verify return:1
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            05:9a:10:46:a4:42:1a:a5:53:ec:6d:db:f6:88:60:3f:0a:f3
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Let's Encrypt, CN = R11
        Validity
            Not Before: Jun 30 04:19:32 2025 GMT
            Not After : Sep 28 04:19:31 2025 GMT
        Subject: CN = cyber.org.il
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:c1:f0:80:03:4e:be:77:d4:f2:f1:f4:35:d2:cd:
                    b8:03:67:92:7e:51:da:0f:81:46:3a:42:b7:ff:41:
                    f4:7a:18:ab:0e:0f:1c:ff:a3:c2:23:6d:e4:dc:68:
                    07:1f:ea:ca:7e:f2:29:53:60:88:d2:dc:c1:15:d4:
                    87:3c:a0:c9:14:c0:d1:87:3f:65:5c:bd:93:bb:a7:
                    2c:30:49:df:3c:7d:6a:93:34:bf:46:32:48:db:a2:
                    ec:d5:d5:76:ac:40:80:30:0d:77:97:bc:39:a1:02:
                    79:7a:f7:ac:ac:5f:17:4e:c4:6b:e2:0c:f7:68:51:
                    6a:89:b4:e4:df:63:65:09:7f:6a:48:b1:a0:a9:95:
                    c8:e8:50:41:c0:ff:85:c4:b2:14:6b:98:2a:b4:ed:
                    c1:00:78:62:51:a0:98:ad:8b:b8:18:a2:73:d8:a6:
                    a6:b9:2c:02:8d:0a:42:4a:5a:99:be:b0:7f:8f:f4:
                    72:21:0a:cf:d5:c1:69:59:a9:14:e1:67:4a:ce:c3:
                    f8:99:1d:9d:57:4f:f3:26:45:3f:43:6d:52:e9:9f:
                    a1:e2:8b:85:a6:bb:85:35:9f:eb:56:a9:4b:af:34:
                    f3:47:d5:ae:fb:28:8c:72:72:2e:c4:a6:49:b2:3a:
                    7a:58:93:be:6f:6d:9b:75:fa:c2:07:c0:11:5d:36:
                    31:90:72:5c:4a:6c:16:e6:28:37:66:82:ca:0f:a9:
                    06:78:ce:8b:76:3c:38:e0:65:a7:bf:4e:00:6a:f4:
                    03:52:9c:e5:d3:f5:24:31:43:08:26:bd:62:bd:41:
                    89:ee:03:1e:dc:fe:2a:4b:a8:bd:6d:3c:a0:fd:ac:
                    d7:27:b7:9f:fb:e5:ff:9b:f2:10:dd:b3:bb:01:e5:
                    d7:09:42:c8:f2:15:f2:ef:b5:07:fa:ea:48:93:2c:
                    2d:57:2f:be:9a:89:4a:0c:26:65:4b:d1:f8:2f:1a:
                    d9:50:4c:9a:0a:6b:a0:3d:fc:6b:94:dc:20:55:95:
                    da:f9:0e:3f:59:81:76:39:a4:66:03:be:fe:93:a2:
                    53:d4:ef:91:fb:34:1e:2b:26:4b:48:b9:fd:6b:aa:
                    78:04:ca:1f:5b:e1:18:54:e1:3b:4e:6c:be:cb:fc:
                    4b:2b:f9:20:53:98:cb:87:69:a8:47:49:84:ad:3e:
                    92:26:e0:b3:55:06:63:3d:2e:c7:9f:ea:61:7e:56:
                    82:5c:01:32:3b:c1:b2:27:06:23:ab:15:76:97:c5:
                    1f:f1:2c:e5:13:68:ca:b3:8f:58:db:ad:31:79:fa:
                    24:af:79:f2:d9:07:b7:2b:07:18:71:a2:cf:2f:ea:
                    e7:8d:32:46:79:4b:b7:e3:21:93:b8:ab:31:12:e4:
                    de:a1:c3
                Exponent: 65537 (0x10001)
```

הידד, הורדתם את הסרטיפיקט. יש בסרטיפיקט שדות רבים, שלא על כולם נעבור כעת. נעבור על השדות המודגשים. שלושה מבין השדות הללו אנחנו כבר מכירים:

- שדה ה-CN, קיצור של Canonical Name – זהו שם הדומיין

- שדה ה-Modulus, כאמור המכפלה של הראשוניים P, Q

- שדה המפתח הציבורי, ה-Exponent

השדות הנוספים שמסומנים:

- שדה ה-Validity קובע בין אילו תאריכים הסרטיפיקט בתוקף. לכל סרטיפיקט יש תאריך פג תוקף. בדומה לתאריך תפוגה של רישיון נהיגה או של כל תעודה אחרת, יש צורך בבדיקות מסויימות לטובת חידוש סרטיפיקט. אך מדוע יש תאריך תחילת תוקף? לכל שרת צריך להיות בכל זמן נתון סרטיפיקט אחד בתוקף. בעלים של דומיין בדרך כלל יבקשו סרטיפיקט זמן מה לפני פקיעת התוקף של הסרטיפיקט הנוכחי, ותאריך תחילת התוקף מונע מצב שלשרת יש יותר מסרטיפיקט אחד, או ששרת חדש מופעל לפני המועד שבו הוא היה אמור לפעול.

- שדה ה-Serial Number. לכל סרטיפיקט יש מספר סידורי ייחודי. המספר הזה מאפשר, כפי שנראה בהמשך, לבדוק אם הסרטיפיקט בתוקף. רגע אחד, מדוע צריך לבדוק אם סרטיפיקט בתוקף, אם כבר יש לנו את שדה ה-Validity? יכולים להיות מצבים שבהם סרטיפיקט מבוטל. חישבו לדוגמה על בעל רישיון נהיגה שביצע עבירת תנועה ורישיונו נשלל. הוא עדיין מחזיק ברישיון עם תאריך תוקף, אבל כאשר שוטר בודק את רישיון הנהיגה שלו הוא יכול לראות שהרישיון למעשה בשלילה.

כעת, נפענח את פקודת הלינוקס. אלה הן למעשה שתי פקודות, שמשורשרות על-ידי pipe. הסימן "|" אומר שהתוצר של פקודה אחת מועבר לפקודה שניה.

```
openssl s_client -connect cyber.org.il:443 | openssl x509 -text -noout
```

- Openssl – זוהי חבילת תוכנה שכוללת פקודות לעבודה עם הצפנות, פרוטוקול TLS שנלמד בהמשך וכן סרטיפיקטים.

- S_client – פקודה שיוצרת לקוח שיודע לקיים תקשורת מוצפנת עם שרת.

- הפרמטר connect מקבל שם דומיין ופורט להתחבר אליהם, במקרה שלנו בחרנו בדומיין cyber.org.il, ובפורט 443 שהוא הפורט המוכר של גלישה מאובטחת HTTPS.

התוצאה של הרצת הפקודה עד כאן, היא שהשרת שולח לנו את הסרטיפיקט שלו. כעת צריך לפענח אותו.

- פורמט x509 הוא הפורמט של סרטיפיקטים.

- הפרמטר text קובע שההדפסה למסך תהיה בפורמט ניתן לקריאה אנושית, ולא base64.
- הפרמטר noout קובע שההדפסה לא תכלול את פורמט ה-base64 (אחרת יודפס גם הטקסט הקריא וגם המקור בפורמט base64).

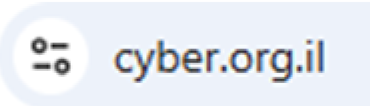
יפה. כעת ביכולתכם להוריד את הסרטיפיקט של כל דומיין שברצונכם, באמצעות סקריפט שיעשה זאת למענכם.

תרגיל: סרטיפיקטים bash

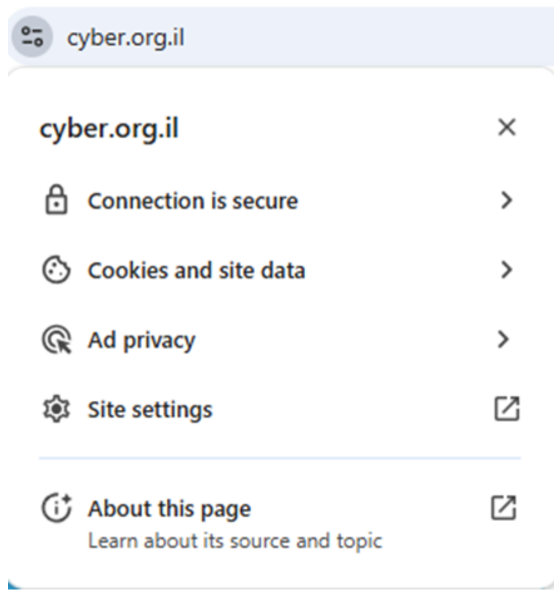
כיתבו סקריפט שמקבל מספר שמות דומיין ושומר לקובץ את ה-serial keys שלהם

תרגיל מודרך – צפיה בסרטיפיקט בדפדפן

בדפדפן הקלידו cyber.org.il. לצד הכתוב, תוכלו לראות סמל של קישור מאובטח:

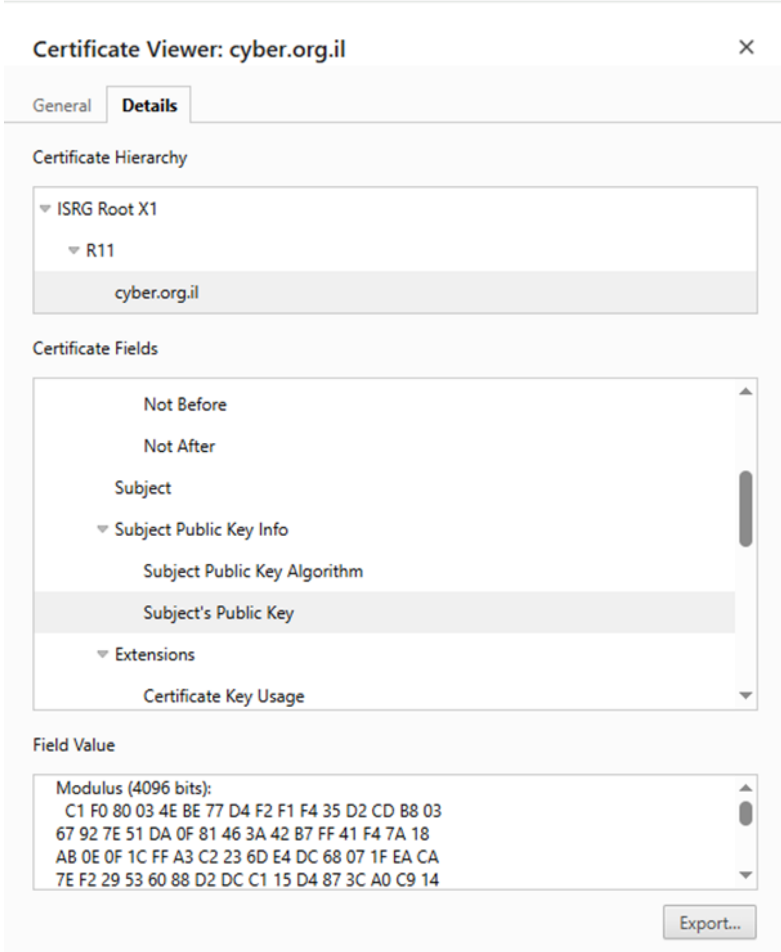


הקליקו על הסמל, ומבין רשימת האפשרויות בחרו Connection Is Secure



במסך הבא, בחרו את אפשרות הצפיה בסרטיפיקט.

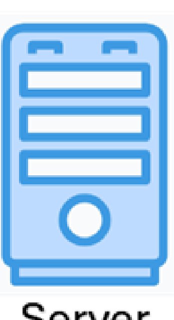
כעת תחת טאב details, תוכלו לצפות בפירוט השדות השונים. אם תגללו ל-Subject's Public Key תוכלו לקרוא בחלונית התחתונה את הפרטים, המודולוס והמפתח הציבורי.



תוכלו לוודא שהערכים הם אותם ערכים שהורדתם באמצעות WSL.

Certificate Authority

כאשר סקרנו את הסרטיפיקט של אליס, ציינו בהערה קטנה, שעל הסרטיפיקט חתום גורם המתכנה Certificate Authority, או בקיצור CA.



ה-CA הוא שרת. שרת שמהווה עוגן של אמון בין השרת והלקוח. דמיינו שאתם חלק מחוג כלשהו או קבוצת חברים. אדם שאינכם מכירים מבקש להצטרף לקבוצה. אתם שואלים חבר שלכם מה דעתו על הבחור החדש והוא אומר "כן, הוא בסדר גמור". החבר שלכם שימש עוגן של אמון. כיוון שאתם בוטחים בחבר שלכם, אתם נותנים אמון גם בבחור החדש.

באופן דומה, בכל גלישה באינטרנט, הלקוח צריך עוגן של אמון שיאפשר לו לדעת שניתן לבטוח בשרת שהוא יוצר איתו קשר. אם אתם גולשים לבנק כלשהו, לדוגמה, אתם זקוקים ל-CA שיאשר לכם "כן, זהו השרת של הבנק ולא שרת מתחזה כלשהו".

לפני שנסביר איך עובד שרת CA, נסקור את שרתי ה-CA הקיימים. השרתים מופעלים על-ידי ארגונים שונים. השרת המוביל בזמן כתיבת שורות אלה הוא של Let's Encrypt, ארגון ללא מטרת רווח המופעל על-ידי ה- Internet Security Research Group. בנוסף לו, יש חברות מסחריות המציעות את שירותיהן.

האיור הבא הוא מהלינק https://w3techs.com/technologies/overview/ssl_certificate, ומציג את נפוצות ה-CAים השונים. אפשר לראות שכמעט כל האתרים בעולם משתמשים בשירותיהם חמישה CAים. האתר מתעדכן ומציג תמונת מצב מעודכנת, כך שמומלץ להיכנס ולצפות בעצמכם.

	absolute usage percentage	market share
Let's Encrypt	59.8%	63.4%
GlobalSign	21.8%	23.1%
Sectigo	6.3%	6.6%
GoDaddy Group	3.8%	4.1%
DigiCert Group	2.1%	2.2%
Actalis	0.6%	0.7%
Certum	0.6%	0.6%
Secom Trust	0.3%	0.3%
SSL.com	0.1%	0.1%

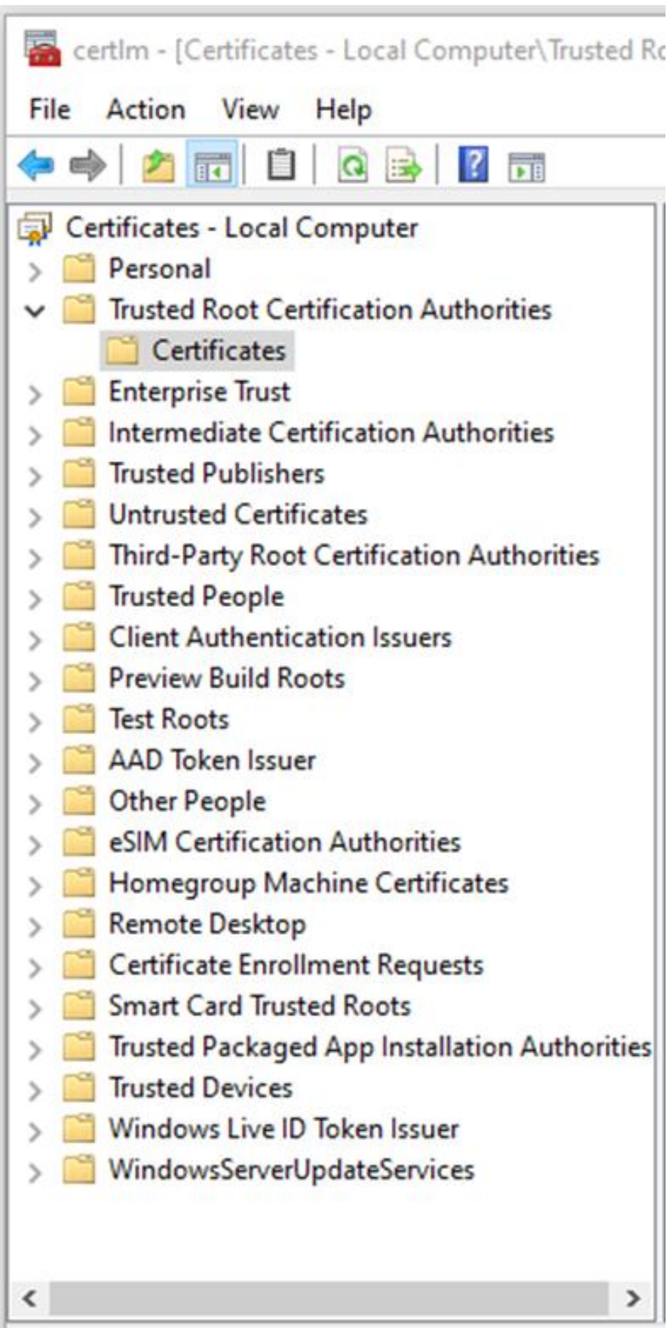
W3Techs.com, 6 September 2025

ה-CAים הללו אינם סתם CAים, אלא נקראים Root CA. כלומר, הם השורש, או העוגן, של כל המשך התהליך. מה הופך CA ל-Root CA? מיד נפענח את זה בעצמנו.

תרגיל מודרך: צפיה ב-Root CA

באמצעות שורת החיפוש של windows, הכנסו ל-certmgr.msc. זהו מנהל הסרטיפיקטים של המחשב שלכם, שמרכז את כל הסרטיפיקטים הידועים למחשב.

תחת Trusted Root Certification Authorities, תחת Certificates, תוכלו למצוא את רשימת כל ה-Root CA המוכרים למחשב שלכם.



א. חפשו בין ה-Root CA את ה-CAים הנפוצים. שימו לב שהשם המוכר נמצא תחת העמודה Friendly Name. כך לדוגמה תמצאו שהשם של "AAA Certificate Services" הוא בעצם Sectigo, הנמצאת באיור האחרון.

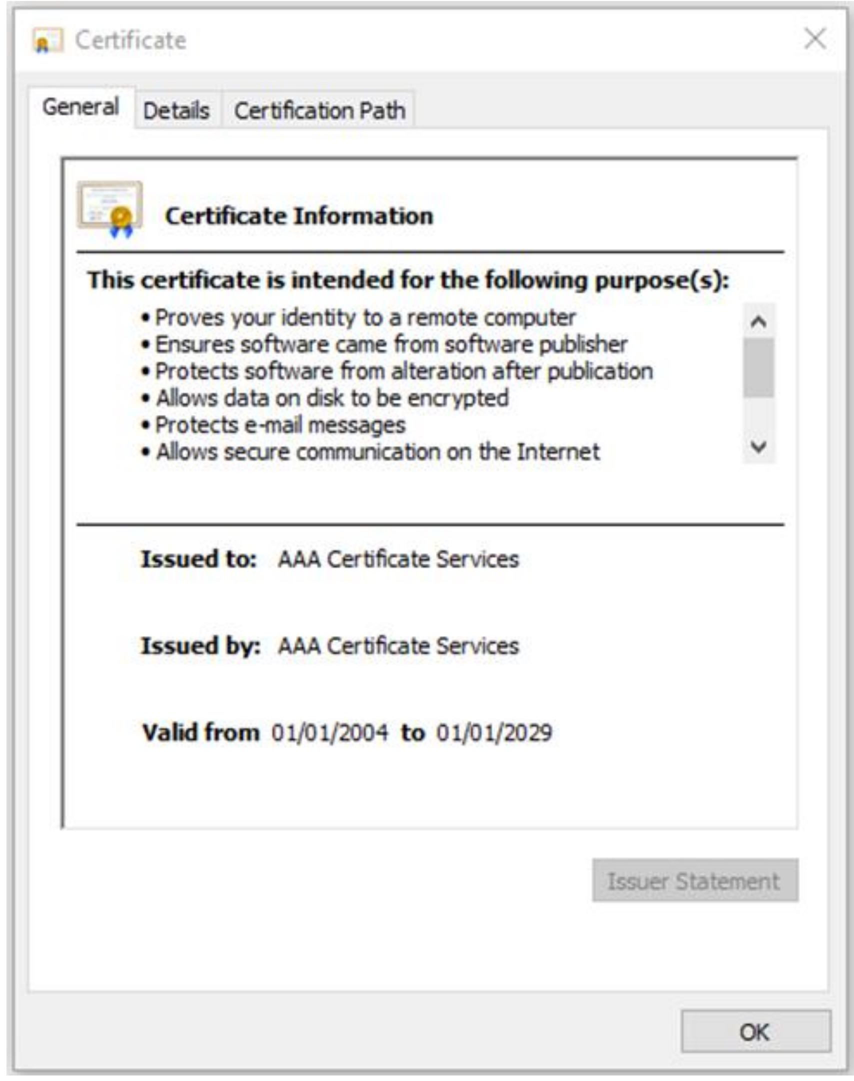
ב. פיתחו את הסרטיפיקט של Let's Encrypt ומיצאו את המפתח הציבורי שלו.

חתימה על Root Certificate

כאשר התחלנו את הדיון על סרטיפיקטים, הזכרנו שסרטיפיקט דורש שמישהו יהיה חתום עליו. מה שמעלה את השאלה, מי חותם על סרטיפיקטים של Root CA?

חיזרו ל-certmgr.msc, הקליקו על סרטיפיקט של Root CA כלשהו. מי חתום עליו?

כן, כפי שנוכחתם זה עתה, מי שחתום על סרטיפיקט של Root CA הוא – הוא עצמו. לדוגמה Sectigo חותמת על הסרטיפיקט של Sectigo.



אם כן, הסיבה ש-Root CA נקרא Root, היא שהוא חותם לעצמו על הסרטיפיקט.

איך הסרטיפיקטים של ה-Root CAים הגיעו אל המחשב שלכם? כאשר לקוח מתקין דפדפן, הדפדפן כולל מספר Root CA Certificates. אם תבחנו את התוקף של הסרטיפיקטים של Root CA, דבר שניתן לבדיקה בקלות באמצעות ה-certmgr.msc, תגלו שהתוקף שלהם הוא שנים רבות קדימה. זאת מכיוון שההתקנה שלהם במחשב אמורה להיות ארוע לא שכיח – פעם בהתקנת דפדפן חדש.

תרגיל מודרך: יצירת Root CA Certificate

בתרגיל זה ניצור Certificate Authority ונכניס את הסרטיפיקט שלו ל-certmgr.msc, כך שהמחשב יכיר בו כבסיס לחתימות. שלבי העבודה:

א. יצירת זוג מפתחות ל-CA שלנו

ב. ה-CA יחתום לעצמו על סרטיפיקט

ג. נייבא את הסרטיפיקט לתוך certmgr

צרו מפתח ל-CA:

```
openssl genrsa    -out   ca.key   4096
```

כדי להתבונן במפתח:

```
cat ca.key
```

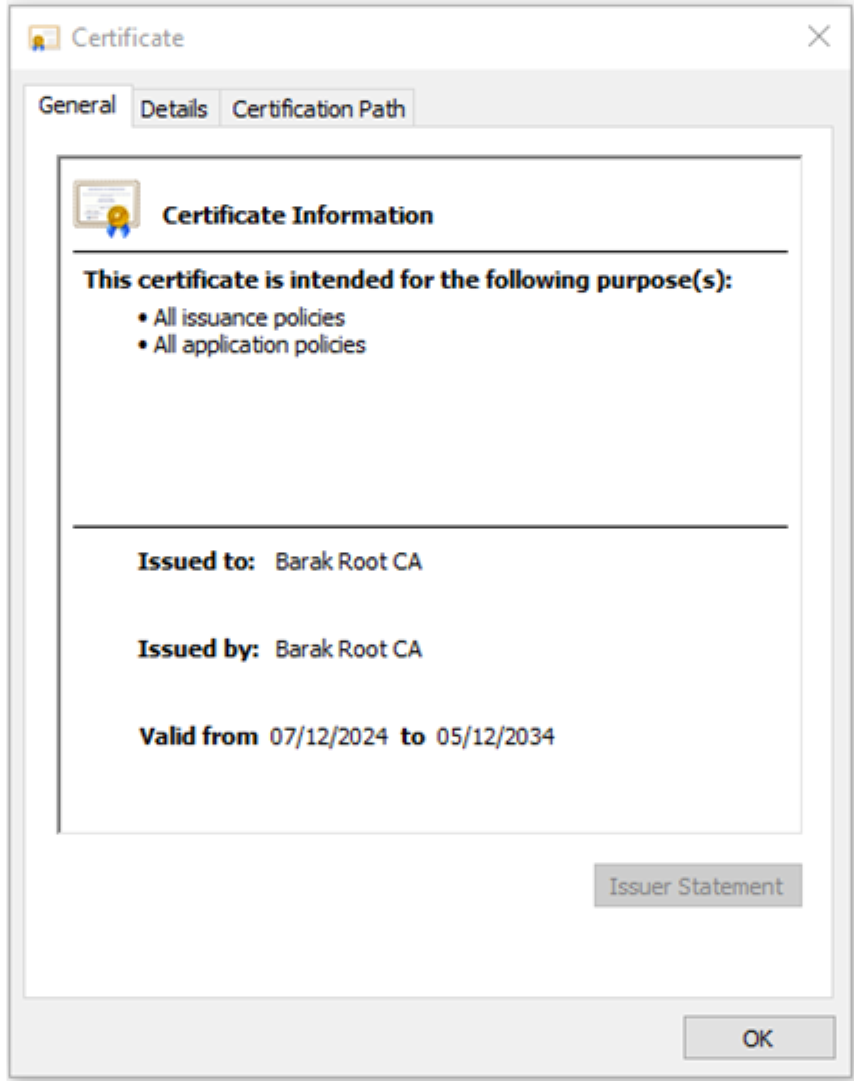
מה שנקבל מטעה מעט – נראה שיש לנו רק מפתח פרטי. למעשה, המידע במפתח הפרטי כולל גם את המודולוס (P*Q) ואת P, Q עצמם. כזכור, אם יש לנו אותם, אפשר לחשב את ה-public key. הפקודה הבאה תציג את כל המידע כולל חישוב המפתח הציבורי:

```
openssl rsa     -in   ca.key   -text   -noout
```

צרו סרטיפיקט חתום על-ידי ה-CA לעצמו:

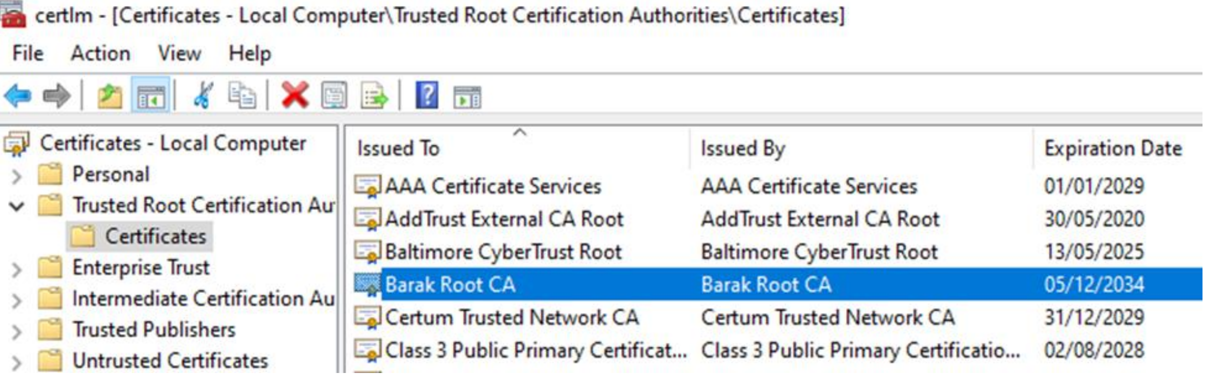
```
openssl req   -new   -x509   -days 3650   -sha256   -key ca.key   -out ca.crt
```

בתהליך היצירה של הסרטיפיקט תתבקשו להזין נתונים שונים. למעט ה-Common Name, ניתן לדלג על כולם, באמצעות לחיצה על enter. באשר ל-Common Name – קיראו לו Root CA, ואפשר להוסיף לו את השם שלכם. אם תתבקשו להזין Challenge Password הזינו סיסמה כלשהי.



סרטיפיקט חתום של Root CA שיוצר לטובת התרגיל

כדי לייבא את הסרטיפיקט אל ה-Certmgr, הקליקו על קובץ ה-crt ועיקבו אחרי ההוראות.



CSR – Certificate Signing Request

כעת נתאר איך דומיין משיג סרטיפיקט.

בעלים של דומיין כלשהו, לדוגמה alice.co.il, מבקשים לקבל סרטיפיקט המאשר שהם הבעלים של הדומיין. במקרה זה אליס, בעלת הדומיין, קודם כל תיצור צמד מפתחות – פרטי וציבורי.

כעת, אליס תפנה ל-CA כלשהו ותשלח לו CSR – בקשת חתימה על סרטיפיקט. ה-CSR הוא פשוט למדי וכולל שלושה שדות:

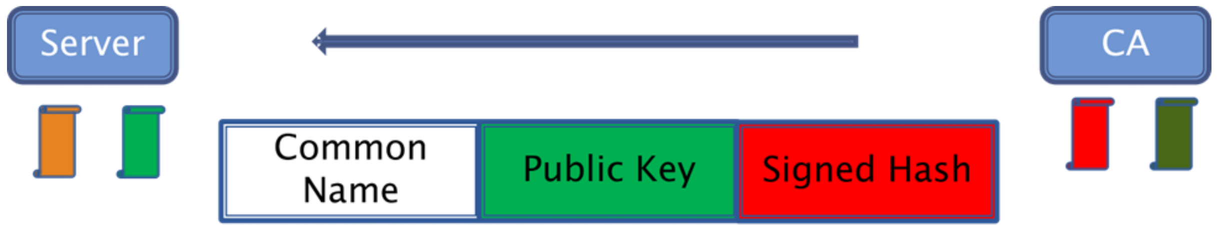
א. שם הדומיין

ב. המפתח הציבורי

ג. Hash חתום (על-ידי המפתח הפרטי של אליס)

ה-CA יחשב בעצמו את ה-Hash על ה-CSR, וישווה אותו ל-Hash החתום. כפי שראינו, אפשר לפתוח את החתימה באמצעות המפתח הציבורי שנכלל בהודעה עצמה.

אם החישוב מתאים, ה-CA ישלח לאליס את הסרטיפיקט שלה, שייראה כך:



א. שם הדומיין של אליס

ב. המפתח הציבורי של אליס

ג. Hash חתום על-ידי המפתח הפרטי של ה-CA

החתימה על-ידי המפתח הפרטי של ה-CA היא לב כל העניין. אליס מחזיקה כעת בסרטיפיקט חתום, שאומר "לקוח יקר, אם אתה סומך על ה-CA הזה שחתום על הסרטיפיקט שלי, אז הנה אישור שאותו CA סומך עלי ומאשר שאני זו אני."

הלקוח יקבל את הסרטיפיקט החתום של אליס ויפתח את ה-Hash החתום. כיצד הוא יעשה זאת? כמובן, ראינו שהמפתחות הציבוריים של ה-Root CA נמצאים אצל הלקוח, ב-certmgr.msc.

תרגיל מודרך: יצירת סרטיפיקט חתום על-ידי ה-CA

1. לפני הכל, צריך להכין מספר קבצים בתיקיה שבה נוצרים הסרטיפיקטים.

א. צרו קובץ בשם serial (ללא סיומת). באמצעות עורך טקסט, כיתבו ערך בעל 10 ספרות הקדצימליות. לדוגמה: 0000ABCDEF.

ב. צרו קובץ ריק בשם index.txt.

ג. העתיקו לתיקייה את הקובץ CA-config.conf מהקישור:

https://data.cyber.org.il/networks/links/CA-config.con

ד. העתיקו לתיקייה את הקובץ CSR-config.conf מהקישור:

https://data.cyber.org.il/networks/links/CSR-config.conf

כעת אפשר להתחיל.

שימו לב ששם הדומיין יכול כמובן להשתנות כרצונכם. היכן שכתוב בפקודות mytestdomain.co.il הכניסו את שם הדומיין שלכם.

2. יצירת מפתח:

```
openssl genrsa    -out   mytestdomain.co.il.key      2048
```

3. יצירת Certificate Signing Request:

```
openssl req -new -sha256 -config CSR-config.conf -key mytestdomain.co.il.key -out mytestdomain.co.il.csr
```

```
barak@DESKTOP-91SIDPQ: ~
barak@DESKTOP-91SIDPQ:~$ openssl req -new -sha256 -config CSR-config.conf -key mytestdomain.co.il.key -out mytestdomain.co.il.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [IL]:
State or Province Name (full name) [Tel Aviv]:
Locality Name (eg, city) [Tel Aviv]:
Organization Name (eg, company) [mytestdomain]:
Common Name [*.mytestdomain.co.il]:
```

דלגו על כל הפרטים באמצעות enter. אם ברצונכם לשנות את שם הדומיין, פשוט עירכו את הקובץ CSR-config.conf.

צפו ב-CSR בעזרת הפקודה:

openssl req -in mytestdomain.co.il.csr -noout -text

אם לא ניתן לצפות עקב תקלה של unable to load x509 request, כיתבו:

openssl x509 -in mytestdomain.co.il.csr -noout -text

```
barak@DESKTOP-91SIDPQ:~$ openssl req -in mytestdomain.co.il.csr -noout -text
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C = IL, ST = Tel Aviv, L = Tel Aviv, O = mytestdomain, CN = *.mytestdomain.co.il
```

4. השתמשו ב-Root CA כדי ליצור סרטיפיקט חתום ל-mytestdomain.co.il:

openssl ca -config CA-config.conf -cert ca.crt -keyfile ca.key
-in mytestdomain.co.il.csr -out mytestdomain.co.il.crt

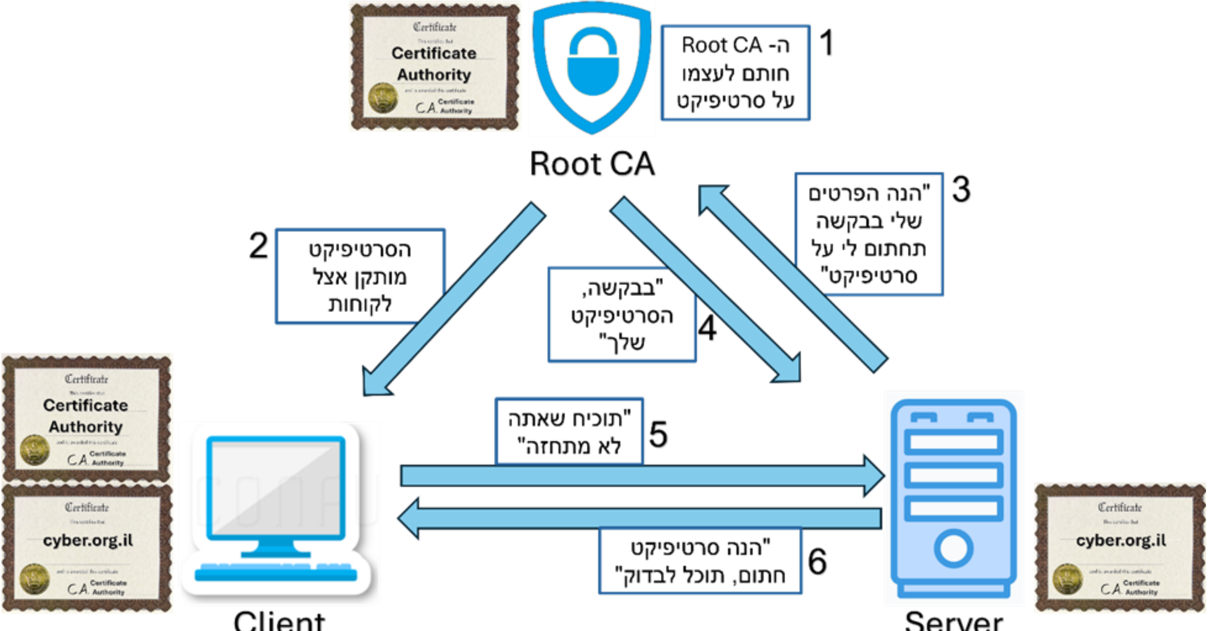
```
barak@DESKTOP-91SIDPQ: ~
Using configuration from CA-config.conf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'IL'
stateOrProvinceName   :ASN.1 12:'Tel Aviv'
localityName          :ASN.1 12:'Tel Aviv'
organizationName      :ASN.1 12:'mytestdomain'
commonName            :ASN.1 12:'*.mytestdomain.co.il'
Certificate is to be certified until Dec  8 21:04:28 2025 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

5. צפו בסרטיפיקט בעזרת הפקודה:

openssl x509 -in mytestdomain.co.il.crt -noout -text

סיכום ביניים

כסיכום ביניים, נסקור את שלבי יצירת האמון באמצעות סרטיפיקט. חשוב לציין שאלה השלבים שכיסינו עד כה, ויש שלבים נוספים, שלהם יוקדש המשך הפרק. עם זאת, לפני שהתמונה הופכת למורכבת יותר, נסכם את הפרטים שלמדנו.



השלבים הם:

1. Root CA מכין לעצמו סרטיפיקט. הוא מכין זוג מפתחות ובאמצעותו מכין סרטיפיקט, שעליו חתום הוא עצמו.
2. הסרטיפיקט מותקן אצל לקוחות שונים, מופץ באמצעות תוכנת הדפדפן.
3. דומיין מבקש סרטיפיקט חתום מה-Root CA (הערה: בהמשך נחדד את הנקודה הזו, כאשר נדון ב-Certificate Chain. כרגע, לטובת התמונה שאנחנו בונים, נניח שהפנייה היא אל ה-Root CA). בעל הדומיין מכין זוג מפתחות ושולח CSR אל ה-Root CA.
4. בעל הדומיין מקבל מה-Root CA סרטיפיקט, שחתום על-ידי המפתח הפרטי של ה-Root CA.
5. לקוח מבצע הקמת תקשורת מאובטחת מול שרת של דומיין.
6. שרת הדומיין שולח ללקוח את הסרטיפיקט שהוא קיבל, כדי שהלקוח יבדוק אותו.

השאלה המעניינת כעת היא, כיצד הלקוח בודק את הסרטיפיקט של שרת הדומיין? בכך נעסוק כעת.

Certificate Verification

לקוח שקיבל סרטיפיקט של דומיין צריך לבצע שלושה דברים:

א. לוודא שהסרטיפיקט אינו מזויף.

ב. לוודא שמי ששלח לו את הסרטיפיקט הוא אכן בעל הסרטיפיקט.

ג. לוודא שהסרטיפיקט עדיין בתוקף.

בנושא תוקף של סרטיפיקטים נדון לקראת סיום הפרק.

כיצד לקוח מוודא שסרטיפיקט אינו מזויף?

נתבונן שוב בסרטיפיקט של www.cyber.org.il. תזכורת: הריצו את הפקודה

openssl s_client -connect www.cyber.org.il:443 | openssl x509 -text -noout

בסוף הסרטיפיקט ישנם שני שדות. Signature Algorithms, Signature Value.

```
Signature Algorithm: sha256WithRSAEncryption
Signature Value:
    3d:ae:2e:f8:5c:c5:e8:13:aa:09:4b:1d:23:5b:9f:e3:88:d8:
    13:5a:16:25:2a:1f:1f:d8:1d:83:03:13:cf:07:78:be:13:ad:
    fe:47:5b:ae:9d:a9:6a:6b:5f:06:11:a4:88:40:e8:4b:70:82:
    71:61:b9:75:46:d4:d7:1f:ee:81:54:e3:a4:2c:9a:1f:f3:7c:
    78:cc:f2:36:93:b7:e4:9f:fd:e2:fa:41:84:94:61:81:9c:ab:
    b2:9a:55:00:0c:d0:28:42:d7:ba:c6:5b:0e:30:a8:26:a5:db:
    bf:a5:81:c3:ab:b4:38:df:e1:71:79:cd:05:5b:46:ed:7b:3f:
    36:ae:f0:b9:03:67:22:ad:dd:a7:30:69:6a:5f:7b:79:8d:08:
    1d:ea:d5:45:68:ff:ee:e3:c5:56:24:21:bc:f3:54:bd:88:f8:
    05:82:3e:91:66:68:48:5b:d9:f4:43:50:de:18:4e:c2:e8:ec:
    cc:0d:23:23:4c:f2:55:29:8a:0d:4f:b6:e2:1e:25:43:b4:40:
    61:8d:36:49:31:67:bc:44:6b:fd:92:25:45:8f:97:73:59:af:
    d2:92:f1:10:69:9c:60:28:7b:49:c5:f1:f9:0a:63:e0:82:fe:
    96:65:3a:6e:34:64:20:a4:86:d4:80:2a:f4:b7:d4:55:28:c8:
    84:66:07:60
```

חתימה על סרטיפיקט

הערכים של השדות הללו נקבעו על-ידי ה-CA שחתם על הסרטיפיקט. ה-CA מכריז שבוצעה חתימה דיגיטלית באמצעות SHA256 בשילוב מפתח פרטי של RSA.

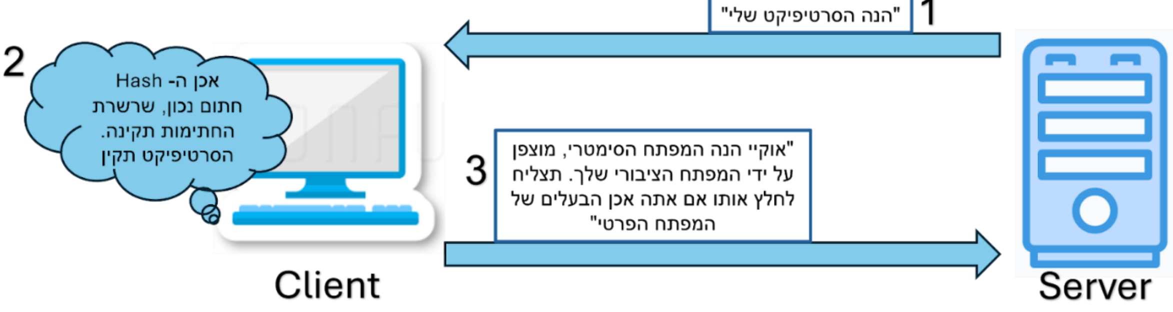
הלקוח יחשב את ה-SHA256 של הסרטיפיקט. הלקוח ישתמש במפתח הציבורי של ה-CA כדי לפתוח את החתימה הדיגיטלית ולקבל את ה-Hash שבתוכה. השוואה בין הערך שהלקוח חישב לערך שנמצא בחתימה הדיגיטלית תאפשר ללקוח לוודא כי זהו סרטיפיקט תקין. יפה.

עם זאת, הלקוח עדיין לא וידא שמי ששלח לו את הסרטיפיקט הוא בעל הסרטיפיקט. קל מאוד להשיג סרטיפיקט של כל דומיין שתחפצו בו. ביצענו זאת בעצמנו מספר פעמים. מה מונע מגורם זדוני לשלוח ללקוח תמים את הסרטיפיקט של cyber.org.il לדוגמה? הלקוח יבדוק, יגלה שמדובר בסרטיפיקט מקורי, וכל הטרחה במנגנון אותנטיקציה תהיה לחינם.

הדרך שבה לקוח מוודא שהשרת הוא אכן בעל הסרטיפיקט שהוא שלח, היא באמצעות בדיקה שלשרת יש את המפתח הפרטי הקשור למפתח הציבורי שנמצא בסרטיפיקט. יש שתי שיטות לעשות זאת, והשיטה שנבחרת תלויה באלגוריתם שהשרת והלקוח תיאמו ביניהם לצורך החלפת מפתחות. כזכור, ישנם שני אלגוריתמים המתאימים להחלפת מפתחות: RSA ו-DH.

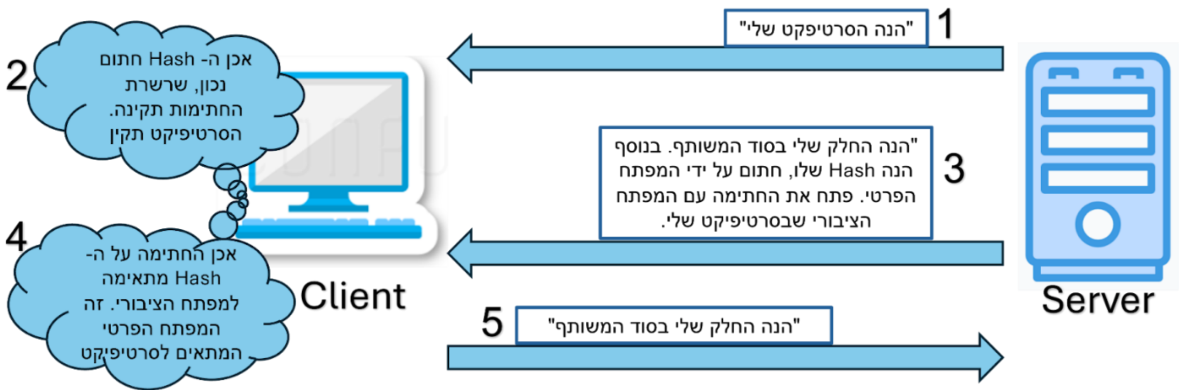
השיטה הראשונה היא שיטה שנשתמש בה אם החלפת המפתחות תבוצע באמצעות אלגוריתם RSA. בשיטה זו **הלקוח** בוחר ערך כלשהו ומצפין אותו באמצעות המפתח הציבורי של השרת, מפתח שכאמור נמצא בסרטיפיקט. הערך שהלקוח בחר אינו בדיוק מפתח ההצפנה, אבל מתוכו, בתהליך מתימטי, ייווצר מפתח ההצפנה. הלקוח אומר לשרת "ראה, אני שולח לך ערך שישמש אותך כדי ליצור את מפתח ההצפנה הסימטרי, כשהערך עצמו מוצפן באמצעות המפתח הציבורי שלך. אם אתה אכן הבעלים של המפתח הפרטי המתאים, תוכל בקלות לחלץ את הערך ששלחתי לך וליצור איתי תקשורת מוצפנת. ואם לא...".

כל מי שנמצאים בדרך בין השרת והלקוח יכולים לקלוט את הערך המוצפן ששלח הלקוח לשרת, אך הם אינם יכולים לפתוח אותו ולהשתמש בו. רק הבעלים האמיתי של המפתח הפרטי, שהסרטיפיקט שייך לו, יכול לעשות זאת.



הלקוח מאמת שהשרת הוא הבעלים של המפתח הפרטי באמצעות החלפת מפתחות בשיטת RSA

השיטה השניה היא שיטה שנשתמש בה אם החלפת המפתחות תבוצע באמצעות אלגוריתם DH. בשיטה זו, **השרת** שולח ללקוח את מפתח ה-DH הציבורי שלו (היזכרו, באלגוריתם DH כל צד בוחר מספר פרטי, ובאמצעות G ,P הוא יוצר מפתח ציבורי שהוא מעביר לצד השני). כדי להוכיח ללקוח שהוא הבעלים של הסרטיפיקט, השרת מבצע למפתח הציבורי של עצמו Hash, חותם עליו באמצעות המפתח הפרטי שלו ומעביר ללקוח. הלקוח משתמש במפתח הציבורי של השרת, מתוך הסרטיפיקט, כדי לפתוח את החתימה ולחלץ את ה-Hash. במקביל, הלקוח מחשב בעצמו את ה-Hash של המפתח הציבורי של השרת. אם הערכים מתאימים, סימן שהשרת הוא הבעלים של המפתח הפרטי שמתאים למפתח הציבורי שבסרטיפיקט, ומכאן שהשרת הוא אכן השרת של הדומיין שהוא טוען שהוא.



השרת מוכיח ללקוח שהוא הבעלים של המפתח הפרטי באמצעות החלפת מפתחות בשיטת DH

לסיכום, ראינו שהלקוח גם מוודא שהשרת אינו מנסה להונות אותו באמצעות סרטיפיקט שאינו שלו, וגם על הדרך מחליף מפתחות הצפנה עם השרת.

סוגי סרטיפיקטים

סביר שבשלב זה כבר שאלתם את עצמכם, מה עושה ה-CA לפני שהוא חותם על סרטיפיקט? הרי לא סביר שכל בקשה לחתימת סרטיפיקט, CSR, תיענה בסרטיפיקט חתום. אילו זה היה המצב, גורם זדוני היה יכול לבקש מ-CA לאשר לכם סרטיפיקט על שם בנק כלשהו, ולבצע הונאות על לקוחות הבנק.

ה-CA כמובן אינו חותם על סרטיפיקט סתם כך – הוא מבצע בדיקות. יש שלוש רמות של בדיקות, וכל רמת בדיקות מאפשרת סרטיפיקט שונה.

הרמה הבסיסית היא DV, קיצור של Domain Validation. כאשר בעלים של דומיין מבקשים סרטיפיקט DV, כל מה שהם צריכים להראות הוא שהם בעלי הדומיין. אפשר להראות את זה לדוגמה באמצעות הוספת קובץ כלשהו לשרת, בדומה לטכניקה שגוגל משתמשים בה כדי לוודא שמשתמשי google analytics הם הבעלים של הדומיין שעליו הם מבקשים נתונים. בעל הדומיין יקבל מה-CA קובץ כלשהו, ואם ה-CA יצליח להוריד את הקובץ הנ"ל מהדומיין ומהמיקום שבו הוא ביקש לשים את הקובץ, הרי שהדומיין שייך למי שביקש ממנו סרטיפיקט.

רמה כזו של בדיקה מאפשרת למנוע מצב שגורם זדוני מבקש סרטיפיקט לדומיין שאינו שלו. לדוגמה, האקר לא יוכל לבקש סרטיפיקט ל-cyber.org.il. עם זאת, הוא כן יוכל להקים דומיין בעל שם מטעה, כגון cyb3r.org.il, ולבקש עליו סרטיפיקט מסוג DV. אם ההאקר יצליח לגרום לגולשים תמימים לחשוב שזהו האתר של המרכז לחינוך סייבר, הוא יוכל להציג להם סרטיפיקט חתום ולקבל את האמון שלהם. בעיה!

הרמה השניה היא OV, קיצור של Organization Validation. רמה זו בודקת שאכן מאחרי הדומיין עומד ארגון אמיתי – ארגון שיש לו רישום במוסדות המדינה, כגון רשם החברות, רשות המיסים, רשם העמותות. ה-CA יבקש מסמכים שמאשרים זאת. כך, אם ההאקר הזדוני קנה את הדומיין שנחזה להיות של המרכז לחינוך סייבר, cyb3r.org.il, הוא לא יצליח להוכיח שמאחרי הדומיין יש ארגון אמיתי.

הרמה השלישית והגבוהה ביותר היא EV, קיצור של Extended Validation. בשביל אישור כזה, ה-CA יבקש אישורים מעורכי דין, שיעידו על כך שהעסק קיים וחוקי. ה-CA ישלח נציג שיבקר פיזית בעסק ויוודא שלא מדובר בכיסוי כלשהו או בהתחזות. הנציג אף יוודא שהאדם שביקש את הסרטיפיקט הוא עובד בעל הרשאה מתאימה, ולא לדוגמה עובד לשעבר, שמבקש לגרום נזק.

סרטיפיקט מסוג DV הוא הסרטיפיקט הפשוט ביותר, והוא ניתן חינם על-ידי Let's Encrypt, הארגון ללא כוונת רווח של ה-Internet Security Research Group. זוהי הסיבה שכפי שראינו, מרבית הסרטיפיקטים הם של ארגון זה.

סרטיפיקט מסוג OV ישמש לרוב עסקים.

סרטיפיקט מסוג EV ישמש מוסדות פיננסיים, עסקים גדולים, גופי ממשלה וכדומה.

כיצד ניתן לזהות את סוג הסרטיפיקט? מסיבה כזו או אחרת, אין לסרטיפיקט שדה שקובע מה הסוג שלו. עם זאת, יש מספר סימנים המעידים על הסוג.

ראשית, מי חתום עליו. כאמור, Let's Encrypt חותמת רק על DV. אך סרטיפיקט DV יכול להיות חתום גם על-ידי אחרים. הדרך לבדוק אם סרטיפיקט הוא DV, הוא לבדוק את שדה ה-Organization ולבדוק אם אין שם רישום. לדוגמה:

Certificate Viewer: *.shufersal.co.il
General Details
Issued To
Common Name (CN) *.shufersal.co.il
Organization (O) <Not Part Of Certificate>
Organizational Unit (OU) <Not Part Of Certificate>

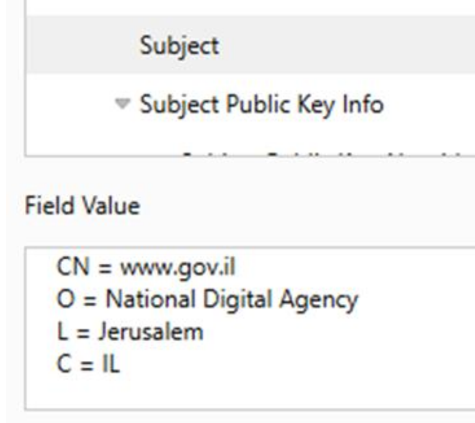
סרטיפיקט DV

לעומת זאת, סרטיפיקט מסוג OV או EV יכלול הן את שדה ה-Organization, הן פירוט על הארגון בשדה ה-Subject והן פירוט של Certificate Policies. כדוגמא לכך, האתר הממשלתי www.gov.il.

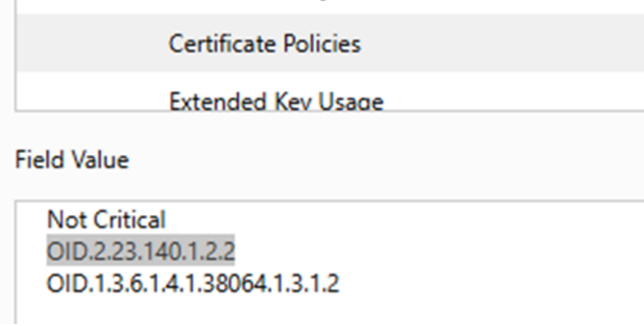
נבחן את שדה ה-Organization:

שדה ה-Organization מכיל פירוט.

נבדוק את שדה ה-Subject:



שדה ה-Subject מכיל פרטים. על סמך מה שראינו עד כה, מדובר בסרטיפקט OV או EV. כדי לקבוע במדויק, נבדוק את ה-Certificate Policies:



ה-Certificate Policies מכיל OID – Object Identifier – של התאגדות של יצרני הדפדפנים וספקי CA. המספר הספציפי שניתן, מעיד על כך שזהו סרטיפיקט OV.

לעומת זאת אם נבדוק את שדה ה-Certificate Policies של bankhapoalim.co.il נמצא כי ה-OID שלו הוא מסוג EV:

Certificate Policies

Field Value

Not Critical
OID.2.16.840.1.114412.2.1
OID.2.23.140.1.1:
Certification Practice Statement Pointer: http://www.digicert.com/CPS

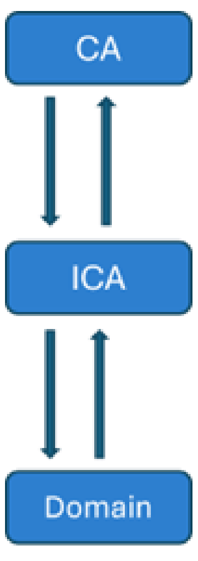
Certificate Chain

כפי שראינו, ה-Root CA הוא הבסיס של האמון בין כל שרת ולקוח, והבסיס כולו תלוי ועומד על כך שהמפתח הפרטי של ה-Root CA יישאר סודי. אם המפתח הפרטי של ה-Root CA ייוודע לגורמים אחרים, הם יוכלו לזייף בקלות את החתימה שלו ולתת סרטיפיקט לכל אתר כרצונם. התוצאה עלולה להיות קשה. לקוחות תמימים עלולים לגשת לשרת מתחזה בלי שתהיה להם כל דרך לדעת זאת, ובכך ליפול למתקפות של הונאה, גניבת כספים והתחזות.

כדי לבצע תיקון, יהיה צורך שה-Root CA יחליף את המפתחות שלו, יוציא לעצמו סרטיפיקט חדש (עד כאן תהליך לא מורכב) ואז – לעדכן את הסרטיפיקט בכל המחשבים בעולם. כיוון שזה תהליך מורכב מאין כמוהו, חייבים להגן על המפתח הפרטי של Root CA.

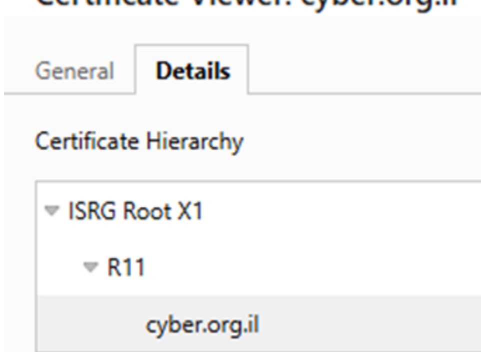
אחד העקרונות של הצפנות הוא ששימוש חוזר במפתח הצפנה מוריד את רמת האבטחה. לכן היינו רוצים שה-Root CA יחתום על סרטיפיקטים מעטים ככל האפשר. לעומת זאת, בתהליך שבו CA חותם על סרטיפיקט, הוא משתמש שוב ושוב במפתח הפרטי שלו.

הפתרון לבעיה הזו הוא להשתמש ב-Intermediate CA, כלומר CA ביניים, או בקיצור ICA. בתהליך זה, ה- Root CA חותם על סרטיפיקטים ל-ICAים. אין צורך בהרבה סרטיפיקטים, כיוון שמספר ה-ICAים אינו צריך להיות גדול. דומיינים שירצו חתימה על סרטיפיקט לא יפנו ל-Root CA אלא לאחד ה-ICAים, שיחתמו להם. השרשרת הזו, שבה CA חותם ל-CA אחר והוא שחותם לדומיין, נקראת Certificate Chain.



Certificate Chain

כדי לראות Certificate Chain לדוגמה, נתבונן בסרטיפיקט של cyber.org.il:



ננתח את מה שרואים בשרשרת זו.

- תחילה, Root CA בשם ISRG Root X1 יצר זוג מפתחות וחתם לעצמו על סרטיפיקט. הסרטיפיקט הופץ דרך הדפדפן ללקוחות בעולם.
- ICA בשם R11 יצר זוג מפתחות ושלח CSR אל ה-Root CA.
- ה-Root CA החזיר ל-ICA סרטיפיקט חתום.
- בעלי הדומיין cyber.org.il יצרו זוג מפתחות ושלחו CSR אל ה-ICA.
- ה-ICA החזיר לשרת שני סרטיפיקטים: את הסרטיפיקט החתום של cyber.org.il וכן את הסרטיפיקט שלו עצמו.

מה מתבצע כאשר לקוח פונה לשרת של cyber.org.il?

הלקוח מקבל שני סרטיפיקטים חתומים. גם את זה של cyber.org.il וגם את זה של R11. הלקוח מבצע תהליך אימות לסרטיפיקטים. הוא מוודא ש-cyber.org.il חתום כהלכה על-ידי R11. לאחר מכן, הוא מוודא ש-R11 חתום כהלכה על-ידי ISRG ROOT X1, שמותקן אצלו ב-Certmgr.msc.

הידד!

או שאולי מוקדם מדי לשמוח? האם תצליחו למצוא שיטה לנצל את התהליך בצורה שתאפשר לדומיין זדוני להשיג סרטיפיקט חתום?

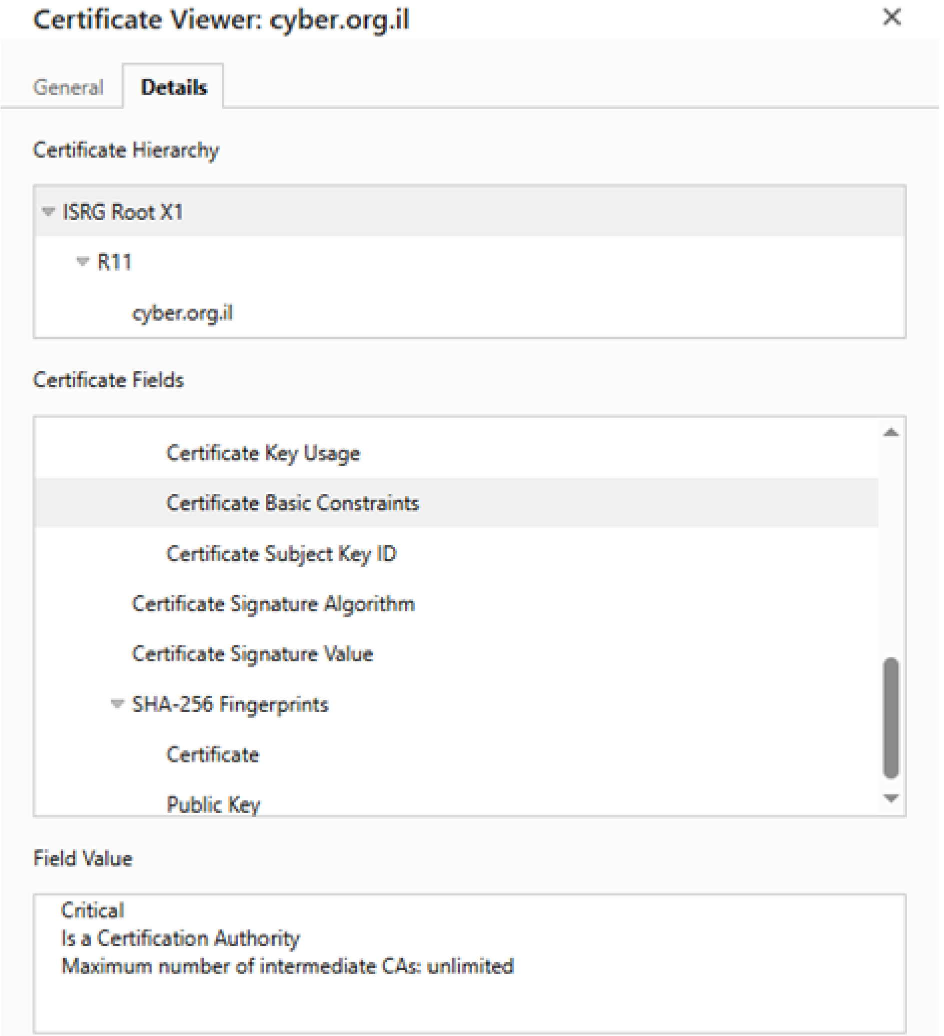
Basic Constraints

ובכן, אם כל מי שיש בידו סרטיפיקט חתום כהלכה יכול לחתום על סרטיפיקטים אחרים, זה פותח פתח לניצול. גורם זדוני יכול להקים שני דומיינים. הראשון דומיין "כשר" ותמים, השני דומיין זדוני, שדרכו יבוצעו הונאות. הדומיין הכשר מקבל סרטיפיקט. כעת, משיש בידו סרטיפיקט תקין, אפשר לחתום על הסרטיפיקט של הדומיין הזדוני באמצעות המפתח הציבורי של הדומיין הכשר וליצור Certificate Chain שתחילתו ב-Root CA וסופו בדומיין זדוני.

אם כך, יש צורך להגביל את היכולת של מי שמקבל סרטיפיקט לחתום לאחרים על סרטיפיקטים. ה-Root CA היה רוצה לומר משהו כגון "הנה ICA, אני מאפשר לך לחתום על סרטיפיקטים לאחרים, אבל אינך רשאי להוריש את התכונה הזו הלאה. לא יכול להיות שמישהו שמקבל ממך סרטיפיקט יורשה להשתמש בו לחתימות נוספות" .

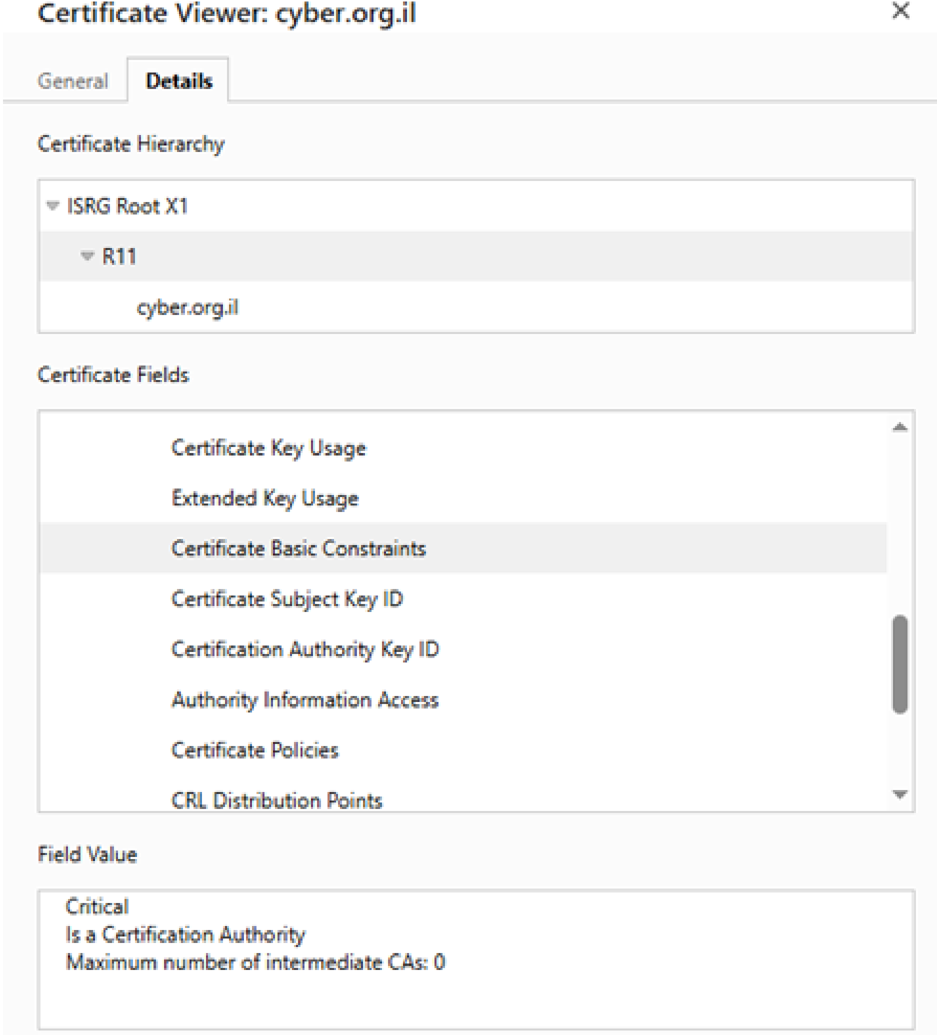
המימוש של הדרישה הזו מתבצע באמצעות שני ערכים שנמצאים בסרטיפיקט, תחת Certificate Basic Constraints. הערך הראשון מגדיר אם בעל הסרטיפיקט הוא CA. אם הוא אכן CA, השדה השני מגדיר כמה דרגות של ICAים יכולות להיות לו.

נתבונן בסרטיפיקט של ה-ISRG Root CA:



כפי שרואים, ה-Root CA חתם לעצמו על סרטיפיקט שמאפשר לו להיות CA, וכן לא הגביל את עצמו בכמות הדרגות של CAים שיכולים להימצא תחתיו. תאורטית, הוא יכול לחתום ל-ICA, שיחתום ל-ICA, שיחתום ל-ICA וכך הלאה.

נתבונן בשדות הנ"ל אצל ה-ICA, שנקרא במקרה הזה R11:



הסרטיפיקט שה-Root CA העניק ל-ICA מאפשר לו להיות CA כמובן, אך נותן לו אפשרות לאפס דרגות של ICAים מתחתיו. כלומר, ה-ICA אינו יכול ליצור ICAים נוספים, אלא רק לחתום על סרטיפיקטים לדומיינים.

איך צפויים להראות השדות הללו בסרטיפיקט של cyber.org.il ? בידקו ותוודאו שאתם צודקים.

Certificate Revocation

האם יצא לכם אי פעם לבטל תעודה למרות שהיא היתה בתוקף? בדרך כלל, מדובר בארוע חריג, אבל דברים חריגים קורים. לדוגמה, רישיון נהיגה עלול להתבטל כי הנהג עבר עבירת תנועה חמורה, או שמסיבות רפואיות אינו

יכול לנהוג יותר. תעודת זהות עלולה להתבטל עקב שינוי אזרחות. דברים מתרחשים בחיים האמיתיים, וגם סרטיפיקט חתום ובעל תוקף עלול – במקרים חריגים – להתבטל.

מקרים חריגים שיובילו לביטול סרטיפיקט הם לדוגמה כשנחשף המפתח הפרטי של בעל הסרטיפיקט, או כשנחשף המפתח הפרטי של ה-CA שחתם עליו (ולכן נדרש שה-CA יחליף מפתח, מה שיוביל לביטול כל הסרטיפיקטים שהוא חתם עליהם), או כשהדומיין אינו פעיל יותר.

האירוע המפורסם ביותר הוא הפריצה לחברת הסרטיפיקטים DigiNotar. מקרה זה היה כה קיצוני עד שהוביל לפיתוח טכנולוגיה חדשה הקרויה Certificate Transparency. על אירוע הפריצה ועל הטכנולוגיה החדשה נדון בהמשך הפרק.

אנחנו מבינים שעשויים להיות מקרים בהם סרטיפיקט מבוטל. השאלה היא איך לקוח שמקבל סרטיפיקט, שתאריך הפג תוקף שלו עדיין לא חלף, לדעת שהסרטיפיקט לא בוטל בעצם? נדון בשלוש שיטות:

- CRL
- OCSP
- OCSP Stapling

CRL

כל CA מחזיק רשימה של כל הסרטיפיקטים שהוא הנפיק ואשר אינם בתוקף, כלומר Revoked. שם הרשימה הזו הוא CRL, קיצור של Certificate Revocation List. הבדיקה האם סרטיפיקט אינו בתוקף היא פשוטה למדי.
בשלב הראשון, על הלקוח להוריד את הרשימה. את הלינק לרשימה תוכלו למצוא בשדה בשם CRL Distribution Points בתוך הסרטיפיקט. הכניסו את הלינק לדפדפן ותורידו את ה-CRL.

כזכור, לכל סרטיפיקט יש מספר ייחודי, Serial Number. כל מה שנותר לעשות הוא לבדוק אם ה-Serial Number של הסרטיפיקט שבידיכם תואם לאחד המספרים שבקובץ. אם כן – הסרטיפיקט מבוטל.

אולם לשיטה זו יש חסרונות.

החיסרון הראשון הוא שלוקח זמן מרגע שסרטיפיקט מבוטל ועד שהוא מופיע ברשימה. דבר זה עלול לגרום לכך שבאופן זמני ניתן יהיה להשתמש בסרטיפיקט שאמור להיות מבוטל.

החיסרון השני הוא שהורדת הקובץ והבדיקה דורשת זמן ויוצרת שיהוי קצר בדפדפן. מה לעשות, שאנשים לא אוהבים להמתין, לכן דפדפנים הפסיקו להשתמש בשיטה זו.

OCSP

כתחליף ל-CRL האיטי, דפדפנים עברו להשתמש ב-OCSP, קיצור של Online Certificate Status Protocol.

בעלי ה-CA הקימו שרתי OCSP, שמבצעים דבר פשוט – הם מקבלים מלקוח בקשה הכוללת סרטיפיקט שברצון הלקוח לבדוק, ומחזירים לו תשובה האם הסרטיפיקט עדיין בתוקף.

לדוגמה, Digicert הקימה את השרת http://ocsp.digicert.com.

תרגיל מודרך: בדיקת סטטוס סרטיפיקט מול שרת OCSP

נבדוק סרטיפיקט שניתן על-ידי Digicert. סביר שמי שישתמש ב-Digicert הוא ארגון שמבקש סרטיפיקט מסוג EV, כגון בנק. בדוגמה זו נבדוק את הסרטיפיקט של בנק הפועלים.

```
openssl s_client -connect www.bankhapoalim.co.il:443
```

פקודת ה-bash הבאה מעתיקה את הסרטיפיקט כפי שהוא, מתחילת השורה BEGIN CERTIFICATE ועד סוף השורה END CERTIFICATE, אל קובץ בשם bankcert.pem.

פורמט PEM הוא פורמט של טקסט מקודד base64, שמשמש סרטיפיקטים. את שם הקובץ, כמו את שם הדומיין ממנו אתם מבקשים סרטיפיקט, תוכלו לקבוע כרצונכם:

```
echo | openssl s_client -connect www.bankhapoalim.co.il:443 2>/dev/null | sed -n ‘/BEGIN CERTIFICATE/,/END CERTIFICATE/p’ > bankcert.pem
```

```
barak@DESKTOP-91SIDPQ:~$ echo | openssl s_client -connect www.bankhapoalim.co.il:443 2>/dev/null
 | sed -n '/BEGIN CERTIFICATE/,/END CERTIFICATE/p' > bankcert.pem
```

נבדוק שהקובץ נשמר נכון:

```
barak@DESKTOP-91SIDPQ:~$ cat bankcert.pem
```

```
-----BEGIN CERTIFICATE-----
MIIG7jCCBdagAwIBAgIQB46Wk1BunbFbz5u3idx4BzANBgkqhkiG9w0BAQsFADBE
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMR4wHAYDVQQDExVE
aWdpQ2VydCBFViBSU0EgQ0EgRzIwHhcNMjQxMjE4MDAwMDAwWhcNMjYwMTEzMjM1
OTU5WjCBqzETMBEGCysGAQQBgjc8AgEDEwJJTDEdMBsGA1UEDwwUUHJpdmF0ZSBP
cmdhbml6YXRpb24xEjAQBgNVBAUTCTUyMDAwMDExODELMAkGA1UEBhMCSUwxFjAU
BgNVBAcTDVRlbCBBdml2LVlhZm8xGzAZBgNVBAoTEkJhbmsgSGFwb2FsaW0gTHRk
LjEfMB0GA1UEAxMWd3d3LmJhbmtoYXBvYWxpbS5jby5pbDCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBANTREBM5m6M3DC5t7Nv5Pv63+4Zb/hm/qWvB+D7j
9JdLYDuPbKvGfwf0WW25cWjtNnefEeIk4uZaiG8iJpUf384B9+/FGFjpi1beqkz4
MWl+SzSXknAdyeU05NarWT8h1R0lqqkf65Hv011u9Nb8/WT4FWPrApELvr+KEDbC
weO6tgEkKw7UQxbr2J95QGiBxEUXHaTdXUguhAedJbYzk7CLMwEJ2rQt6eoHVob2
lEEF4TU/yOkZ5pH7hfMUTC7mw9sHpW/lUQOsI2q/q1PogD/8mKrjiVnpNG81cAFt
pHZQ2MtzfAD9f3SRL2GvzEKFzbvMQB4/gwTsPhWMloJ47mcCAwEAAaOCA3IwggNu
MB8GA1UdIwQYMBaAFGpOUL+YaJ1beyB11FkBeUhmkjIGMB0GA1UdDgQWBBQOEuud
tA0jl785BtwU3mJjCbd9njA1BgNVHREELjAsghZ3d3cuYmFua2hhcG9hbGltLmNv
LmlsghJiYW5raGFwb2FsaW0uY28uaWwwSgYDVR0gBEMwQTALBglghkgBhv1sAgEw
MgYFZ4EMAQEwKTAnBggrBgEFBQcCARYbaHR0cDovL3d3dy5kaWdpY2VydC5jb20v
Q1BTMA4GA1UdDwEB/wQEAwIFoDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUH
AwIwdQYDVR0fBG4wbDA0oDKgMIYuaHR0cDovL2NybDMuZGlnaWNlcnQuY29tL0Rp
Z2lDZXJ0RVZSU0FDQUcyLmNybDA0oDKgMIYuaHR0cDovL2NybDQuZGlnaWNlcnQu
Y29tL0RpZ2lDZXJ0RVZSU0FDQUcyLmNybDBzBggrBgEFBQcBAQRnMGUwJAYIKwYB
BQUHMAGGGGh0dHA6Ly9vY3NwLmRpZ2ljZXJ0LmNvbTA9BggrBgEFBQcwAoYxaHR0
cDovL2NhY2VydHMuZGlnaWNlcnQuY29tL0RpZ2lDZXJ0RVZSU0FDQUcyLmNydDAM
BgNVHRMBAf8EAjAAMIIBfgYKKwYBBAHWeQIEAgSCAW4EggFqAWgAdwCWl2S/VViX
rfdDh2g3CEJ36fA61fak8zZuRqQ/D8qpxgAAAZPYvi8LAAAEAwBIMEYCIQCwcfV6
Y2OljjRZIS3GwFA86YXOzs/3/GmKzG0tHQwfPgIhAODgiy7yAaNDM43xMGm+tNRk
2Q60b8itGjryMgvgiKWGAHYAZBHEbKQS7KeJHKICLgC8q08oB9QeNSer6v7VA8l9
zfAAAAGT2L4vBwAABAMARzBFAiBDR81WdC8/De8GnRAYhE2OrY+Pd4ca+HOfmxkQ
lX9lAwIhAIgrIifPGjXBZX9ZK4TuvdbyiS3FoL3AgYZ0/9oJV9ZFAHUASZybad4d
fOz8Nt7Nh2SmuFuvCoeAGdFVUvvp6ynd+MMAAAGT2L4vIgAABAMARjBEAiARMCmS
JBuYZ7eeZe+3VSL0OBAmL4dEs9Z/YAP4TQ7KyAIgIhj/bpVPtsvsoZcRRzAGUYql
b1Do7OVWI7TU0yaNpDQwDQYJKoZIhvcNAQELBQADggEBAKC3h1BNmS0/O1vCt6Wq
FmtUy0PGLzHOooPzNgjUJFeW+nVILoWu3uzK7DqzilJg0ntSk7QwWGAzupae9htT
Niz9ljr/een6g/yc9yWGzW7RW52rnql9UAS4usdDMio1Qgb5dsGSXbzm4cviQg8T
ufE3GIxrVAuubUMr0jPE7eBCZO6JOKrs/m1100ZsXZI39C/uTKV/sYN7PlYpUAAS
nAbKjqLvQQsjnZrEGp8mio7K/m7Eviytymy11WgfM/1rzjaHceaVvGOuXJUubNkm
7os83RnQEmzEdtJ86UfNokHkpsPklHsVqw892+ffle8SXLFbwNeNMS3vxq619okI
wG4=
-----END CERTIFICATE-----
```

```
barak@DESKTOP-91SIDPQ:~$
```

נבדוק מהו שרת ה-ocsp שמטפל בסרטיפיקט הנ"ל:

openssl x509 -in bankcert.pem -noout -ocsp_uri

```
barak@DESKTOP-91SIDPQ:~$ openssl x509 -in bankcert.pem -noout -ocsp_uri
http://ocsp.digicert.com
```

כעת נפנה אל השרת:

openssl ocsp -url http://ocsp.digicert.com -cert bankcert.pem

```
barak@DESKTOP-91SIDPQ:~$ openssl ocsp -url http://ocsp.digicert.com -cert bankcert.pem
No issuer certificate specified
```

התגובה שקיבלנו אומרת שלא סיפקנו לשרת ה-OCSP את הסרטיפיקט של ה-ICA, שהוא זה שחתום על הסרטיפיקט שלנו. למזלנו, לא צריך להוסיף גם את הסרטיפיקט של ה-Root CA...

אם ניכנס אל הסרטיפיקט, נוכל לקרוא את הכתובת שניתן להוריד ממנה את הסרטיפיקט של מי שחתם על הסרטיפיקט, כלומר ה-ICA, תחת השדה CA Issuers:

```
Authority Information Access:
    OCSP - URI:http://ocsp.digicert.com
    CA Issuers - URI:http://cacerts.digicert.com/DigiCertEVRSACAG2.crt
```

נוריד את הסרטיפיקט של ה-ICA:

wget http://cacerts.digicert.com/DigiCertEVRSACAG2.crt

```
barak@DESKTOP-91SIDPQ:~$ wget http://cacerts.digicert.com/DigiCertEVRSACAG2.crt
--2025-09-17 19:00:37--  http://cacerts.digicert.com/DigiCertEVRSACAG2.crt
Resolving cacerts.digicert.com (cacerts.digicert.com)... 104.75.232.13
Connecting to cacerts.digicert.com (cacerts.digicert.com)|104.75.232.13|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1344 (1.3K) [application/pkix-cert]
Saving to: ‘DigiCertEVRSACAG2.crt’

DigiCertEVRSACAG2.crt 100%[=======================>]   1.31K  --.-KB/s    in 0s

2025-09-17 19:00:38 (17.1 MB/s) - ‘DigiCertEVRSACAG2.crt’ saved [1344/1344]
```

נמיר את הקובץ שהורדנו לפורמט מתאים:

openssl x509 -in DigiCertEVRSACAG2.crt -inform DER -out bankICAcert.pem

וכעת נשלח את שניהם אל שרת ה-OCSP:

openssl ocsp -url http://ocsp.digicert.com -issuer bankICAcert.pem -cert bankcert.pem

```
barak@DESKTOP-91SIDPQ:~$ openssl ocsp -url http://ocsp.digicert.com -issuer bankICAcert.pem -cert bankcert.pem
WARNING: no nonce in response
Response verify OK
bankcert.pem: good
        This Update: Sep 17 10:09:01 2025 GMT
        Next Update: Sep 24 09:09:01 2025 GMT
```

הידד, הצלחנו לוודא שהסרטיפיקט הוא בתוקף, עד לתאריך העדכון הבא.

OCSP Stapling

שיטת OCSP היא אמנם מהירה יותר ללקוח מאשר הורדת CRL, אבל גם לה יש חסרונות. תיאורטית, כל לקוח שרוצה לקיים גלישה מאובטחת ולא להוריד קובץ CRL, יפנה לשרת ה-OCSP. מבחינת שרת ה-OCSP, הוא נדרש לעמוד בקצבי פניות גבוהים למדי.

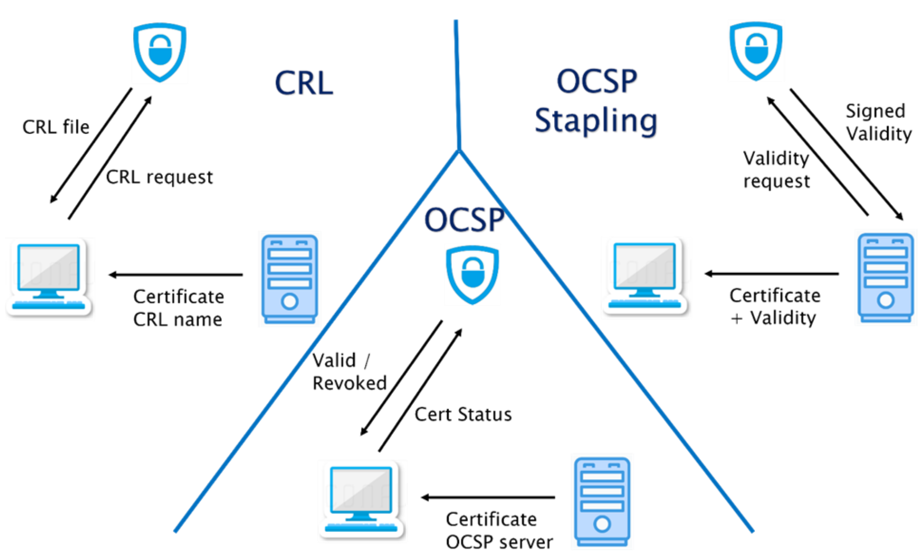
בעלי הדומיין עלולים לקבל מבעלי שרת ה-OCSP בקשה לתשלום, בסגנון "יש כמות גדולה של לקוחות שפונים אלינו כדי לקבל אישור שהסרטיפיקט שלך בתוקף. אם אתה מעוניין שנמשיך לשרת את הלקוחות שלך, שלם יותר על השירות שלנו."

גם הלקוח נמצא בבעיה חדשה – פרטיות. שרת ה-OCSP עלול לרכז מידע על כל האתרים שהלקוח גולש אליהם, והפנייה של הלקוח אל השרת אינה מוצפנת, כך שגם מי שבדרך יכול לאסוף את המידע על גלישת הלקוח.

שיטת OCSP Stapling עובדת בצורה שונה.

השרת פונה מיוזמתו אל שרת OCSP ומשיג אישור שהסרטיפיקט בתוקף. אישור זה יהיה חתום על-ידי שרת ה-OCSP, כדי לוודא שהוא אמיתי. כאשר הלקוח יפנה אל השרת, השרת ישלח לו הן את הסרטיפיקט של עצמו והן את האישור שהוא קיבל משרת ה-OCSP. שיטה זו עונה על כל היתרונות:

- מהירות: הלקוח אינו צריך להוריד קובץ, ואפילו לא לפנות לשרת נוסף.
- עומס על השרת: שרת ה-OCSP מקבל כמות דלילה יחסית של פניות – רק שרתים שרוצים לקבל אישור שהסרטיפיקט שלהם בתוקף, אחת לשבוע.
- שמירה על פרטיות הלקוח: שרת ה-OCSP אינו יודע לאילו לקוחות הסרטיפיקט שלו מועבר.
- יתרון כלכלי לבעלי הדומיין: בניגוד לשיטת OCSP רגיל, שרת ה-OCSP אינו יכול לחייב את בעלי הדומיין לפי כמות הלקוחות שפונים אליו.



סיכום שלוש שיטות הבדיקה שסרטיפיקט בתוקף

Certificate Transparency

היה היתה חברה בשם DigiNotar שהיתה Root CA. הסרטיפיקטים שלה היו מוכרים על-ידי דפדפנים. גורם כלשהו, כנראה מדינה, הצליח לחדור לשרתים שלה ולהוציא סרטיפיקטים חתומים. העניין נחשף בעקבות דיווחים של גולשים מאיראן שלא הצליחו לגשת לחשבון הג'ימייל שלהם. בדיקה העלתה שמישהו זייף אתר של גוגל והצליח לקבל עליו סרטיפיקט. בדיקה מעמיקה גילתה שזה רק קצה הקרחון ויש מאות סרטיפיקטים שזויפו. תוך זמן קצר, דפדפנים הפסיקו לסמוך על סרטיפיקטים של DigiNotar והיא נסגרה. תוכלו לצפות בתיאור המקרה בלינק הבא:

[Ep 3: DigiNotar, You are the Weakest Link, Good Bye!]

זיוף הסרטיפיקטים של DigiNotar היה בעל השפעה עמוקה על סרטיפיקטים ויצר שינוי במודל PKI ובפרוטוקול TLS. הטכנולוגיה שפותחה כדי להתמודד עם מקרים דומים נקראת Certificate Transparency.

מקרה DigiNotal חשף בעיה במנגנון הסרטיפיטקים. אנחנו יודעים לזהות סרטיפיקט שהוא בתוקף, כלומר לא Revoked, אבל איך אפשר לגלות אילו סרטיפיקטים **צריכים להיות** Revoked? אתם בעלים של דומיין, ו-CA כלשהו חותם על סרטיפיקט עם שם הדומיין שלכם. איך בכלל תדעו שזה קרה? והמקרה ש-CA עלול לחתום על סרטיפיקט שהוא לא אמור לחתום עליו עלול לקרות ממגוון תרחישים. מישהו יכול לפרוץ ל-CA, כמו במקרה DigiNotar. מישהו יכול להצליח להטעות את ה-CA ולחלץ ממנו חתימה. מישהו יכול לקבל סרטיפיקט על דומיין עם שם מטעה (לדוגמה cyb3r.org.il). מכל הסיבות הללו נדרש שינוי מערכתי.

הרעיון הכללי של Certificate Transparency, או בקיצור CT, הוא שכל סרטיפיקט בעולם יירשם במאגר מיוחד. ל-CT יש שלושה נדבכים:

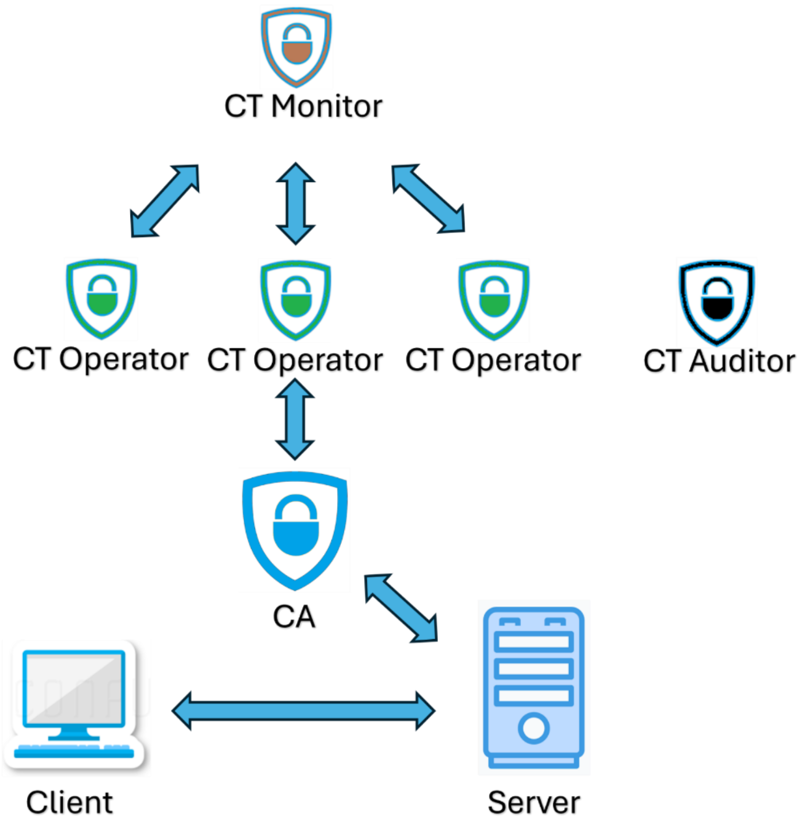
מאגרי מידע שהסרטיפיקטים שמורים בהם. מאגרים אלו צריכים להיות בעלי כמה תכונות. ראשית, ביזור. לא טוב שיהיה גורם אחד שמחזיק במאגר יחיד. הרישום של הסרטיפיקטים צריך להתחלק בין כמה מאגרים שמנוהלים על-ידי גופים שונים ומגבים זה את זה. שנית, אי אפשר למחוק רשומות. אחרת, מי שהשיג סרטיפיקט מזויף עלול למחוק את הרישום שלו במאגר. שלישית, יכולת לוודא במהירות שכל המידע במאגר נשמר בצורה מלאה וללא שינוי.

יכולת חיפוש במאגרים. כל בעלים של דומיין יכול לבצע חיפוש על כל הסרטיפיקטים שהוציאו על השם שלו ולבדוק שאין סרטיפיקט לא מוכר. בעלים של דומיין עשוי גם לבדוק שלא הוציאו סרטיפיקטים עם שמות מטעים, שדומים לשלו.

אכיפה. אפשר להקים מאגרים ויכולות חיפוש, אבל כדי שהמאגרים יהיו שימושיים, צריך גורם כלשהו שיאכוף על מי שמוציא סרטיפיקט את החובה לרשום אותו במאגר. הגורם הזה הוא הדפדפן. הדפדפן לא יקבל סרטיפיקט שאין הוכחה כי הוא נשלח לרישום במאגר מידע.

CT Operators, Auditors, Monitors

האיור הבא ממחיש את רכיבי הרשת החדשים שנוספו עבור CT:



רכיבי הרשת של Certificate Transparency ומיקומם לצד מודל PKI

- CT Operator הם מאגרי המידע ששומרים את הלוגים של כל הסרטיפיקטים. כאשר CA יוצר סרטיפיקט, הוא צריך לשלוח אותו לפחות ל-CT Operator אחד. נוסף על שמירת הסרטיפיקטים עצמם, ה-CT Operators שומרים מבנה מתימטי שמאפשר לבדוק שכל הסרטיפיקטים שהוכנסו אליהם נשמרים ללא שינוי ולא נמחקו. המבנה המתימטי נקרא Merkle Hash Tree. הרעיון הכללי של עצי מרקל הוא לבצע Hash לכל "עלה" (במקרה שלנו – סרטיפיקט) ולאגוד ביחד קבוצות של Hashים של "עלים" ל-Hash משותף. גם את הקבוצות הללו נאגד לתוך Hashים וכך הלאה, עד שנקבל Hash של כל ה"עלים". כעת אם מישהו ניסה להכניס שינוי באחד הסרטיפיקטים או למחוק אותו, המבנה המתימטי מאפשר לגלות זאת יחסית בקלות וכן לקבוע איזה "עלה" השתנה או נמחק.

הסרטון הבא של Computerphile מסביר על עצי מרקל וחשיבותם לטכנולוגיית הבלוקצ'יין:

[The Blockchain & Bitcoin -Computerphile]

- תפקיד ה-CT Auditors הוא לעבור על המאגרים של ה-CT Operators ולוודא שהרישומים שלהם מדוייקים ושלא בוצעו פעולות של מחיקה או שינוי סרטיפיקטים. ארגון המידע ב-Merkle Hash Tree מקל עליהם לבצע זאת.
- כיוון שיש אוסף של CT Operators וכיוון ששרת CA לא חייב לרשום סרטיפיקט חדש בכולם, יש שוני ברישומים של CT Operators. ה-CT Monitors מעניקים יכולות חיפוש של סרטיפיקט אצל כל ה- CT Operators. ה-Monitors עשויים להתריע בפני בעלים של דומיין על רישום של סרטיפיקט חדש על שם הדומיין שלו, או על רישום שמות דומיין דומים (כגון cyb3r.org.il) שעלולים להיות מנוצלים למטרות הונאה.

Signed Certificate Timestamp

מה קורה מרגע שבעלים של דומיין מבקש סרטיפיקט ועד שלקוח, שגולש לשרת של הדומיין, מקבל אישור שהדומיין רשום אצל CT Operator?

את ההתחלה אנחנו כבר מכירים. בעל הדומיין ישלח Certificate Signing Request – CSR – אל שרת ה-CA. שרת ה-CA מחזיר לו את הסרטיפיקט חתום.

בחלק זה התעמקנו עד כה.

החלק החדש הוא ששרת ה-CA שולח את הסרטיפיקט ל-CT Operator כלשהו, כדי שיוסיף אותו ללוגים שלו. ה-CT Operator מחזיר לשרת Signed Certificate Timestamp – SCT. זהו אישור חתום על-ידי ה- CT Operator, שמאשר "הסרטיפיקט הזה הוגש לי". האישור חתום במפתח הפרטי של ה-CT Operator.

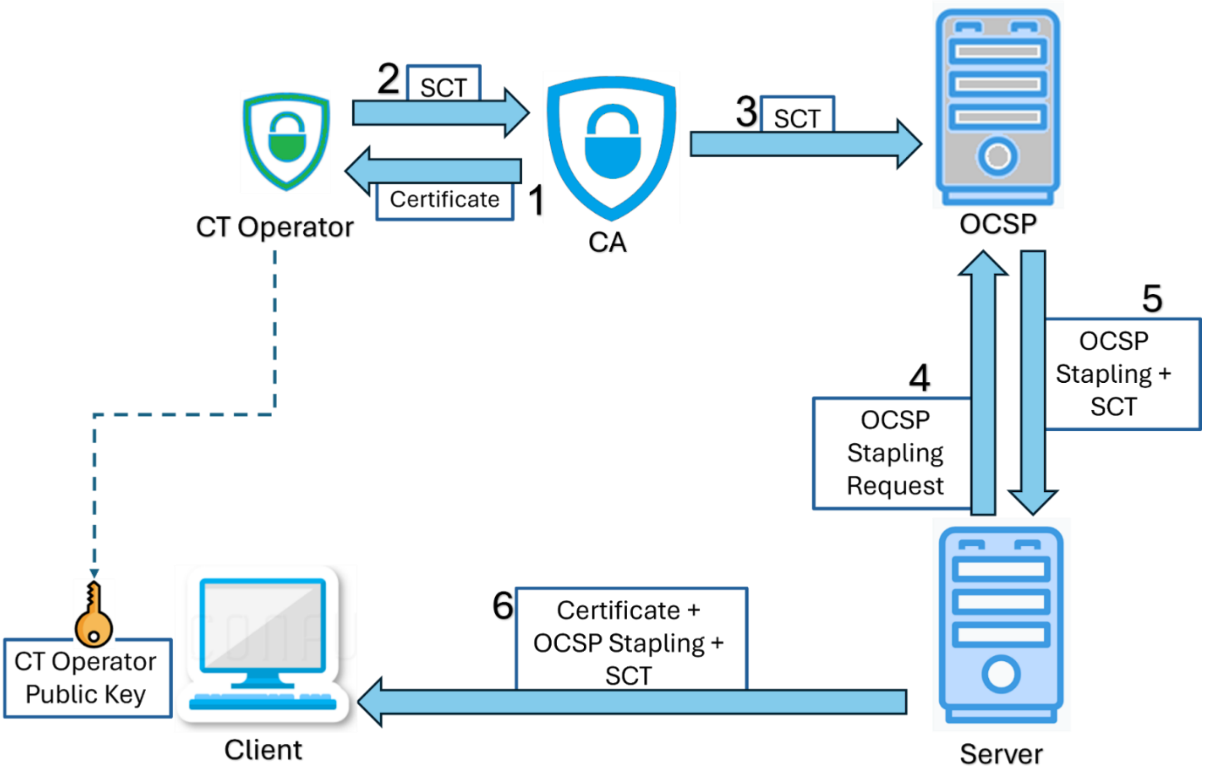
כאן נכנס התפקיד החשוב של הדפדפן. הדפדפן דורש לראות את ה-SCT. סרטיפיקט שיישלח בלי SCT לא יתקבל. הדפדפן למעשה אוכף על בעלי הדומיינים ועל שרתי ה-CA את הפנייה אל ה-CT Operators, שאם לא כן – היו כאלה ששולחים את הסרטיפיקט לרישום אצלם, והיו כאלה (במיוחד מי שמטרתם זדונית) שלא היו שולחים.

מה הדפדפן עושה עם ה-SCT? בתוך קוד הדפדפן שמורים המפתחות הציבוריים של כל ה-CT Operators. לכן, כאשר הדפדפן מקבל SCT חתום הוא יודע לפתוח את החתימה ולוודא שהיא מקורית.

קיימות שלוש שיטות להעברת ה-SCT אל הלקוח:

- OCSP או OCSP Stapling
- העברה בתוך שדה בפרוטוקול TLS
- העברה בתוך הסרטיפיקט

שימוש ב-OCSP מתבצע באופן הבא:



העברת SCT באמצעות OCSP Stapling

1. ה-CA (שכבר קיבל CSR מהשרת) שולח את הסרטיפיקט ל-CT Operator.

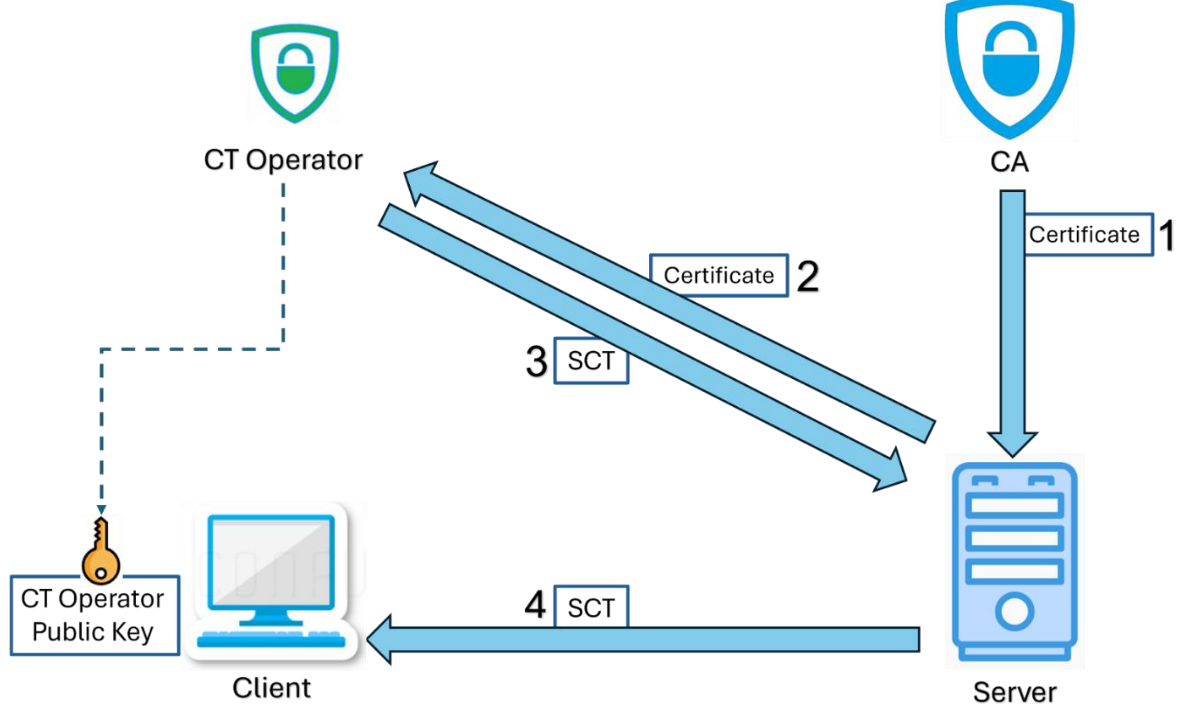
2+3: ה-CA מקבל חזרה SCT ומעביר אותו לשרת OCSP.

4+5: השרת שולח בקשת OCSP Stapling ומקבל אותה חתומה יחד עם ה-SCT.

6: השרת מעביר ללקוח את ה-SCT יחד עם הסרטיפיקט ואישור ה-OCSP. הלקוח יודע לאמת את ה-SCT באמצעות המפתח הציבורי של ה-CT Operator, שמגיע יחד עם הדפדפן.

במקרה שהלקוח מבצע OCSP, ולא OCSP Stapling, הוא יקבל את ה-SCT ישירות משרת ה-OCSP.

שימוש בשדה בתוך פרוטוקול TLS מתבצע באופן הבא:



העברת SCT באמצעות שדה בפרוטוקול TLS

1: השרת מקבל את הסרטיפיקט מה-CA (בתגובה לבקשה שלו, CSR, שלא מוצגת באיור).

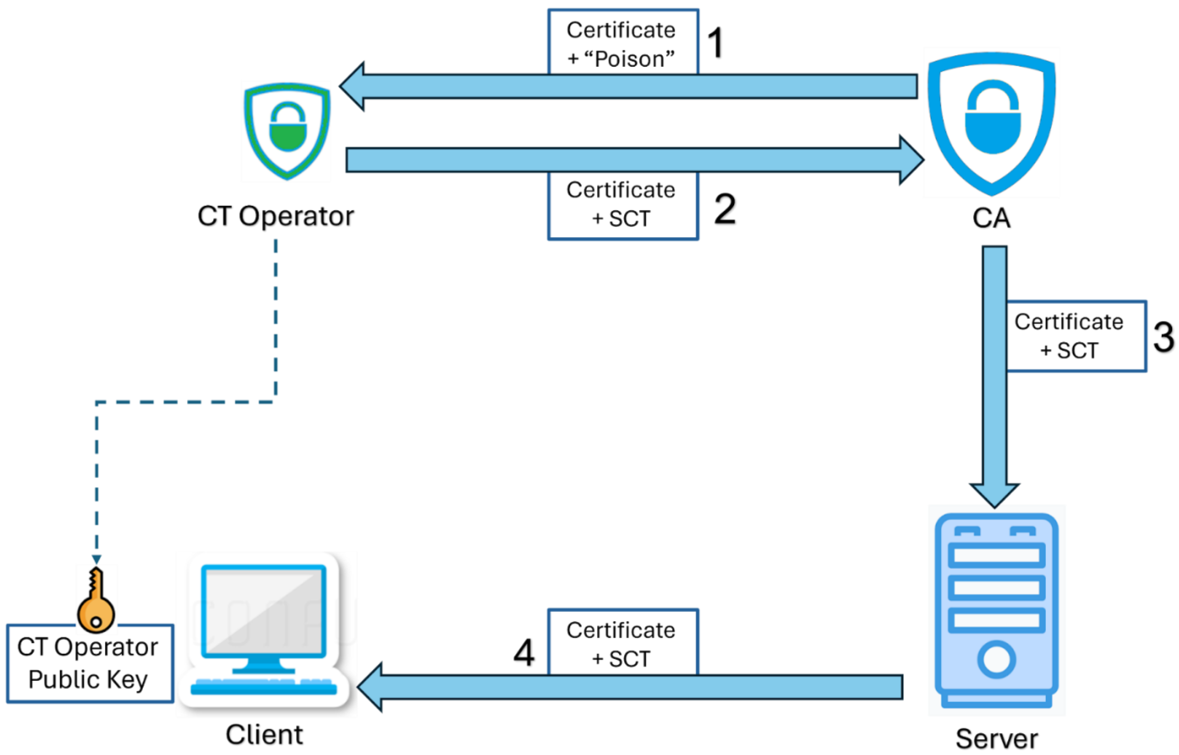
2+3: השרת שולח את הסרטיפיקט ל-CT Operator ומקבל ממנו SCT.

4: אחד השדות של פרוטוקול TLS כולל אפשרות לשלוח SCT.

העברת SCT בתוך הסרטיפיקט היא טריקית מעט. יש פה בעיה של מבוי סתום (או באנגלית Deadlock). זהו מונח מקובל במדעי המחשב, שמציין תרחיש שבו שני גורמים מחכים זה לזה – המתנה שלעולם לא תסתיים. ה-CA רוצה שהסרטיפיקט שהוא מכין כבר יכלול בתוכו את ה-SCT. לכן, הוא לא יכול לסגור את הסרטיפיקט ולהעביר אותו ל-CT Operator עד שהוא לא מקבל את ה-SCT. מצד שני, כל עוד הוא אינו מעביר את הסרטיפיקט ל-CT Operator הוא לא יקבל ממנו SCT...

הפתרון לבעיה הוא שה-CA מכין סרטיפיקט "מורעל". לתוך הסרטיפיקט הרגיל שהוא יוצר, ה-CA מוסיף שדה שנקרא CT PreCertificate. הערך של השדה – המחרוזת "Poison". ה"רעל" מסמן למי שמקבל אותו שזה סרטיפיקט שעדיין לא עבר CT Operator.

ה-CT Operator יחליף את ה-Poison בחתימה האמיתית שלו ויחזיר ל-CA. התהליך נראה כך:



העברת SCT בתוך סרטיפיקט

1: שרת ה-CA יוצר סרטיפיקט "מורעל", עם שדה של PreCertificate.

2: ה-CT Operator מחליף את ה-Poison ב-SCT חתום ומחזיר ל-CA.

3+4: השרת מקבל את הסרטיפיקט, הכולל בתוכו גם SCT, ומעביר אותו ללקוח.

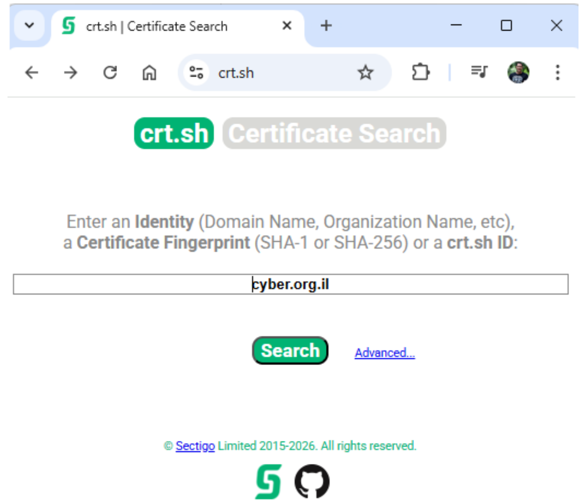
צפיה ב-SCT

היכנסו לאתר

https://crt.sh/

לפנינו CT Monitor המופעל על-ידי Sectigo. אפשר לחפש בו את כל הסרטיפיקטים שניתנו לדומיין כלשהו. ה-Monitor, כזכור, מאגד את כל הרשומות מכל ה-Operators.

נחפש בו את הסרטיפיקט של cyber.org.il:



נקבל רשימת סרטיפיקטים ארוכה למדי, כיוון שלשרתים רבים תחת הדומיין יש סרטיפיקטים משלהם. נתמקד בסרטיפיקטים העדכניים של www.cyber.org.il:

crt.sh ID	Logged At ⇧	Not Before	Not After	Common Name	Matching Identities	
24187561606	2026-02-04	2026-02-04	2026-05-05	cyber.org.il	cyber.org.il www.cyber.org.il	C=US, O=Let's Encrypt, CN=R12
24187539334	2026-02-04	2026-02-04	2026-05-05	cyber.org.il	cyber.org.il www.cyber.org.il	C=US, O=Let's Encrypt, CN=R12

כפי שרואים יש, לכאורה, שני סרטיפיקטים, שיש להם אותו תוקף.

אך הסרטיפיקט התחתון הוא ההעתק שכולל את ה-Poison. בתוך ה-X509 Extensions אפשר לראות אותו:

```
X509v3 extensions:
    X509v3 Key Usage: critical
        Digital Signature, Key Encipherment
    X509v3 Extended Key Usage:
        TLS Web Server Authentication, TLS Web Client Authentication
    X509v3 Basic Constraints: critical
        CA:FALSE
    X509v3 Subject Key Identifier:
        CE:7B:25:67:61:65:02:8A:07:84:B5:07:A6:5E:32:CB:11:4E:0F:DB
    X509v3 Authority Key Identifier:
        keyid:00:B5:29:F2:2D:8E:6F:31:E8:9B:4C:AD:78:3E:FA:DC:E9:0C:D1:D2

    Authority Information Access:
        CA Issuers - URI:http://r12.i.lencr.org/

    X509v3 Subject Alternative Name:
        DNS:cyber.org.il
        DNS:www.cyber.org.il
    X509v3 Certificate Policies:
        Policy: 2.23.140.1.2.1

    X509v3 CRL Distribution Points:

        Full Name:
          URI:http://r12.c.lencr.org/79.crl

    CT Precertificate Poison: critical
        NULL
```

לעומת זאת, הסרטיפיקט העליון כולל את ה-SCT:

```
CT Precertificate SCTs:
    Signed Certificate Timestamp:
        Version   : v1 (0x0)
        Log Name  : DigiCert Wyvern 2026h1
        Log ID    : 64:11:C4:6C:A4:12:EC:A7:89:1C:A2:02:2E:00:BC:AB:
                    4F:28:07:D4:1E:35:27:AB:EA:FE:D5:03:C9:7D:CD:F0
        Timestamp : Feb 12 18:41:08.385 2026 GMT
        Extensions: none
        Signature : ecdsa-with-SHA256
                    30:44:02:20:50:58:06:DE:B7:43:D0:09:09:8E:CD:A8:
                    07:98:9F:FA:51:85:DC:1E:8E:73:98:07:1E:3F:4F:7A:
                    62:35:4E:6F:02:20:51:4D:94:05:0D:C4:AA:CC:1F:BC:
                    BB:A5:59:95:5D:33:C6:7C:AC:26:BF:6C:10:B0:D7:13:
                    C0:59:64:12:02:1E
    Signed Certificate Timestamp:
        Version   : v1 (0x0)
        Log Name  : Let's Encrypt Willow 2026h1
        Log ID    : E3:23:8D:F2:8D:A2:88:E0:AA:E0:AC:F0:FA:90:C9:85:
                    F0:B6:BF:F5:D2:A5:27:B0:01:FC:1C:44:58:C4:B6:E8
        Timestamp : Feb 12 18:41:10.523 2026 GMT
        Extensions: 00:00:05:00:32:5F:3C:7A
        Signature : ecdsa-with-SHA256
                    30:44:02:20:60:AD:66:EB:66:C3:22:CD:71:34:7D:43:
                    3E:F9:57:B4:00:1A:C4:C6:3C:E1:27:94:91:ED:99:7B:
                    ED:58:06:5B:02:20:53:A2:B4:EE:B0:86:EA:07:DE:C1:
                    DD:35:83:9E:50:85:0D:74:D5:0A:31:3D:F9:F0:6D:5E:
                    43:06:20:59:7F:52
```

כפי שרואים, יש שני CT Operators שהסרטיפיקט נשלח אליהם והם חתמו על SCT.

תרגיל מסכם: הדמיית סרטיפיקט של מקום עבודה

בתרגיל מודרך זה נדמה מצב שבו מקום עבודה מעוניין להאזין ולפקח על תעבורת האינטרנט של העובדים. מקומות עבודה עשויים לעשות זאת במקרים, לדוגמה, שיש צורך לוודא שהעובדים אינם מכניסים בטעות נוזקות אל הארגון. אם העובדים משתמשים בסוקטים מאובטחים, הרשת של מקום העבודה משמשת רק כצינור ואין לה יכולת לפתוח את התקשורת ולבדוק אם אין בהם דבר מה מזיק.

כדי לבצע זאת, אחת השיטות של מקום העבודה הוא ליצור סרטיפיקט של Root CA ולהתקין אותו אצל כלל העובדים. כל הגלישה של העובדים תעבור דרך שרת של מקום העבודה, שיבצע התחזות אל האתרים אליהם הם גולשים. השרת של מקום העבודה יקים במקביל סוקטים מאובטחים אל השרתים האמיתיים שאליהם העובדים רצו לגלוש. שיטה זו נקראת Man In The Middle, או בקיצור MITM.

אם כך, מהם הצעדים שמקום העבודה יצטרך לעשות כדי להתחזות? נמחיש זאת על-ידי אתר לדוגמה. חלק מהשלבים ביצענו כבר בתרגילים מודרכים קודמים בפרק זה. השלבים שעלינו לבצע:

1. ניצור Certificate Authority ונכניס את הסרטיפיקט שלו ל-certmgr, כך שהמחשב יכיר בו כבסיס לחתימות. שלבי העבודה:

א. יצירת זוג מפתחות ל-CA שלנו

ב. ה-CA יחתום לעצמו על סרטיפיקט

ג. נייבא את הסרטיפיקט לתוך certmgr

שלב זה בוצע בתרגיל מודרך – יצירת Root CA Certificate

2. ניצור לדומיין שלנו סרטיפיקט חתום על-ידי ה-CA. שלבי העבודה:

א. יצירת זוג מפתחות לדומיין שלנו

ב. יצירת CSR

ג. ה-CA יענה ל-CSR של הדומיין ויעניק לו סרטיפיקט חתום

שלב זה בוצע בתרגיל מודרך – יצירת סרטיפיקט חתום על-ידי ה-CA

3. ניצור שרת HTTPS שמוסר את הסרטיפיקט החתום ללקוח (הדפדפן). הדפדפן יוודא מול ה-certmgr.msc שאכן הסרטיפיקט כשר.

להלן הוראות ביצוע לשלב 3.

הקוד הבא יוצר שרת HTTPS:

```
import socketserver
import http.server
import ssl

HOST = "0.0.0.0"
PORT = 443
CERT_FILE = "mytestdomain.co.il.crt"
KEY_FILE = "mytestdomain.co.il.key"

handler = http.server.SimpleHTTPRequestHandler
with socketserver.TCPServer((HOST, PORT), handler, bind_and_activate=False) as httpd:
    httpd.allow_reuse_address = True
    httpd.server_bind()
    httpd.server_activate()
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile=CERT_FILE, keyfile=KEY_FILE)
    httpd.socket = context.wrap_socket(httpd.socket, server_side=True)
    print(f"Serving HTTPS on {HOST}:{PORT} (Ctrl+C to stop)...")
    httpd.serve_forever()
```

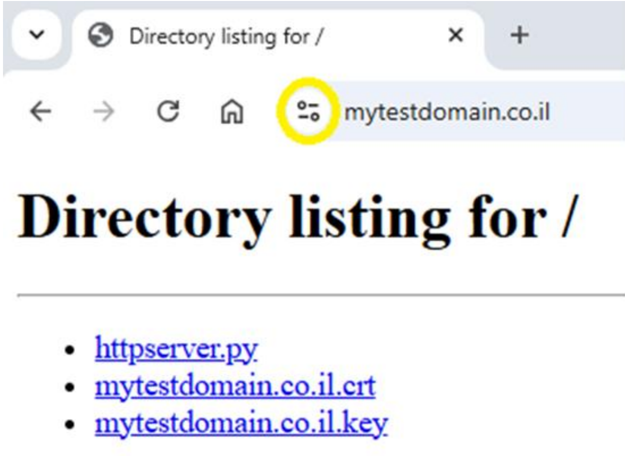
העתיקו את קובץ ה-key וקובץ ה-crt לתוך התיקייה שבה נמצא הסקריפט, לטובת פשטות (או שתשנו את הקוד כך שיכלול את הנתיב המלא לקבצים).

כעת צריך לגרום למחשב שלנו לפנות אל השרת המקומי כאשר הוא ניגש לדומיין המומצא שלנו. הדומיין הזה הרי אינו נמצא אצל שרתי DNS. נשתמש בטריק שלמדנו בפרק על DNS.

פיתחו את הקובץ hosts, שנמצא בנתיב c:\windows\system32\drivers\etc, במצב admin. הוסיפו לו את השורה:

127.0.0.1 www.mytestdomain.co.il

כעת, כאשר תגלשו בדפדפן ל-https://www.mytestdomain.co.il תקבלו את תשובת השרת:



החלק היפה בתרגיל זה הוא האיור המודגש בצהוב. הדפדפן קיבל את הסרטיפיקט, אימת אותו ומראה סימן כי הקישור אל האתר הפיקטיבי שלנו הוא מאובטח. הסרטיפיקט ששתלנו ב-Root CA עשה את העבודה!

...רגע אחד.

למדנו כי הדפדפן מצפה לראות SCT כדי לאשר את השרת בתור שרת מאובטח. בתרגיל זה לא ביצענו שום פעולה שהכניסה SCT לתוך הסרטיפיקט שלנו וגם לא השתמשנו בשיטה אחרת. אם כך, מדוע הדפדפן שלנו בטח בדומיין הפיקטיבי שיצרנו?

התשובה היא שדפדפן כרום, כמו דפדפנים אחרים, מחריג מהבדיקה סרטיפיקטים שנחתמו על-ידי Root CA. פעולת ההתחזות שלנו הצליחה בצורה מלאה כיוון ששתלנו סרטיפיקט של Root CA.

סיכום

בפרק זה למדנו כיצד משיגים אבטחה באמצעות סרטיפיקטים. התחלנו מהסבר על מהו בכלל סרטיפיקט ואיך משיגים סרטיפיקט. למדנו אילו סוגי סרטיפיקטים קיימים. הבנו איך עובדת שרשרת יצירת האמון באינטרנט, החל מ-Root CA, דרך ICA וכלה בדומיין שמבקש סרטיפיקט.

למדנו כיצד מונעים מצב שבו גורם כלשהו לוקח סרטיפיקט שאינו שלו ומשתמש בו כדי להתחזות. ראינו כי הלקוח אינו פתי, אלא מוודא שלשרת יש את המפתח הפרטי המתאים לסרטיפיקט שהוא מציג.

הבנו אילו צעדים נעשים כדי למנוע נפוצות של סרטיפיקטים שעלולים לשמש להונאה: המפתח הפרטי של ה- Root CA גדול מאד ונעשה בו שימוש מועט, רק גורם מורשה בתור CA יכול לחתום על סרטיפיקט, ישנן רשימות של סרטיפיקטים פגי תוקף.

לבסוף, למדנו איך מתבצעת בדיקת תוקף לסרטיפיקט בשלוש שיטות: CRL, OCSP, OCSP Stapling.

על הדרך, ביצענו מספר תרגולים מעשיים שכללו הורדה של סרטיפיקטים וכן הכנה של סרטיפיקטים. ראינו כי מי שיש לו גישה ל-certmgr יכול לשתול שם סרטיפיקט של Root CA ובכך כל דומיין שברצונו יעבור אותנטיקציה בהצלחה. כעת אנחנו מוכנים לשלב הבא – כיצד כל מה שלמדנו מתחבר יחד, לפרוטוקול שמבצע את כל מה שנדרש עבור Confidentiality, Integrity, Authentication?

פרק 20: TLS 1.2

מוקדש לזכרם של סא"ל יצחק הרוש, סמל אורן הרשקו, רס"ן אומרי חי בן משה, סגן ערן שלם, סגן איתן אבנר בן יצחק, סגן רון אריאלי. שמותיהם הותרו לפרסום עם תחילת הכתיבה של פרק זה. הצער עמוק. מי ייתן ויהיו הנופלים האחרונים.

הקדמה

מטרת הפרק היא להבין את תהליך ה-handshake של פרוטוקול TLS 1.2. לפני שניכנס להסבר על גרסה 1.2, נזכיר מה שפירטנו מוקדם יותר על גרסאות ה-TLS. ל-TLS ולקודמו, SSL, יש גרסאות שהוכרזו לא מאובטחות ואינן בשימוש, ושתי גרסאות שעדיין בשימוש: גרסה 1.2 היא הגרסה הוותיקה ביותר, משנת 2008. גרסה 1.3 היא העדכנית יותר, משנת 2018.

פרק זה יעסוק ב-TLS 1.2 הן מכיוון שגרסה זו נמצאת עדיין בשימוש נפוץ והן מכיוון שהבנה שלה תסייע להבין את גרסה 1.3, שיוקדש לה פרק נפרד.

למעשה, כל הפרקים שלמדנו עד כה, מטרתם היתה לצקת את בסיס הידע הנדרש כדי שתיאור ה-handshake יהיה קל יותר. היה מאוד קשה לפתוח הסנפה ולהסביר מה אנחנו רואים, בלי כל הידע שלמדנו: הצפנות סימטריות, החלפת מפתחות, אלגוריתמי Hash, חתימות וכמובן סרטיפיקטים.

נתחיל בהסבר מהו TLS record, לאחר מכן נסקור את ה-TLS handshake בשתי גרסאות – RSA ו-DH, ונסיים בהסבר איך נוצרים מפתחות ההצפנה.

כדי להבין איך הכל מתחבר, נשתמש הרבה ב-Wireshark, ותוך כדי גם נלמד כיצד לפענח הצפנה של הסנפות בעזרת מפתחות.

TLS Records

פרוטוקול TLS מחליף רשומות, או records, בין השרת והלקוח. כלומר, השרת והלקוח עדיין מעבירים פקטות זה לזה, אך כל פקטה יכולה להכיל record אחד או יותר. ה-records מאורגנים במבנה שכולל שלושה שדות, ולאחר מכן את ה-records עצמן. שלושת השדות הם:

- Content Type
- Version
- Length

נתחיל בלימוד תוך כדי הסנפה. הורידו את הקובץ הבא, הכולל הסנפה לאתר המרכז לחינוך סייבר:

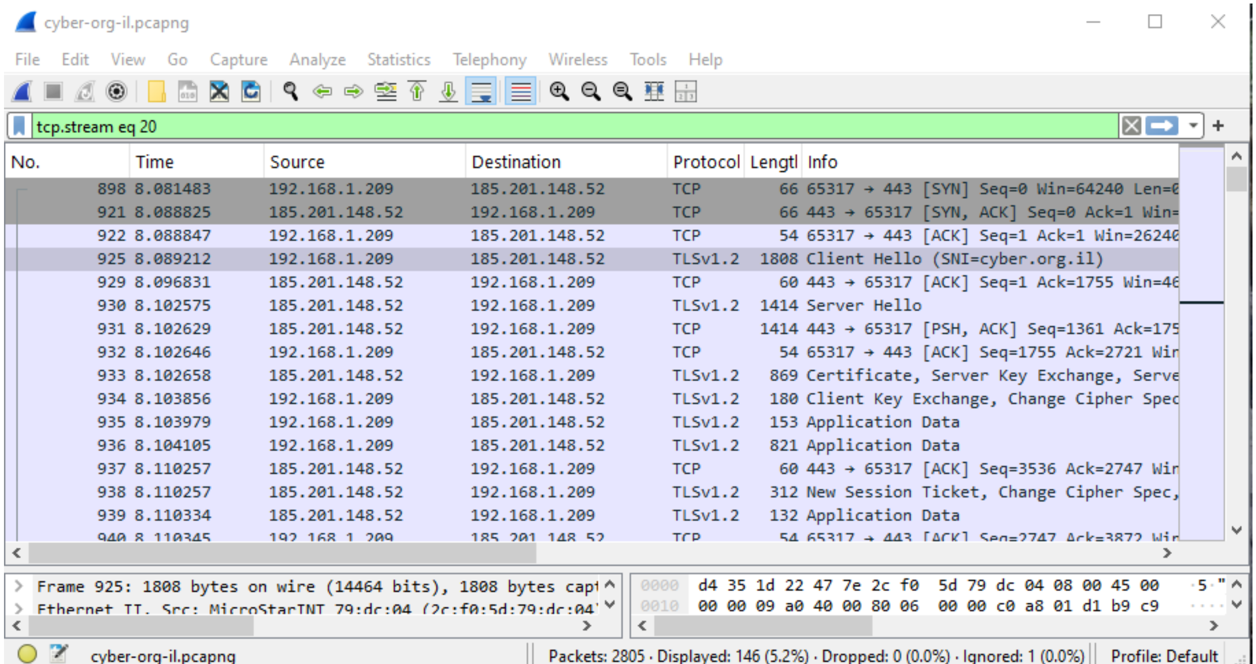
https://data.cyber.org.il/networks/cyber-org-il-tls.pcapng

אתם כמובן יכולים לבצע את ההסנפה הזו בכוחות עצמכם, אולם סביר שעם הזמן גרסת ה-TLS של האתר תשתנה, לכן עדיף לעבוד עם קובץ ההסנפה בשלב זה.

באמצעות פילטור, קבלו את זרם הפקטות השייך לתקשורת בין הלקוח לבין cyber.org.il. תזכורת – אפשר להתחיל מפילטר:

frame contains "cyber"

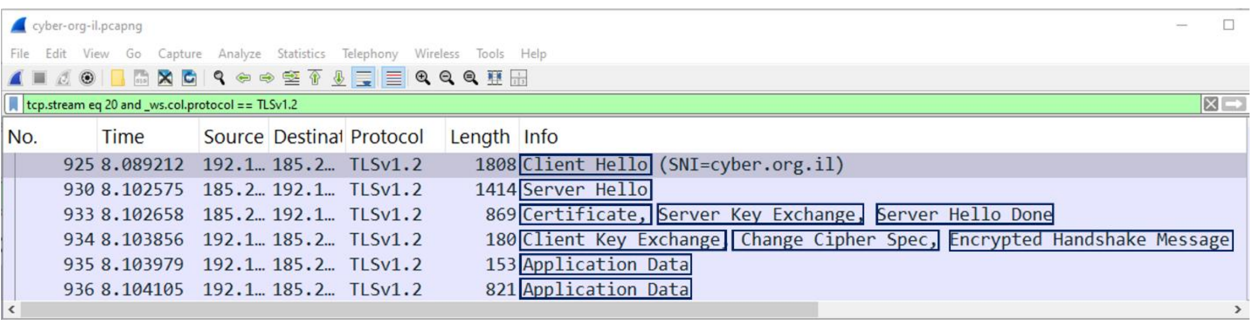
ולהמשיך עם קליק ימני ו-Follow TCP Stream:



ניתן לראות פקטות משני פרוטוקולים – TCP ו-TLS 1.2. שימו לב שהפקטה הראשונה של TLS 1.2 נשלחת מהלקוח אל השרת מיד לאחר סיום ה-TCP Three Way Handshake. השרת והלקוח סיימו את לחיצת היד של TCP ומיד מתחילים בהקמת קישור TLS. אין שום דבר אחר שצריך להתרחש בין לבין. שימו לב גם לשוני מהסנפות HTTP שסקרנו בחלקו הראשון של הספר, בהן לאחר ה-Three Way Handshake נשלחה מהלקוח בקשת HTTP.

לאחר הקמת קישור ה-TCP, הפקטות שנשלחות ומסומנות תחת פרוטוקול TCP הן חלקים של TLS 1.2. אין לנו עניין בחלקים אלא בצירוף שלהם ל-TLS Records, לכן לטובת הצגה נוחה יותר, נדייק את הפילטר כך שיציג רק פקטות של פרוטוקול TLS 1.2. הוסיפו לפילטר הקיים:

and _ws.col.protocol == TLSv1.2



לטובת ההמחשה, כל אחד מה-Records, מעכשיו נקרא להם "רשומות", מודגש בריבוע כחול. אפשר לראות שכל התקשורת בין השרת והלקוח מורכבת מרשומות מסוגים שונים, ושלעיתים פקטה אחת של TLS כוללת יותר מאשר רשומה אחת. כעת נסקור את מבנה הרשומות של TLS.

הקליקו על פקטה 933. ניתן לראות שפקטה זו מורכבת משלוש רשומות:

```
Wireshark · Packet 933 · cyber-org-il.pcapng
> Frame 933: 869 bytes on wire (6952 bits), 869 bytes captured (6952 bits) on interfac
> Ethernet II, Src: TechnicolorD_22:47:7e (d4:35:1d:22:47:7e), Dst: MicroStarINT_79:d
> Internet Protocol Version 4, Src: 185.201.148.52, Dst: 192.168.1.209
> Transmission Control Protocol, Src Port: 443, Dst Port: 65317, Seq: 2721, Ack: 1755,
> [3 Reassembled TCP Segments (2853 bytes): #930(1281), #931(1360), #933(212)]
v Transport Layer Security
  > TLSv1.2 Record Layer: Handshake Protocol: Certificate
v Transport Layer Security
  > TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
  > TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
```

כל רשומה שנבחר, בין מפקטה זו או מפקטה אחרת, תכלול את השדות שהצגנו בפתיחה:

```
˅ Transport Layer Security
  ˅ TLSv1.2 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 2848
    > Handshake Protocol: Certificate
˅ Transport Layer Security
  ˅ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 589
    > Handshake Protocol: Server Key Exchange
  ˅ TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 4
    > Handshake Protocol: Server Hello Done
```

Content Type

יש רשומות TLS מסוגים שונים. בשלב התחלתי זה אנחנו עדיין בשלב ה-TLS Handshake, שסימונו בפרוטוקול הוא 22, לכן כל הרשומות שלנו הן מסוג 22. אם תקליקו על פקטה 935 לדוגמה, תראו שהרשומה שלה היא מסוג Application Data, שסימונו הוא 23.

המיספור של ה-Content Type לפי הפרוטוקול הוא:

- 20: סוג "Change Cipher Spec". משמעותו היא "התחל להצפין" ונלמד עליו בקרוב.
- 21: סוג "Alert". אזהרה יכולה להיות Warning או Fatal. נקבל Fatal במקרים כגון כשלון של ה- TLS Handshake, או כאשר הסרטיפיקט פג תוקף.
- 22: כפי שראינו, "Handshake".
- 23: כפי שראינו, "Application Data".

Version

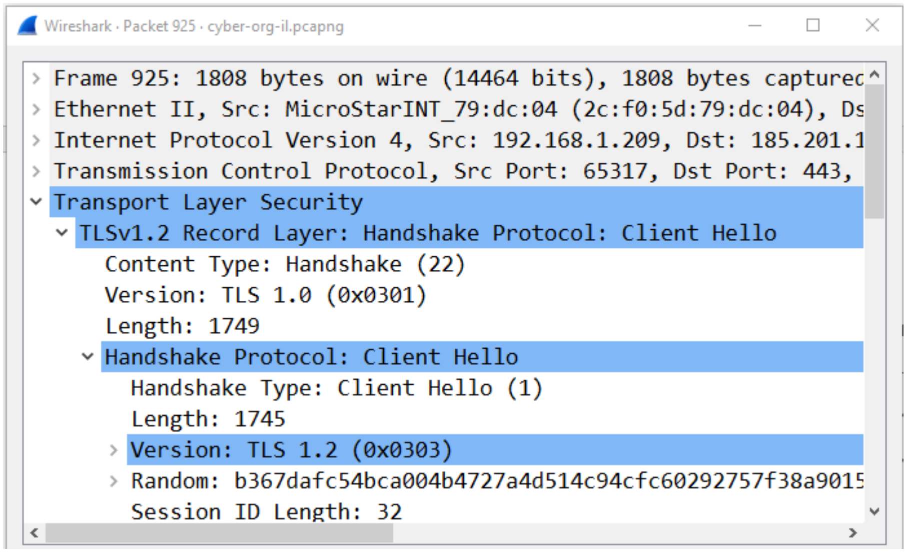
שדה זה מציין את מספר הגרסה של ה-TLS שנעשה בה שימוש.

- 0x0301 – TLS 1.0
- 0x0302 – TLS 1.1
- 0x0303 – TLS 1.2
- 0x0304 – TLS 1.3

התקשורת שאנחנו מנתחים כרגע היא TLS 1.2 ולכן הערך של השדה הוא 0x0303.

כפי שנראה מיד, לעיתים קרובות, רישום הגרסה ברשומה אינו מדויק. הסיבה לכך היא שבין השרת והלקוח יש רכיבי רשת שייתכן ונוצרו טרם תקופת TLS 1.2. רכיבים אלו עלולים להפיל פקטות שגרסת ה-TLS שלהן חדשה. לכן, התקשורת בין השרת והלקוח "מתחזה" לגרסה נמוכה יותר.

הבה ניווכח בתופעה הזו. פיתחו את פקטה 925.



בשדה ה-Version של הרשומה נכתב TLS 1.0. עם זאת, בדיקה מעמיקה יותר לתוך המידע שעובר מתחת, מראה כי אכן מדובר בגרסה 1.2.

על שדה האורך אין צורך לפרט, כי בשלב זה אתם כבר מכירים איך הוא עובד. אחריו, יופיע המידע שעובר ברשומה.

לאחר שהבנו כי השרת והלקוח מחליפים ביניהם רשומות TLS, נוכל לעבור על ה-Handshake.

TLS Handshake – מטרות

בסעיף זה אנחנו קוצרים את הפירות של כל ההכנה הארוכה שעשינו בפרקים הקודמים. כל מה שלמדנו עד כה נועד להביא אותנו לנקודה שבה נוכל להבין את הפסקאות הבאות.

תיאום Cipher Suites

בפרקים הקודמים ראינו שכדי להשיג Confidentiality, Integrity, Authentication השרת והלקוח נדרשים לכמה כלים. ראשית הם נדרשים **לאלגוריתם הצפנה סימטרי**. אלגוריתם זה יכול להיות לדוגמה AES. האלגוריתם עצמו אינו מספיק, ונדרש גם מפתח הצפנה משותף. כדי לתאם את מפתחות ההצפנה, נדרש **אלגוריתם החלפת מפתחות**. אלגוריתם זה יכול להיות RSA או DH. הבחירה באלגוריתם החלפת המפתחות קובעת מי שולח מה

ולמי. אם נבחר DH, כל צד ישלח לצד השני את החלק הגלוי שלו ביצירת הסוד המשותף. אם נבחר RSA, צד הלקוח יבחר סוד ויצפין אותו באמצעות המפתח הציבורי של השרת. אם השרת והלקוח תיאמו את האלגוריתמים הללו ביניהם, יש להם Confidentiality.

כדי שהלקוח יבטח בסרטיפיקט של השרת, וכך יתקיים ביניהם Authentication, השרת והלקוח צריכים לתאם ביניהם **אלגוריתם חתימה**. אלגוריתם זה עשוי להיות RSA שלמדנו, אך יש גם אפשרות של DSA. אלגוריתם DSA הוא קיצור של Digital Signature Algorithm. מבלי להיכנס לפרטים של DSA, אפשר להבין משמו שהוא עשוי לשמש כחלופה ל-RSA לטובת המלאכה של חתימה דיגיטלית.

השרת והלקוח צריכים גם להבטיח Integrity, כדי שאף גורם לא ישנה את הפקטות שעוברות ביניהם. הם מבצעים זאת באמצעות הוספת HMAC, Hashed Message Authentication Code, כך שנדרש לתאם ביניהם גם **אלגוריתם Hash** וגם מפתח סודי עבור ה-HMAC, מפתח שקראנו לו Hashing key.

אם כך, לפני שהשרת והלקוח יכולים להקים סוקט מאובטח הם צריכים לתאם ביניהם את ארבעת הדברים הללו:

א. מהו האלגוריתם שישמש להחלפת מפתחות ההצפנה הסימטריים?

ב. מהו אלגוריתם החתימה?

ג. מהו אלגוריתם ההצפנה הסימטרי?

ד. מהו אלגוריתם ה-Hash?

התשובה לארבע השאלות הללו היא מה שנקרא ה-**Cipher Suite** שהשרת והלקוח בחרו. לדוגמה, הצירוף

DH-RSA-AES256-SHA256

הוא צירוף של רביעיית אלגוריתמים שעשוי להבחר בתור Cipher Suite.

<u>יצירת Master Secret</u>

בנוסף לבחירת ארבעת החלקים של ה-Cipher Suite, השרת והלקוח צריכים להחליף גם ביניהם את כל המידע לטובת יצירת המפתחות שישמשו אותם בהמשך. כמה מפתחות צריך ליצור?

כאשר דנו בסוקטים בשכבת התעבורה, ראינו שסוקט מורכב משני זרמים של מידע, זרמים שאינם תלויים זה בזה. לדוגמה, כאשר לקוח פותח סוקט TCP מול השרת הוא בוחר Sequence number כלשהו (או בקיצור – SEQ). גם השרת בוחר SEQ, שאין לו קשר ל-SEQ של הלקוח. באותו האופן, הכיוון של הסוקט שבין הלקוח והשרת מאובטח על-ידי מפתחות שונים מאשר הכיוון שבין השרת והלקוח. אם כך אנחנו מגיעים למסקנה שנדרשים שני מפתחות הצפנה סימטריים, אחד לכל צד.

כמו כן, כל כיוון צריך גם Hashing key. נזכיר כי HMAC פועל בדרך הבאה: לוקחים מפתח סודי שתואם בין הצדדים, ומחברים אותו למסר שמעוניינים להעביר. מבצעים Hash על התוצאה, ומקבלים HMAC. שולחים את המסר המקורי – ללא המפתח כמובן – יחד עם ה-HMAC.

כלומר סוקט מאובטח צריך ארבעה מפתחות:

- Encryption Key לצד השרת
- Encryption Key לצד הלקוח
- Hashing Key לצד השרת
- Hashing Key לצד הלקוח

עקרונית, השרת והלקוח יכלו להשתמש באלגוריתם החלפת המפתחות (RSA או DH) כדי להעביר כל אחד מארבעת המפתחות. מעשית, כדי לייעל את התהליך, השרת והלקוח מתאמים ביניהם מפתח יחיד, שבדרך מסויימת "מתחלק" לארבעת המפתחות. התיאום של אותו מפתח יחיד, שנקרא **Master Secret**, הוא המטרה הנוספת של ה-TLS Handshake.

<u>ביצוע Authentication לצד השרת</u>

כחלק מתהליך ה-TLS Handshake, הלקוח צריך לוודא שהשרת הוא אכן מי שהוא נחזה להיות. פעולה זו מורכבת משני שלבים. בשלב הראשון השרת מעביר ללקוח סרטיפיקט, שבו הוא טוען "אני הבעלים של הדומיין שבקשת". אך זה לא מספיק, שרת יכול להוריד סרטיפיקט של דומיין אחר ולשלוח אותו. לכן בשלב השני הלקוח צריך לוודא שהשרת הוא אכן הבעלים של הסרטיפיקט שהוצג לו. כדי לעשות את זה, השרת יצטרך להוכיח ללקוח שיש לו את המפתח הפרטי שמתאים למפתח הציבורי שבסרטיפיקט. בהמשך כמובן נראה איך זה מתבצע.

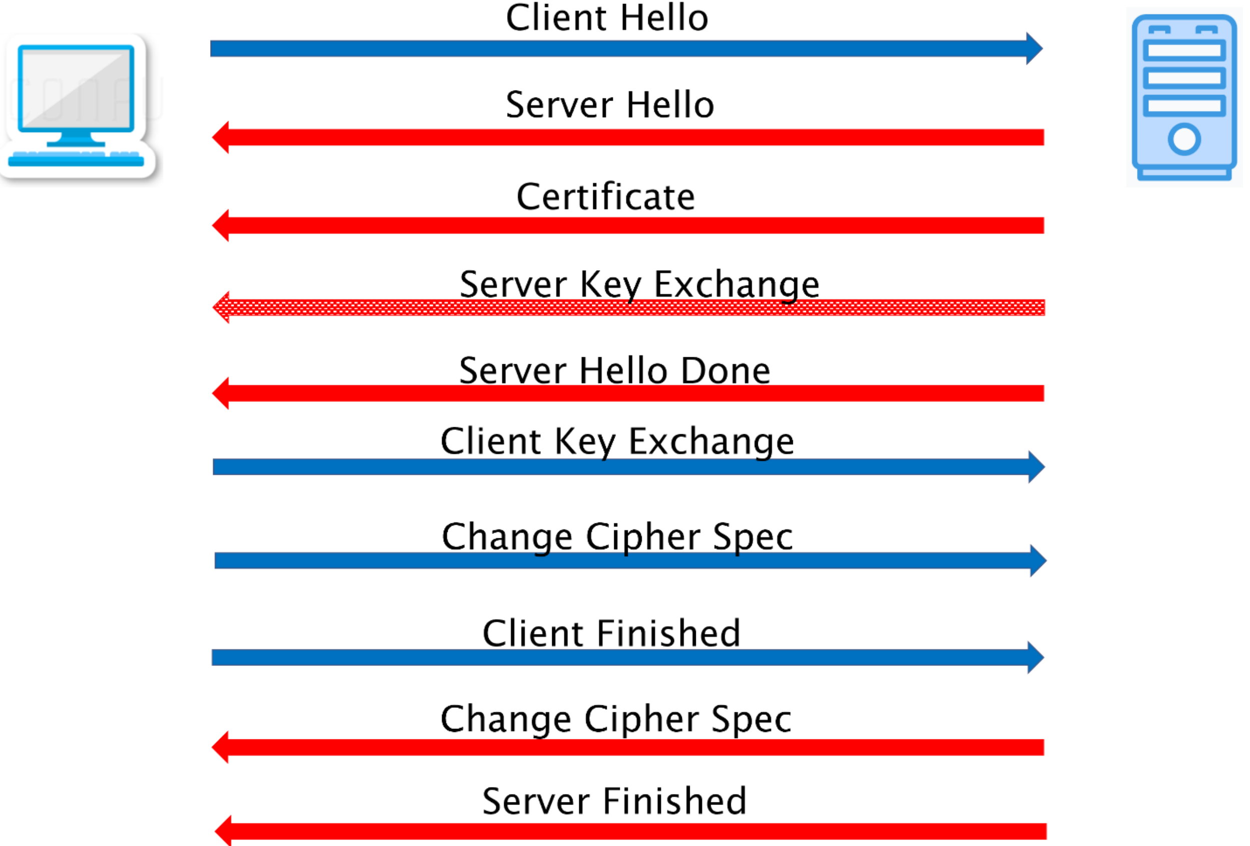
<u>מניעת MITM</u>

השרת והלקוח ערים לכך שגורם זדוני כלשהו עלול לבצע ביניהם מתקפת Man In The Middle בזמן ה-Handshake. מתקפה כזו תכלול מסרים מהונדסים כך שבסופו של דבר כל צד יפתח סוקט מאובטח מול הגורם הזדוני, שישיג את מפתחות ההצפנה ואת ה-Hashing Keys של שני הצדדים. לאחר מכן, הגורם הזדוני יתווך בין השרת והלקוח ויוכל למעשה לקרוא ואף לשנות את המידע שעובר ביניהם. לכן, הצדדים צריכים לוודא שכל צד לתקשורת קיבל בדיוק את מה שהצד השני שלח, לכל משך ה-Handshake, ללא שינוי.

Handshake – בגרסת DH

כפי שתארנו, אחד מתפקידי ה-TLS Handshake הוא לתאם את ה-Master Secret, הסוד המשותף שממנו ייגזרו כל המפתחות. לכן הגיוני שה-Handshake משתנה לפי הדרך הנבחרת להחלפת מפתחות סימטריים.

האיור הבא מתאר את רשומות ה-TLS המוחלפות בין השרת והלקוח כאשר אלגוריתם החלפת המפתחות שנבחר הוא DH. אחת הרשומות אינה קיימת בגרסת RSA של ה-Handshake, לכן היא מסומנת בצבע שונה.



Client Hello

פיתחו את פקטה 925 בקובץ ההסנפה, הכוללת את הרשומה של Client Hello.

תחילת הרשומה כוללת מספר שדות שאנחנו כבר מכירים ולכן נעבור עליהם רק בקצרה. הרשומה מתחילה בשלושת השדות סוג, גרסה ואורך, שסקרנו בתחילת הפרק. הגרסה היא TLS1.0, כזכור המטרה היא רק לעבור בשלום רכיבים ישנים שאולי נמצאים בין השרת והלקוח. לאחר מכן מתחילה הרשומה עצמה, מסוג Handshake. כיוון שלתהליך ה-Handshake עצמו יש סוגים רבים של רשומות, שמיד נכיר אותן, יש צורך לציין שזוהי רשומה מסוג

Client Hello, שסימונה בפרוטוקול הוא "1". לאחר מכן, יש שדה אורך נוסף וגרסה – הפעם, הגרסה הנכונה: TLS 1.2.

עברו לשדה ה-Cipher Suites. הרחיבו את התצוגה כדי לראות את הפרטים:

TLS 1.3

TLS 1.2

```
v Cipher Suites (16 suites)
    Cipher Suite: Reserved (GREASE) (0x3a3a)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
```

הלקוח מצהיר על כל האפשרויות שהוא תומך בהן. במקרה זה, הלקוח מסר לשרת שהוא תומך ב-16 אפשרויות. האפשרות הראשונה שמורה לניסיונות וחידושים בתקן, ואינה רלבנטית. שלוש האפשרויות הבאות שייכות לגרסת TLS 1.3, ונדון בהן בהמשך.

האפשרויות שנותרו הן מהצורה הבאה:

“TLS_X_Y_WITH_Z_W”

כאשר במקום X תבוא שיטת החלפת המפתחות הסימטריים.

Y – אלגוריתם החתימה

Z – אלגוריתם ההצפנה הסימטרית

W – אלגוריתם ה-Hash

לדוגמה:

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

החלפת מפתחות סימטריים – DH (אם לדייק, ECDHE, קיצור של Elliptic Curve Diffie Hellman Ephemeris, וריאציה של דיפי הלמן).

חתימה – RSA.

הצפנה סימטרית – AES 128 GCM. כשסקרנו הצפנות סימטריות הזכרנו שזוהי וריאציה של קוד הבלוק AES, שבה הבלוק הוא בגודל 128 ביט ונעשה שימוש ב-Galois/Counter Mode, כלומר כל בלוק מוצפן במפתח הכולל מספר סידורי, ייחודי לבלוק, ובנוסף לכך מתקיימת בדיקה שלא בוצע שינוי בפקטה. למעשה, מדובר בשילוב של Encryption ו-Integrity בו זמנית. אלגוריתם שמבצע את שני הדברים הללו בו זמנית נקרא AEAD, קיצור של Advanced Encryption with Associated Data.

אלגוריתם HASH – SHA256.

יש כמה Cipher Suites חריגים. ארבעת האחרונים חורגים מהמבנה שהוצג, ולכאורה הם כוללים רק שלושה אלגוריתמים. לדוגמה:

TLS_**RSA**_WITH_AES_128_GCM_SHA256

הסיבה לכך היא ש-RSA הוא ייחודי, בכך שיכול לשמש הן כאלגוריתם החלפת מפתחות והן כאלגוריתם חתימה. במקרה ש-RSA כתוב כך, במבנה של שלשת אלגוריתמים, הכוונה היא ש-RSA משמש בתפקיד כפול. עד כאן אודות ה-Cipher Suites.

Server Hello

השרת קיבל מהלקוח פניה, Client Hello, והוצעו לו מספר Cipher Suites לבחור מהם.

השרת בוחר לעבוד בגרסת TLS 1.2, ומודיע ללקוח מהו ה-Cipher Suite שהוא בחר (השדה התחתון):

```
˅ Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 70
    Version: TLS 1.2 (0x0303)
  › Random: 2a4310376d0f8b53d2eb78cbff686826f271dcd86b13cccee0c8d345732a1c8d
    Session ID Length: 0
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
```

בהמשך נתייחס לשדות נוספים שעוברים ב-Server Hello.

Certificates

ברשומה זו השרת מעביר ללקוח את הסרטיפיקט שלו, יחד עם סרטיפיקט של ה-ICA שחתם עליו (בהנחה שהוא לא נחתם ישירות על-ידי ה-Root CA).

ניתן לראות שני סרטיפיקטים:

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Certificate
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 2848
  ⌄ Handshake Protocol: Certificate
      Handshake Type: Certificate (11)
      Length: 2844
      Certificates Length: 2841
    ⌄ Certificates (2841 bytes)
        Certificate Length: 1545
      › Certificate […]: 30820605308204eda003020102021206 0fe9e33
        Certificate Length: 1290
      › Certificate […]: 30820506308202eea003020102021100c212324
```

נפתח את הסרטיפיקט הראשון ונמצא מי ה-Issuer שלו ומי ה-Subject שלו:

```
⌄ Certificate […]: 30820605308204eda0030201020212060fe9e33
  ⌄ signedCertificate
      version: v3 (2)
      serialNumber: 0x060fe9e33f58be8d30bc4f7845c787218a1f
    › signature (sha256WithRSAEncryption)
    ⌄ issuer: rdnSequence (0)
      › rdnSequence: 3 items (id-at-commonName=R12,id-at-or
    › validity
    ⌄ subject: rdnSequence (0)
      › rdnSequence: 1 item (id-at-commonName=cyber.org.il)
```

כפי שרואים, R12, שהוא כזכור ICA, חתם לדומיין cyber.org.il.

פיתחו את הסרטיפיקט השני ותמצאו שם את הסרטיפיקט של 12R, חתום על-ידי ה-Root CA.

כעת נחזור מעט אחורה. פיתחו את ה-Client Hello, נמצא שם את ה-Extension של ה-SCT שלמדנו עליו בפרק על סרטיפיקטים. הלקוח מבקש אותו מהשרת:

```
⌄ Extension: signed_certificate_timestamp (len=0)
    Type: signed_certificate_timestamp (18)
    Length: 0
› Extension: server_name (len=17) name=cyber.org.il
```

ה-SCT עצמו עובר בתוך הסרטיפיקט. היזכרו שפתחנו את הסרטיפיקט של cyber.org.il ומצאנו בתוכו את ה-SCT. סגרנו את המעגל!

Server Key Exchange

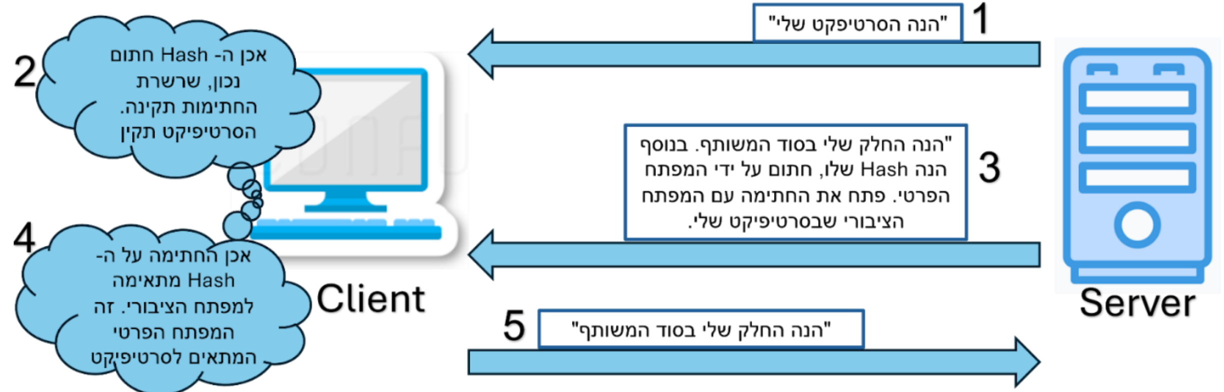
ברשומה הקודמת, השרת הכריז על בחירה באלגוריתם דיפי הלמן לטובת יצירת הסוד המשותף, הוא ה- Master Secret, שממנו כאמור ייגזרו כל המפתחות (Encryption ו-Hashing). עתה, דבר אינו מעכב את השרת להתחיל בתהליך ולשלוח ללקוח את ה-Public Key הנדרש לטובת יצירת הסוד המשותף:

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 589
  ⌄ Handshake Protocol: Server Key Exchange
      Handshake Type: Server Key Exchange (12)
      Length: 585
    ⌄ EC Diffie-Hellman Server Params
        Curve Type: named_curve (0x03)
        Named Curve: secp256r1 (0x0017)
        Pubkey Length: 65
        Pubkey: 04d6136caa0068fb47f7dad1188f377133904a4347ce82a664abe
      ⌄ Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
          Signature Hash Algorithm Hash: SHA512 (6)
          Signature Hash Algorithm Signature: RSA (1)
        Signature Length: 512
        Signature […]: 158b250af81cc6a29db74d09c29852acb6e1fbc4dffe81
```

ניתן לראות תחת "EC Diffie-Hellman Server Params" את הערך של ה-Pubkey שיצר השרת. הלקוח יוכל עכשיו לחשב באמצעותו את הסוד המשותף.

כפי שרואים בהמשך, השרת מצרף גם חתימה. חישבו, מדוע יש צורך גם בחתימה?

ניזכר במה שלמדנו על סרטיפיקטים. הלקוח חייב לוודא שהשרת הוא אכן הבעלים האמיתי של הסרטיפיקט שנשלח אליו (הרי כל אחד יכול להוריד סרטיפיקט ולשלוח אותו) והדרך שלו לעשות זאת היא באמצעות וידוא שהשרת הוא הבעלים של המפתח הפרטי הצמוד למפתח הציבורי שבסרטיפיקט. השרת מוכיח ללקוח את הבעלות על המפתח הפרטי באמצעות שליחת חתימה על ה-Pubkey. הלקוח ישתמש במפתח הציבורי של השרת, מתוך הסרטיפיקט, כדי לפתוח את החתימה ולוודא שה-Hash מתאים לזה של ה-Pubkey.



כאשר בוחנים את החתימה האחרונה, רואים כי לפניה מופיע שדה של Signature Algorithm. מדוע יש צורך בשדה זה? הרי ה-Cipher Suite כבר כולל אלגוריתמי Hash וחתימה.

נשווה אותם. בתעבורה הספציפית שאנחנו בוחנים:

הסיכום ב-Cipher Suite הוא RSA + SHA256.

החתימה על ה-Pubkey היא RSA pkcs1 + SHA512.

ובכן, התשובה מתחלקת לאלגוריתם החתימה ולאלגוריתם ה-Hash.

אלגוריתם חתימה: ב-Cipher Suite סוכם על RSA, בלי פירוט ספציפי. אלגוריתם RSA כולל סכמות שונות, שנועדו למנוע פרצות אבטחה (לדוגמה, להבטיח שהבחירה של P, Q לא תיתן מספרים קרובים מדי). כלומר השרת אומר ללקוח "אני מכבד את ההסכם בינינו לעבוד RSA, אני בוחר בסכמה הספציפית של RSA pkcs1".

אלגוריתם Hash: החתימה על ה-Pubkey היא חתימה רגישה במיוחד. היא זו שמאפשרת ללקוח לאמת את הזהות של השרת על פי הסרטיפיקט שקיבל. לכן השרת יכול להשתמש ב-Hash יותר מאובטח מאשר מה שסוכם ב-Cipher Suite.

Server Hello Done

כל המשמעות של רשומה קצרה זו היא "לקוח יקר, אני סיימתי לשלוח אליך כל מה שהייתי צריך לשלוח, בשלב זה. הכדור אצלך".

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 4
  ⌄ Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```

Client Key Exchange

הלקוח יודע בשלב זה מה ה-Cipher Suite שהשרת בחר, במקרה זה DH. הלקוח קיבל את מפתח ה-DH הציבורי של השרת וגם וידא שהשרת הוא אכן הבעלים של המפתח הציבורי בסרטיפיקט, וכעת הוא עונה עם Pubkey משלו:

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 70
  ⌄ Handshake Protocol: Client Key Exchange
      Handshake Type: Client Key Exchange (16)
      Length: 66
    ⌄ EC Diffie-Hellman Client Params
        Pubkey Length: 65
        Pubkey: 0463936e928333ee1ec381b8bc36c8e9197205d263e190a4
```

Change Cipher Spec

בשלב זה, גם השרת וגם הלקוח אמורים להיות מסוגלים לחשב את אותו Master Secret. כעת, הלקוח שולח הודעה שמשמעותה "עבור למוצפן".

```
⌄ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.2 (0x0303)
    Length: 1
    Change Cipher Spec Message
```

שימו לב, שרשומה זו היא בעלת Content Type שאינו Handshake, בניגוד לכל הרשומות עד כה. הסיבה לכך היא שרשומה זו עשויה להישלח גם תוך כדי תעבורת תקשורת לאחר ה-Handshake, אם אחד הצדדים מבקש להחליף מפתחות הצפנה.

פענוח תקשורת מוצפנת

אם נפתח את הרשומה נמצא שכתוב שם שזוהי רשומה מוצפנת, בלי פירוט:

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 40
    Handshake Protocol: Encrypted Handshake Message
```

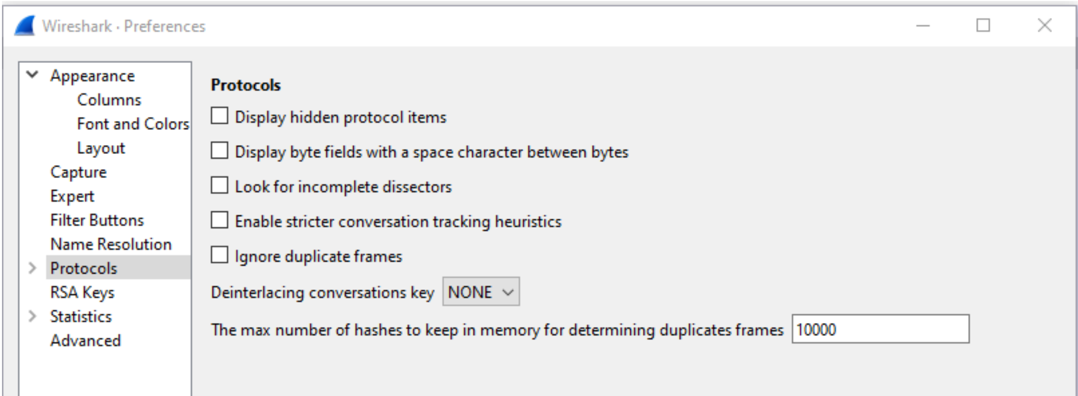
למזלנו, Wireshark יודע לקבל בתור קלט קובץ מפתחות, הכולל את ה-Master Secret של התעבורה, ולהשתמש בו כדי לפתוח את ההצפנה של כל הרשומות.

הורידו את הקובץ הבא:

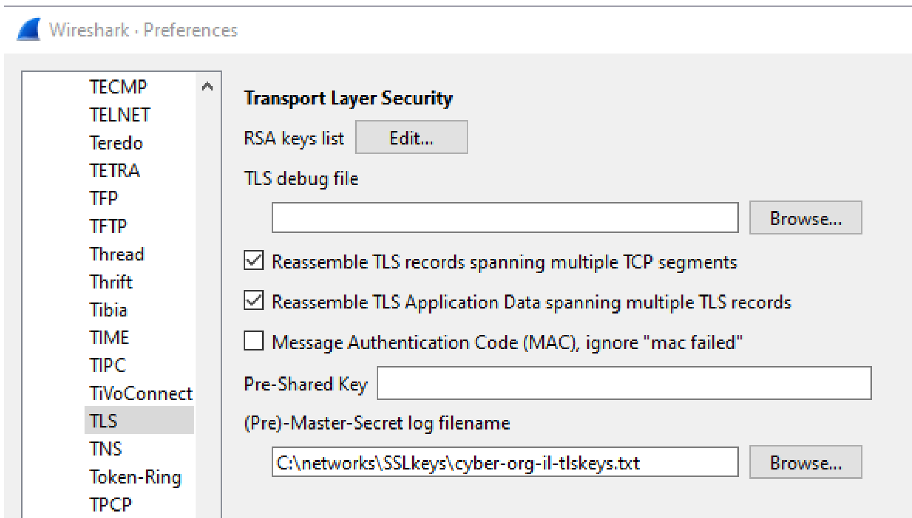
https://data.cyber.org.il/networks/cyber-org-il-tlskeys.txt

שימרו את הקובץ בתיקייה לבחירתכם. לדוגמה, C:\networks\SSLkeys.

כדי לייבא את הקובץ אל Wireshark, היכנסו לתפריט Edit ובתוכו Preferences:



שם, תחת Protocols, חפשו TLS (אם תכתבו את האות t התפריט יקפוץ לשם מיד ותחסכו זמן גלילה). בתוך TLS, הגדירו היכן נמצא קובץ המפתחות:



לאחר שתקליקו על OK, נכונה לכם הפתעה קטנה – שם הרשומה האחרונה השתנה מ- Encrypted Handshake Message ל-Client Finished.

כדי ליצור קובץ מפתחות משלכם, פעלו לפי ההוראות הבאות:

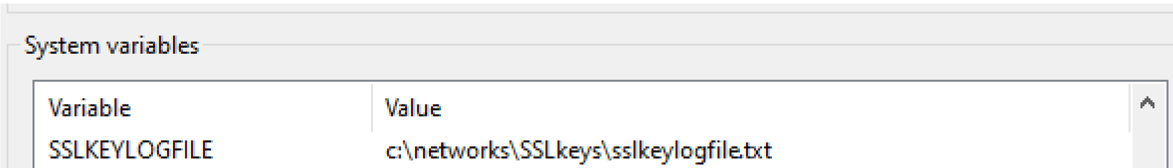
א. בתפריט החיפוש של Windows כיתבו env, והקליקו על Edit the system environment variables.

ב. בתוך חלון ה-System Properties הקליקו על Environment Variables.

ג. הוסיפו משתנה סביבה חדש בשם SSLKEYLOGFILE. ערכו של המשתנה יהיה שם הקובץ שתרצו לשמור אליו את המפתחות, כולל הנתיב המלא.

ד. בצעו אתחול למחשב ותוכלו לוודא שהקובץ נוצר.

ה. שימו לב – הקובץ ישמור את כל מפתחות הגלישה שלכם מעתה ואילך. אחרי זמן מה, הקובץ עלול לגדול מאד, מה שיגרום לכך שייקח ל-Wireshark זמן רב למצוא את המפתחות שהוא צריך. לכן, מומלץ אחת לזמן מה למחוק את הקובץ וליצור חדש במקומו. כיוון שהקובץ תפוס על-ידי מערכת ההפעלה לא תוכלו למחוק אותו סתם כך, אלא תצטרכו לשנות את משתנה הסביבה SSLKEYLOGFILE לערך חדש (שם קובץ אחר), לאתחל את המחשב, ורק אז תוכלו למחוק את קובץ המפתחות הישן.



Client Finished

בפתיחת ההסבר על ה-Handshake ראינו שאחת המטרות היא מניעת MITM. כלומר, לוודא ששום גורם אינו נמצא בין השרת ללקוח ומשנה את ההודעות ביניהם. השרת והלקוח רוצים לוודא שכל ההודעות שלהם התקבלו בדיוק כפי שהן על-ידי הצד השני.

ברשומת ה-Client Finished, הלקוח מוסר לשרת את המידע שנדרש כדי שהשרת יבדוק אם ההודעות שלו התקבלו על-ידי הלקוח בלא שינוי.

איך אפשר לוודא שאף אחד לא שינה הודעה?

הלקוח צובר את כל הרשומות שעברו עד כה בינו לבין השרת ומבצע לצבר הרשומות "ערבוב". בקרוב נפרט על הערבוב, אך בשלב זה נניח שמתקבל מספר כלשהו. את המספר הזה גם מצפינים וגם מוסיפים לו HMAC, כדי שאף אחד לא יוכל לשנות אותו.

השרת מחשב באופן עצמאי את אותו החישוב שהלקוח ביצע, הרי גם לו יש את אותן הרשומות. הוא קולט את מה שהלקוח העביר, בודק את ה-HMAC ומסיר את ההצפנה. אם המספר שחישב השרת זהה למספר שחישב הלקוח – הכל תקין.

כדי להבין את חשיבות רשומת ה-finished, נתאר מתקפה אפשרית ונראה איך הרשומה הזו מונעת אותה.

הורדת רמת אבטחה – Downgrade Attack: בוב הוא הלקוח, פונה לשרת של אליס. ביניהם נמצאת איב. בוב שלח לאליס Client Hello והציע אוסף של Cipher Suites. חלק מהם חדשים יותר, וחלק מהם מיושנים ואפשר לפרוץ אותם. לדוגמה, צופן סימטרי בשם DES (ראשי תיבות של Digital Encryption System) שקיים בפרוטוקול TLS 1.2 אך הוכרז מזמן כלא בטוח. איב אינה מעבירה לאליס את כל ההצעות של בוב, אלא מסירה את האפשרויות הבטוחות ומותירה רק אפשרויות פרוצות כגון DES.

אליס מקבלת את ה-Cipher Suites ש"טופלו" על-ידי איב מבלי לדעת על כך. אליס חושבת "וואו, בוב ממש מיושן. כל ההצעות שלו לא ממש בטוחות לשימוש. אבל אין ברירה, אם אתעקש על הצעה מאובטחת לא נצליח להקים קשר בכלל. אקח את הכי טוב שבוב מציע לי". אליס בוחרת ב-Cipher Suite חלש, שאיב מעבירה לבוב. בוב אולי מופתע שמכל ההצעות שלו אליס בחרה את הכי פחות מאובטחת, אבל זה מה שאליס בחרה, ואליס היא השרת.

איב ממשיכה להעביר את יתר הרשומות בין אליס ובוב כמו שהן. היא מסניפה אותן ולאחר מכן בודקת אפשרויות שונות עד שמצליחה לשבור את מפתח ההצפנה הסימטרי. חשוב לציין שפעולת שבירת המפתח לוקחת זמן מה גם עבור פרוטוקולי הצפנה לא בטוחים. כלומר, כאשר אליס ובוב מסיימים את ה-handshake ומעבירים את רשומת ה-finished, איב עדיין אינה יודעת לקרוא את המידע המוצפן ואין לה יכולת להחליף את רשומת ה-finished ברשומה אחרת מבלי שאליס ובוב יבחינו בכך. כזכור, הרשומה מוגנת גם באמצעות הצפנה וגם באמצעות integrity.

ברשומת ה-Client Finished ששלח בוב, הוא "ערבב" בין היתר את רשומת ה-client hello **המקורית** ששלח לאליס. כל החישוב של בוב מתבסס על תוצאה של הערבוב הזה. לעומת זאת, לאליס הגיעה רשומת Cient Hello "מטופלת" על-ידי איב, שכזכור שינתה בה את ה-Cipher Suites. אליס תחזור על החישוב שעשה בוב אך תקבל תוצאה שונה. בעקבות זאת אליס תוכל לדעת שתהליך ה-handshake בינה לבין בוב נפגע באמצעות מתקפת MITM. אליס תשלח הודעת Alert מסוג Fatal ותסיים את ההתקשרות. כך, רשומת ה-Client finished השיגה את מטרתה, מניעת MITM.

מה קורה אם התוקף מצליח ב-Downgrade Attack? הורדת גרסה מאפשרת לתוקף להשתמש בפרצות ידועות, שטופלו בגרסאות מתקדמות יותר. לדוגמה, מתקפה בשם POODLE פועלת בהצלחה על SSL גרסה 3 (כזכור, גרסה קודמת ל-TLS 1.0), ומאפשרת לתוקף לפענח מידע מוצפן. כתוצאה מכך SSL 3 הוכרז כלא בטוח לשימוש. תוקף שמצליח לשכנע את הצדדים לעבור לגרסה זו, מקבל דרך סלולה לשימוש ב-POODLE.

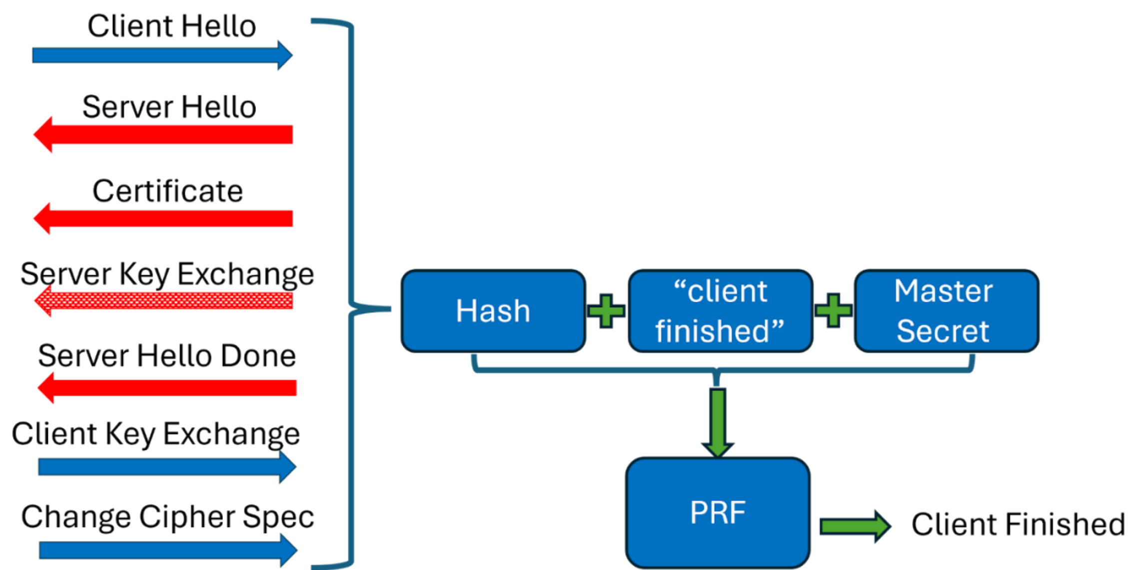
למעוניינים להרחיב, הסבר מפורט על מתקפת POODLE:

The POODLE Attack

אם כך, מהו החישוב שמבוצע ברשומת ה-Client Finished?

ראשית, מבוצע Hash על כל הרשומות שנשלחו עד כה. ה-Hash מחובר למחרוזת "Client Finished" ול- Master Secret שהלקוח חישב, שאמור כמובן להיות זהה בינו לבין השרת. חיבור כל המחרוזות הללו עובר פונקציית ערבול נוספת שנקראת PRF והתוצאה היא ה-Client Finished. אלגוריתם ה-Hash שהשתמשו בו הוא אותו אלגוריתם Hash שסוכם ב-Cipher Suites. התוצאה של ה-PRF עוברת הצפנה והוספת בדיקת Integrity לפני שהיא נשלחת אל השרת.

לטובת שלמות ההסבר, הנה פסקה אודות PRF. ניתן לדלג, או לצאת למסע חיפוש אינטרנטי, כרצונכם. PRF היא פונקציה המייצרת מספרים "כביכול אקראיים". מחשבים אינם באמת מסוגלים לייצר מספרים אקראיים, כיוון שכל החישובים המתמטיים שלהם ניתנים לשחזור. לכן, כל הפונקציות שאמורות ליצור מספרים אקראיים הן למעשה Pseudo Random Functions, או בקיצור PRF. הפונקציות הללו מסוגלות לייצר מספרים שנראים אקראיים לכאורה, אבל למעשה הם תלויים בערך התחלתי שהוזן לפונקציה. ערך זה נקרא Seed. מה שהלקוח עושה הוא לחשב Seed שמבוסס על שלושת החלקים שנסקרו, ולהעביר אותו ל-PRF, שמייצרת מספר אקראי כביכול, המועבר לשרת.



Change Cipher Spec, Server Finished

נדלג לעת עתה על הרשומה ששלח השרת, New Session Ticket. רשומה זו אינה הכרחית בתהליך ה-Handshake.

השרת עונה ללקוח ב-Change Cipher Spec מצידו, שאומר "גם אני מתחיל להצפין את התקשורת ממני אליך". הודעה זו זהה להודעה שהלקוח שלח.

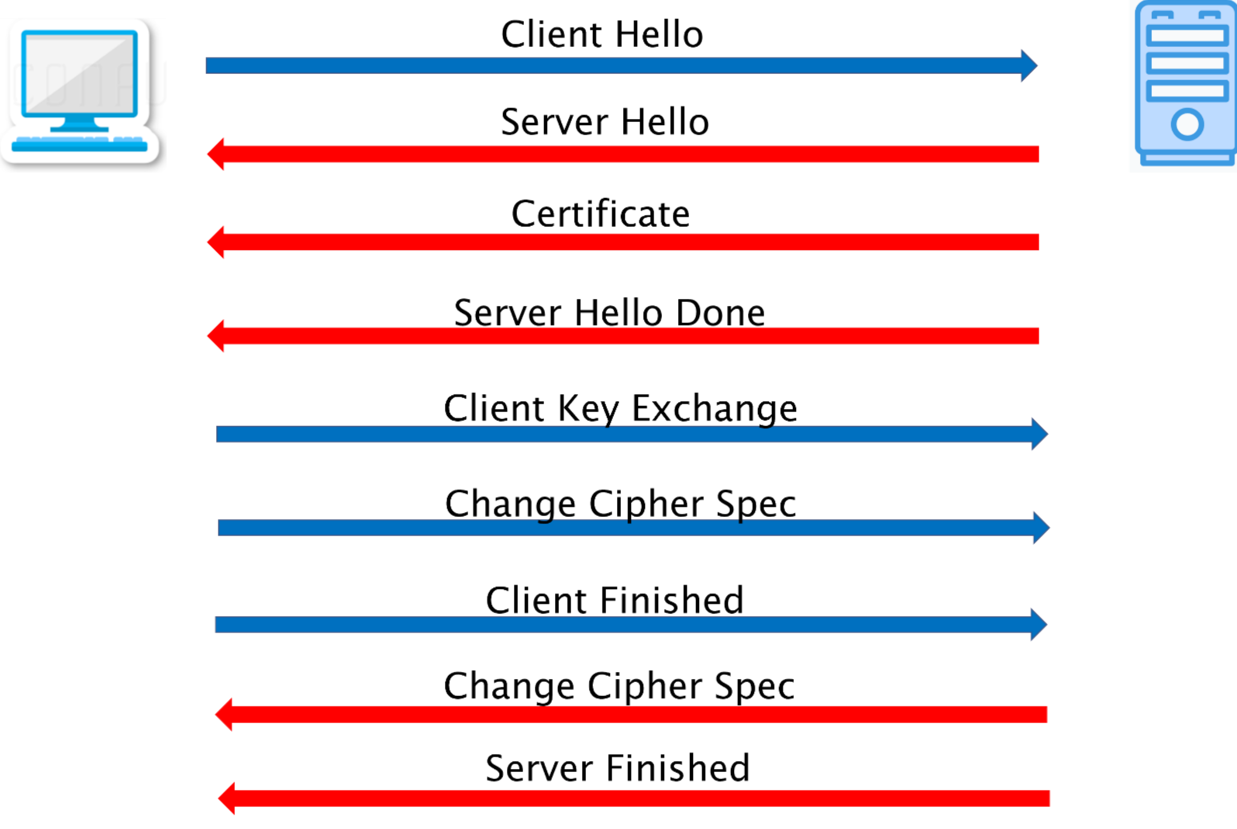
רשומת ה-Server Finished מיועדת לאפשר, הפעם ללקוח, לבדוק שאין MITM בינו לבין השרת. השרת שולח תקציר חתום של כל ההודעות בדומה למה שהלקוח שלח ב-Client Finished, אך עם שני הבדלים קלים:

- הרשומות שעליהן השרת מבצע Hash כוללות את הרשומות שנשלחו מאז ה-Client Finished ועד כה.
- המחרוזת שהשרת מוסיף ל-Hash היא מן הסתם "Server Finished".

Handshake בגרסת RSA

התמקדנו ב-Handshake בגרסת DH מכיוון שהוא נפוץ יותר. די קשה למצוא שרת שיחליף מפתחות באמצעות RSA. למעשה, בגרסה הבאה של TLS (1.3), האפשרות להחליף מפתחות באמצעות RSA כבר אינה קיימת, כיוון שהיא נחשבת פחות בטוחה. עם זאת, לטובת שלמות ההסבר, הנה סקירה של RSA Handshake.

החלפת מפתחות בגרסת RSA כוללת את כל הרשומות שאנחנו מכירים, אך אין צורך ברשומה של Server Key Exchange. הסיבה לכך היא שמי שבוחר את הסוד המשותף הוא רק הלקוח. ברשומת ה-Client Key Exchange הלקוח ישלח לשרת את הסוד כשהוא מוצפן באמצעות המפתח הציבורי של השרת.



הפעולה הזו, כפי שכבר תיארנו בפרק על סרטיפיקטים, לא רק מחליפה את המפתח של הלקוח עם השרת אלא גם מאפשרת ללקוח לבדוק שהשרת הוא אכן הבעלים של המפתח הפרטי המתאים לסרטיפיקט. רק הבעלים של המפתח הפרטי יצליח לחלץ את הסוד שהלקוח בחר.

תרגיל מודרך פילטור RSA Handshake

הורידו את קובץ ההסנפה הבא:

https://data.cyber.org.il/networks/RSA_handshake.pcapng

המשימה שלכם היא לפלטר רק TLS Handshake שמשתמשים ב-RSA כשיטת החלפת מפתחות. חישבו – איך עושים זאת?

נזכור, שמי שבוחר את שיטת החלפת המפתחות הוא השרת. המקום שהשרת מכריז בו על הבחירה שלו היא רשומת ה-Server Hello. לכן עלינו לחפש פקטות שיש בהן Server Hello. היכנסו ל-Server Hello כלשהו ומיצאו איזה שדה קובע את סוג הרשומה.

התשובה –

שדה ה-type בתוך tls.handshake. קיבעו את הערך המתאים כדי לפלטר Server Hello.

אם הצלחתם, קיבלתם רשימה של כל רשומות ה-Server Hello.

כעת, איך אפשר לפלטר רק רשומות בהן השרת בחר את RSA כאלגוריתם החלפת מפתחות?

נזכור, כי כל ה-Cipher Suites נמצאים ב-Client Hello, כולל המזהים המספריים שלהם בתקן TLS. פתיחת ה-Cipher Suites מראה לנו כי RSA משמש להחלפת מפתחות בארבע אפשרויות, המסומנות בערכים 0x002f, 0x0035, 0x009c, 0x009d.

דייקו את הפילטר שלכם והגיעו אל ה-Handshake הנדרש.

תשובה:

```
tls.handshake.type == 2 and (tls.handshake.ciphersuite == 0x002f or
 tls.handshake.ciphersuite == 0x0035 or
 tls.handshake.ciphersuite == 0x009c or
 tls.handshake.ciphersuite == 0x009d)
```

לאחר שמצאתם את הפקטה עם ה-Server Hello, בצעו Follow TCP stream והוסיפו פילטור לפי סוג פרוטוקול, TLSv1.2.

התוצאה:

RSA_handshake.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.stream eq 141 and _ws.col.protocol == TLSv1.2

No.	Time	Source	Destinat	Protocol	Lengt	Info
27…	15.929875	192.1…	69.17…	TLSv1.2	424	Client Hello (SNI=pixel.rubiconproject.com)
28…	15.993470	69.17…	192.1…	TLSv1.2	1454	Server Hello
28…	15.994682	69.17…	192.1…	TLSv1.2	365	Certificate, Server Hello Done
28…	15.994949	192.1…	69.17…	TLSv1.2	372	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
29…	16.059985	69.17…	192.1…	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
29…	16.060401	192.1…	69.17…	TLSv1.2	1466	Application Data
29…	16.125069	69.17…	192.1…	TLSv1.2	1367	Application Data

כפי שרואים, אין רשומה של Server Key Exchange.

פיתחו את רשומת ה-Client Key Exchange, ודאו כי הסוד המשותף מוצפן באמצעות מפתח RSA:

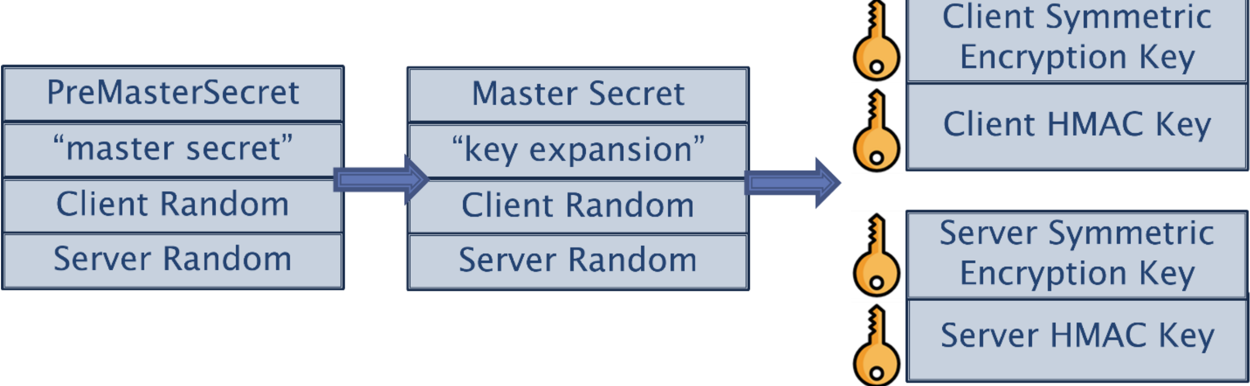
```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 262
  ⌄ Handshake Protocol: Client Key Exchange
      Handshake Type: Client Key Exchange (16)
      Length: 258
    › RSA Encrypted PreMaster Secret
```

שלבי יצירת מפתחות

במהלך סקירת ה-Handshake למדנו שהשרת והלקוח מנסים לתאם ביניהם מפתח שנקרא Master Secret, ושממנו נגזרים יתר המפתחות. כעת נתעמק בתהליך.

הסוד המשותף שהשרת והלקוח מתאמים ביניהם, בין באמצעות DH או RSA, נקרא Pre Master Secret. היזכרו בכך, שכאשר ביצענו יבוא של קובץ מפתחות לתוך Wireshark, התבקשנו להזין שם קובץ שכולל Pre Master Secret.

כלומר, נקודת ההתחלה של תהליך יצירת המפתחות היא שהצדדים תיאמו Pre Master Secret.



אל ה-Pre Master Secret מתווספת המחרוזת "master secret" ועוד שני מספרים.

כאשר סקרנו את הרשומות הללו דילגנו על ה-Random, כעת אנחנו חוזרים למבט נוסף. פיתחו את ה- Client Hello ואת ה-Server Hello של הסנפת cyber.org.il ומצאו אותם:

```
⌄ Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 1745
  › Version: TLS 1.2 (0x0303)
  ⌄ Random: b367dafc54bca004b4727a4d514c94cfc60292757f38a9015b0b52617d7822ae
      GMT Unix Time: May 19, 2065 06:05:32.000000000 Jerusalem Daylight Time
      Random Bytes: 54bca004b4727a4d514c94cfc60292757f38a9015b0b52617d7822ae
```

```
⌄ Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 70
    Version: TLS 1.2 (0x0303)
  ⌄ Random: 2a4310376d0f8b53d2eb78cbff686826f271dcd86b13cccee0c8d345732a1c8d
      GMT Unix Time: Jun 20, 1992 14:02:15.000000000 Jerusalem Daylight Time
      Random Bytes: 6d0f8b53d2eb78cbff686826f271dcd86b13cccee0c8d345732a1c8d
```

לפי התקן, ארבעת הבתים הראשונים של ה-Random אמורים להיות מבוססים על התאריך, וזאת כדי למנוע שימוש חוזר ב-Random, שיקל על מתקפות. למעשה, פרט זה אינו נאכף והן השרת והן הלקוח שולחים זמנים אקראיים (השרת לא באמת נמצא ב-1992, והלקוח לא באמת נמצא ב-2065...).

ארבעת החלקים שנסקרו עוברים דרך PRF – אותה פונקציה כאילו רנדומלית שסקרנו כשעברנו על רשומות ה-Finished – והתוצאה היא ה-Master Secret.

ה-Master Secret עצמו עדיין אינו סוף הדרך. הוא מתחבר למחרוזת "Key Expansion" ושוב ל-Random של השרת והלקוח, התוצאה עוברת דרך PRF נוסף, ואז נחתכת לארבעה חלקים.

ההסבר על ארבעת החלקים נתון בפתיחת הפרק, כשסקרנו את ה-Master Secret והמפתחות הנגזרים ממנו.

כעת, כשאנו מכירים את ה-Random, אפשר לחזור אל קובץ המפתחות SSLKEYLOGFILE. בקובץ שמורים מפתחות רבים, ו-Wireshark צריך למצוא את ה-Master Secret של סוקט ספציפי. בקובץ המפתחות כל מפתח כתוב ליד ה-Client Random.

שדה ה-SNI

פיתחו את ה-Client Hello ושימו לב לשדה **Server Name Indication**, או בקיצור SNI.

```
⌄ Extension: server_name (len=17) name=cyber.org.il
    Type: server_name (0)
    Length: 17
  ⌄ Server Name Indication extension
      Server Name list length: 15
      Server Name Type: host_name (0)
      Server Name length: 12
      Server Name: cyber.org.il
```

זהו שדה שמעניין מאוד להבין אותו ואת המשמעות שלו. מדוע בעצם יש צורך בשדה הזה? למה הלקוח צריך לציין את שם הדומיין שאליו הוא פונה?

התשובה היא שמרבית הדומיינים בעולם נמצאים על שרתי אחסון משותפים. שרתי אחסון יכולים להכיל דומיינים שונים. חישבו לדוגמה שהשרת המאחסן את cyber.org.il מאחסן גם example.com. מנקודת מבטו של הלקוח, הוא ביצע שאילתת DNS ומצא את כתובת ה-IP של השרת של cyber.orgl.il. מנקודת מבטו של שרת האחסון, כל הפקטות שהלקוח שלח עד כה לשרת היו רק הקמת קישור TCP עם הפורט המתאים, אותו הפורט 443 משרת את כל הדומיינים שהוא מאחסן. הלקוח עדיין לא מסר לשרת האחסון עם איזה דומיין הוא רוצה לתקשר. אם כך, איך שרת האחסון ידע איזה סרטיפיקט להעביר ללקוח? את של cyber או את של example?

נבחן את הדברים בהיבט פרטיות וצנזורה. שם הדומיין שהלקוח פונה אליו עובר בצורה גלויה לחלוטין. כל גורם בדרך יכול לדעת למי הלקוח גולש, ועל סמך השדה הזה ניתן גם לצנזר את הדומיין. ISP יכול לדוגמה לקבל הנחיה שלא להעביר Client Hello שמיועדים ל-cyber.org.il, מה שיחסום את הגישה לדומיין. מה שהלקוח היה רוצה לעשות, הוא שהמידע על איזה דומיין הוא ניגש אליו לא יהיה גלוי. הלקוח היה רוצה להקים קישור TLS עם שרת האחסון, לסיים איתו Handshake מוצפן, ואז להעביר את המידע על הדומיין המבוקש בצורה מוצפנת. הבעיה היא שבתהליך הקמת הקישור, השרת חייב לשלוח סרטיפיקט, והסרטיפיקט משוייך לדומיין ספציפי.

זוהי אחת הבעיות הפתוחות כרגע ב-TLS. ניתן לעקוף אותה באמצעות שימוש ב-VPN – שדה ה-SNI הגלוי יהיה של שירות ה-VPN ולא של הדומיין הסופי שניגשים אליו.

פתרון עתידי לבעיה הוא ECH, קיצור של Encrypted Client Hello. הרעיון באופן כללי הוא לשנות את פרוטוקול DNS, כך שיכלול לא רק את ה-IP אלא גם את ה-Public Key של **השרת המאחסן**. הלקוח ישתמש בו כדי להצפין את ה-Client Hello כולו, או אפילו רק את שדה ה-SNI. השרת המאחסן יוכל לפתוח את ה-SNI ולראות לדוגמה שהלקוח ביקש את cyber.org.il. משם ה-Handshake ימשיך כרגיל.

RTT

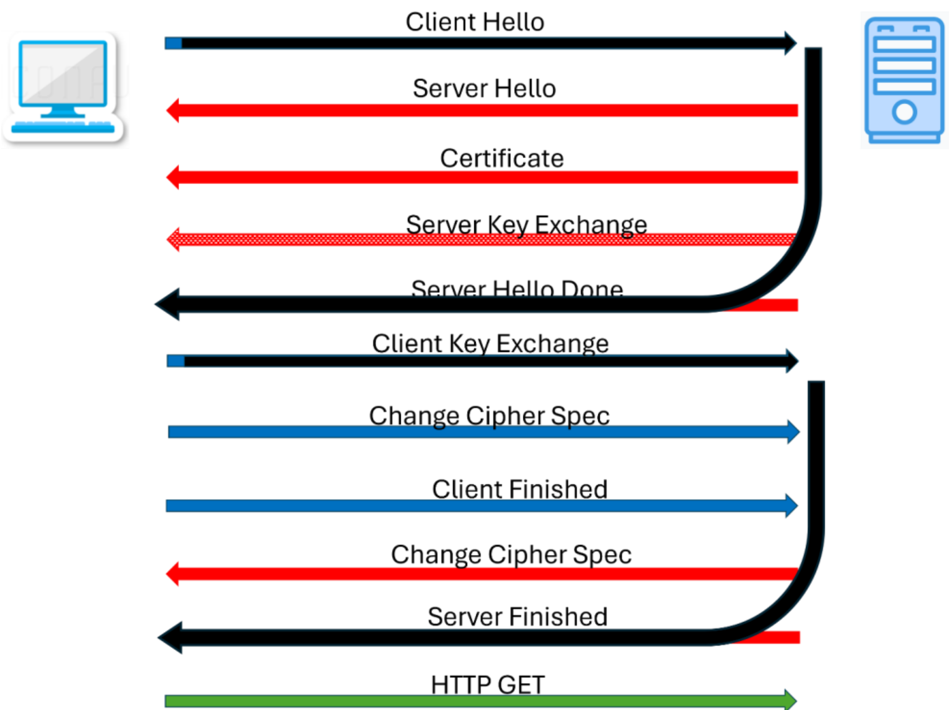
לפני שנעבור לנושא הבא, תהליך Handshake מקוצר, עלינו להבין נושא מרכזי בתהליך ה-Handshake והוא Round Trip Time, או בקיצור RTT.

RTT הוא פרק הזמן שלוקח מרגע שצד אחד לתקשורת שולח פקטה ועד שהתשובה עליה מתקבלת. דוגמה פשוטה לכך היא פינג, פרוטוקול ICMP שלמדנו. הלקוח שולח פינג לשרת ומודד כמה זמן לוקח עד שהתקבלה תשובה.

כיוון שפרק הזמן בין בקשה לתשובה משתנה בין קישור לקישור, תלוי במרחק שבין השרת והלקוח ובכמות הרכיבים ביניהם, מודדים מהירות של פרוטוקולים באמצעות יחידות של RTT. לא, אין לנו יכולת לנבא מראש כמה זמן ייקח לסיים תהליך Handshake של TLS, אבל אנחנו כן יכולים להגיד כמה RTT יש בין תחילת ה-Handshake לסופו. מה דעתכם, כמה ישנם?

לטובת המספור, חשוב להבין שמספר רשומות שנשלחות בפקטה אחת אינן משנות את ה-RTT (כן, אנחנו מניחים שרוב הזמן הוא בהעברת הפקטות, לכן פקטה ארוכה תיקח בקירוב אותו זמן כמו פקטה קצרה). יתרה מכך, מספר פקטות שנשלחות אחת אחרי השניה, בלי שהגורם השולח צריך להמתין לתשובה, גם לא יעלו את ה-RTT. אם אין המתנה לתשובה מהצד השני, אפשר להתייחס לכל כמות של פקטות בתור פקטה אחת ארוכה.

אם כך, נספור כמה RTTים יש מרגע שהלקוח שולח את הפקטה הראשונה ב-TLS, עד שהלקוח יכול לשלוח את הפקטה הראשונה של פרוטוקול שכבת האפליקציה, לדוגמה – בקשת HTTP GET.



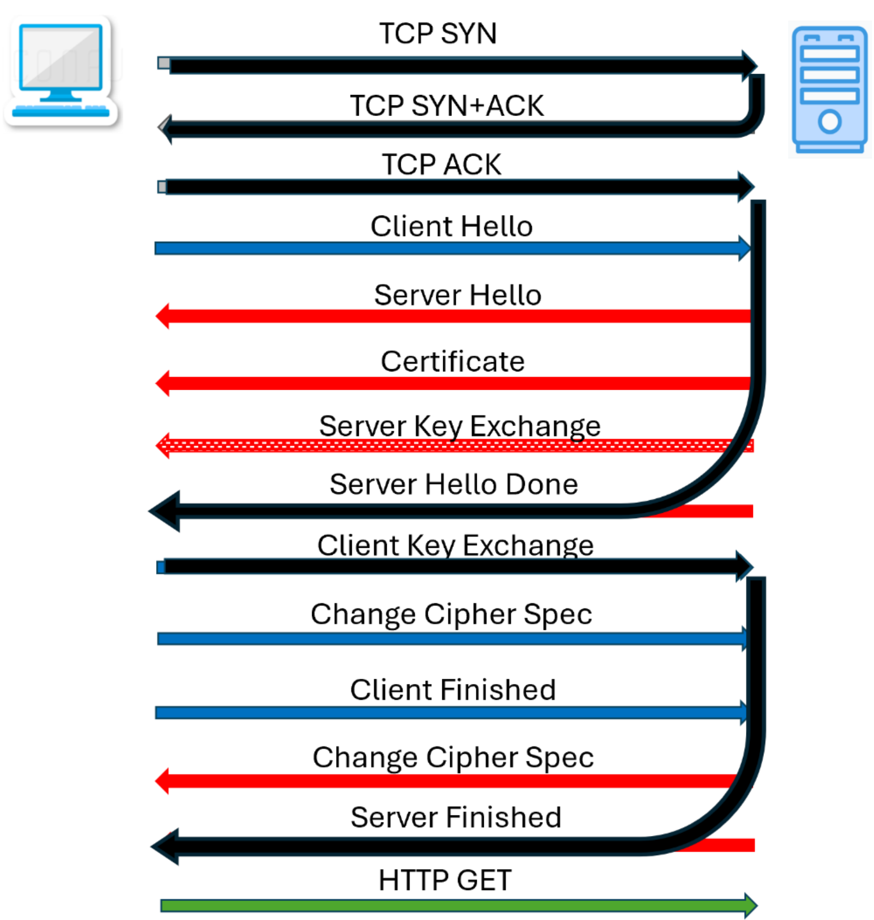
הלקוח שולח Client Hello ונאלץ להמתין עד ה-Server Hello Done. זהו RTT אחד.

הלקוח שולח Client Key Exchange ורשומות נוספות, ונאלץ להמתין עד ה-Server Finished. זהו RTT נוסף.

כלומר, לאחר שני RTT הלקוח יכול להתחיל בשליחת HTTP GET.

לכן ה-TLS Handshake של גרסה 1.2 נחשב "2-RTT".

חשוב להדגיש, שהספירה של ה-RTTים של TLS לוקחת בחשבון רק את הרשומות של TLS. כאשר TLS עובר מעל TCP, מתרחש לפני תהליך ה-Three Way Handshake של TCP. תהליך זה מוסיף כמובן RTT נוסף:



תהליך ה-Three Way Handshake מוסיף RTT אחד, למרות שהוא שלוש פקטות, מכיוון שהלקוח אינו צריך לעשות הפסקה בין השליחה של הפקטה האחרונה, ה-ACK, לבין ה-Client Hello.

אם כך, מדוע מודדים בנפרד את ה-RTT של TLS ושל TCP? האם לא היה מובן יותר לחבר אותם יחד ולומר "תהליך ה-Handshake של TLS בגרסה 1.2 הוא 3-RTT, מתחילת יצירת הקשר ועד שעובר מידע של שכבת האפליקציה"?

ובכן יש סיבה טובה מדוע מפרידים את ה-RTT של TLS מה-RTT של TCP. כאשר נלמד על פרוטוקול QUIC נבין זאת. בינתיים, נסקור שינויים ושיפורים ב-RTT של TLS, גם בגרסה 1.2 וגם בהמשך בגרסה 1.3.

Session Resumption

כפי שראינו, תהליך ה-Handshake הוא 2-RTT. לעיתים אין ברירה אלא לבצע אותו, אבל מה אם כרגע סיימנו גלישה לאתר כלשהו, סגרנו את הדפדפן ואז החלטנו להיכנס שנית?

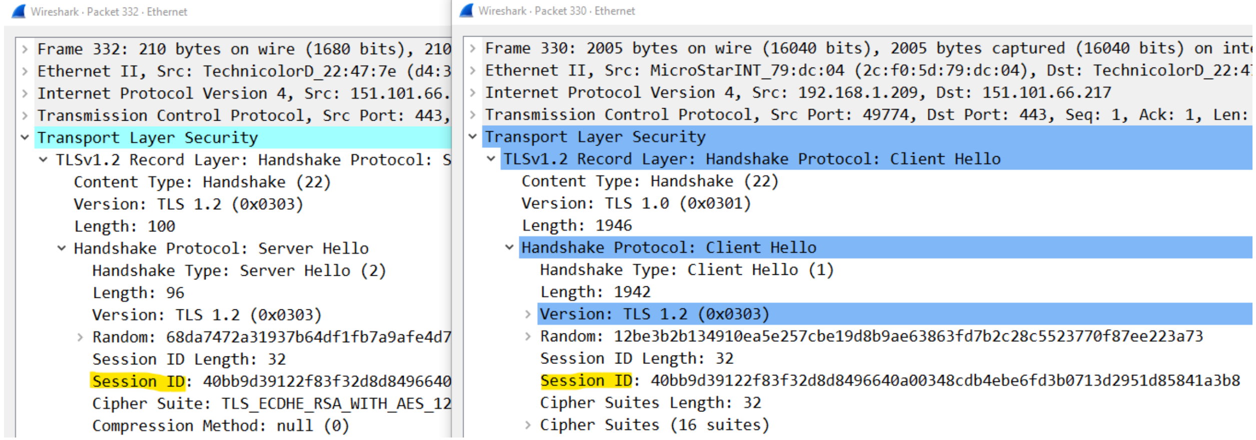
פיתחו את ה-Client Hello ואת ה-Server Hello ותמצאו שם שדה Session ID. שדה ה-Session ID מסייע לשרת וללקוח להקים סוקט מאובטח מהיר תוך דילוג על חלק משלבי ה-Handshake.

בתקשורת הראשונה אי פעם עם השרת, הלקוח יכניס ערך 0 ל-Session ID. השרת יענה לו עם Session ID כלשהו, ייחודי ללקוח זה.

בפעם הבאה שהלקוח יפנה לשרת, הלקוח ישלח לשרת את ה-Session ID שהשרת סיפק לו. כעת יש שתי אפשרויות. השרת יכול להגיד "אני לא זוכר אותך, או שה-Session ID שלך ישן מדי. בוא נתחיל Handshake מחדש". במקרה זה, השרת ישלח Session ID שונה מאשר מה שהלקוח שלח לו, ויבוצע Handshake מלא. לחלופין, השרת יכול להגיד "היי! ברוך הבא שוב. בוא נקצר עניינים" ולהשיב ללקוח עם אותו Session ID שהלקוח שלח לו ב-Client Hello.

בהסנפה שלנו מול www.cyber.org.il, הלקוח הציע לשרת Session ID קודם ביניהם, אך השרת בחר להתעלם מהאפשרות להאיץ את התהליך. אם תרצו לראות תהליך מקוצר, בצעו גלישה לאתר כרצונכם ועשו refresh לדפדפן (F5).

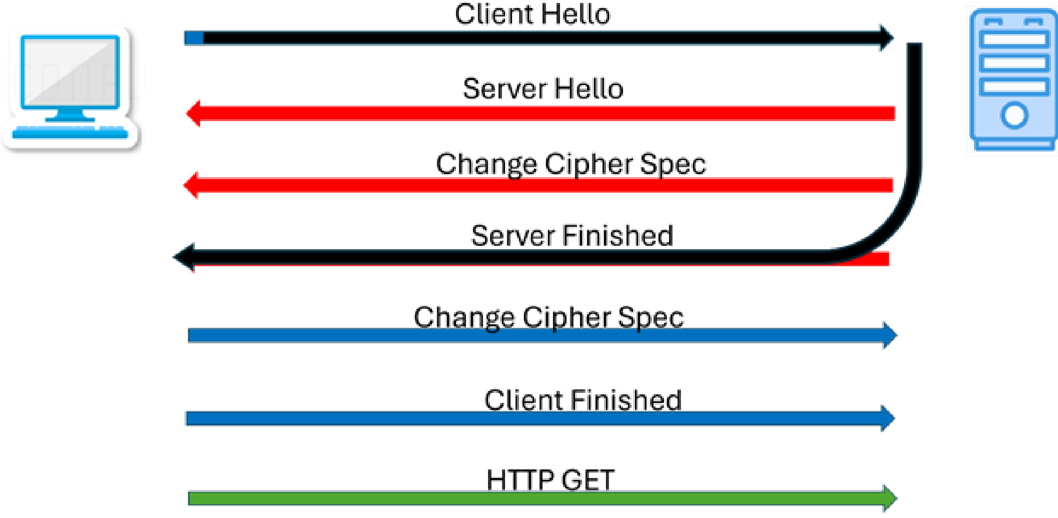
אם תפתחו את ה-Session ID של ה-Client Hello ושל תשובת השרת, ה-Server Hello, תוכלו לוודא שיש להן את אותו ערך (0x40bb9d...):



בעקבות תשובת השרת שענה עם אותו ID, המשך התהליך היה שונה וקצר יותר:

No.	Time	Source	Destinat	Protocol	Length	Info
330	4.316467	192.1…	151.1…	TLSv1.2	2005	Client Hello (SNI=www.themarker.com)
332	4.374574	151.1…	192.1…	TLSv1.2	210	Server Hello, Change Cipher Spec, Finished
333	4.374848	192.1…	151.1…	TLSv1.2	105	Change Cipher Spec, Finished

רואים שכמות הרשומות שהוחלפו היא קטנה יותר (לדוגמה, השרת לא העביר סרטיפיקט ללקוח). נבדוק מה קרה ל-RTT.



הלקוח שלח Client Hello וחיכה עד ה-Server Finished. RTT אחד.

הלקוח שלח רשומות נוספות, אך לא חיכה לתשובת השרת וכבר יכול היה לשלוח את ה-HTTP GET. במילים אחרות, מרגע שהחל את התהליך ועד שיכול היה לשלוח HTTP GET, הלקוח המתין RTT אחד.

חסכנו RTT.

Session Ticket

הרעיון של Session Resumption הוא מצוין, אבל המימוש על-ידי Session ID הוא בזבזני מבחינת השרת. שימוש ב-Session ID מאלץ את השרת להחזיק בזיכרון את כל ה-Session ID של לקוחות שסיימו התחברות אליו ואת כל המידע שצריך בשביל יצירת המפתחות מחדש: ה-Master Secret ושני ה-Randomים. בלעדיהם, השרת ייאלץ לבצע תאום חדש של סוד משותף.

העניין הוא ששרתים אינם אוהבים להחזיק אצלם מידע של לקוחות. זה מעמיס על השרת וגם מאפשר ניצול לרעה – לדוגמה, גורם זדוני עלול להעמיס את השרת בכמות גדולה של Session ID ומפתחות שצריך לשמור. שרתים מעדיפים שהלקוח יעמיס על עצמו את שמירת המידע הנדרש.

פיתחו את ה-Client Hello וחפשו שם את ה-Session Ticket.

```
⌄ Extension: session_ticket (len=192)
    Type: session_ticket (35)
    Length: 192
    Session Ticket […]: d5fce2961f9af00c2ecbb4f723fc9d050aebf
```

ה-Session Ticket כולל:

- פרק זמן שבו הוא בתוקף
- גרסת ה-TLS שתואמת בין הצדדים
- ה-Cipher Suite שתואם
- ה-Master Secret שתואם
- ה-Randomים של השרת והלקוח

רגע, אם שולחים את כל המידע הזה בגלוי, כל אחד יכול לפענח את התקשורת בין השרת והלקוח. מה נעשה? הנה החלק הנחמד – כל המידע מוצפן באמצעות המפתח הציבורי של השרת, כך שרק השרת יכול לפענח אותו.

מהיכן מגיע ה-Session Ticket?

כעת נחבר את החלק האחרון בפאזל של תהליך ה-TLS 1.2 Handshake. חיזרו אל קובץ ההסנפה של cyber.org.il. דילגנו על רשומה אחת ששלח השרת ללקוח – New Session Ticket.

cyber-org-il.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.stream eq 20 and _ws.col.protocol == TLSv1.2

No.	Time	Source	Destinat	Protoc	Lengt	Info
925	8.089212	192.1…	185.2…	TLSv…	1808	Client Hello (SNI=cyber.org.il)
930	8.102575	185.2…	192.1…	TLSv…	1414	Server Hello
933	8.102658	185.2…	192.1…	TLSv…	869	Certificate, Server Key Exchange, Server Hello Done
934	8.103856	192.1…	185.2…	TLSv…	180	Client Key Exchange, Change Cipher Spec, Finished
938	8.110257	185.2…	192.1…	TLSv…	312	New Session Ticket, Change Cipher Spec, Finished

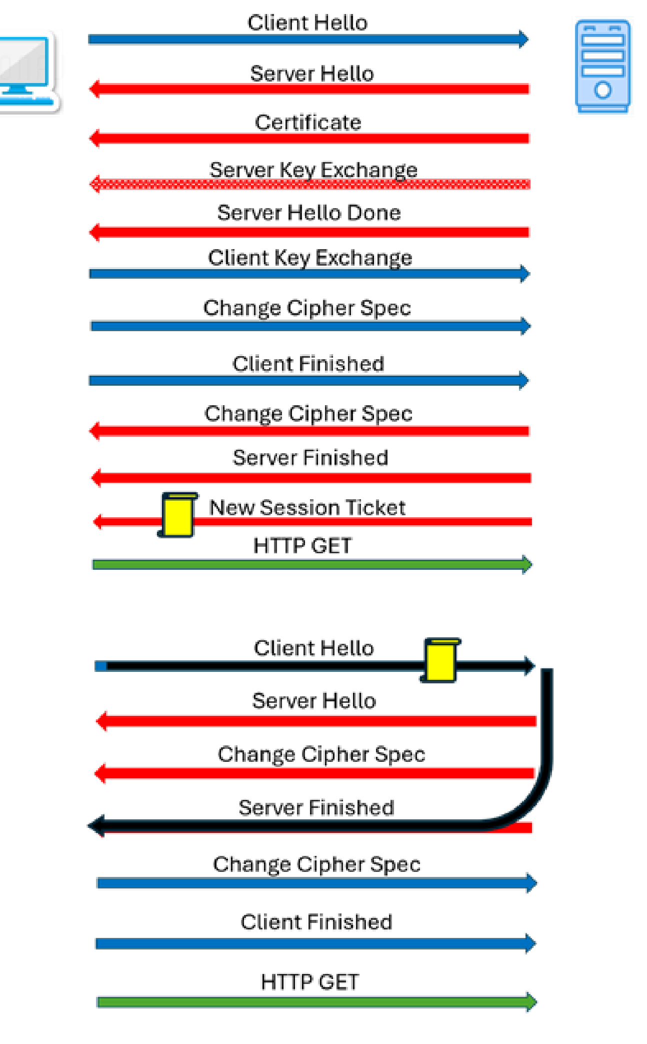
ברשומה האחרונה לפני שהשרת מתחיל להצפין, הוא שולח ללקוח את הפרטים הנדרשים, למקרה שהלקוח ירצה לחדש תקשורת מולו בזמן הקרוב:

```
˅ Handshake Protocol: New Session Ticket
    Handshake Type: New Session Ticket (4)
    Length: 198
  ˅ TLS Session Ticket
      Session Ticket Lifetime Hint: 300 seconds (5 minutes)
      Session Ticket Length: 192
      Session Ticket […]: 50c2c995532766015f74b6a8f55ba91db2f
```

כעת הפרטים שמורים אצל הלקוח והשרת אינו נדרש לשמור אצלו דבר לאחר סיום ההתקשרות.

תהליך ה-Handshake נראה דומה מאד לתהליך שמתרחש ב-Session Resumption, פרט לכך שהלקוח משתמש ב-Extension בשם Session Ticket. במידה והשרת מקבל את בקשת הלקוח לעשות את קיצור הדרך, השרת יוסיף ל-Server Hello גם Extension בשם Session Ticket, שמאשר את הבקשה.

הנה כך נראה התהליך, כולל שימוש ב-Session Ticket. תחילה, תהליך Handshake רגיל של 2-RTT. השרת מעביר ללקוח New Session Ticket. בהתקשרות הבאה, הלקוח משתמש בו ומתקיים 1-RTT Handshake:



1-RTT TLS 1.2 Session Ticket

20.1 תרגיל פענוח הסנפת TLS

הורידו את הקובץ הבא:

https://data.cyber.org.il/networks/TLS12files.rar

תחת הקובץ נמצאים קובץ הסנפה וקובץ מפתחות.

ענו על השאלות הבאות:

א. כמה פקטות מכילות Client Hello? השתמשו בפילטר כדי להשאיר רק Client Hello.

ב. התמקדו בפקטה הראשונה שמכילה Client Hello. לאיזה דומיין מבוצעת הגלישה?

ג. צרו פילטר שמפלטר רק Client Hello אל הדומיין הספציפי שמצאתם בסעיף הקודם.

ד. פלטרו את כל הרשומות בתהליך ה-Handshake שהוא ההמשך של ה-Client Hello מהסעיף הקודם. איך ניתן, בהתבסס על שמות וסדר הרשומות שנשלחות בלבד ובלי לבדוק את ה-Server Hello, לדעת אם ה-Handshake הוא DH או RSA?

ה. מהו ה-Cipher Suite שנבחר על-ידי השרת? איזה אלגוריתם נבחר עבור:

a. Authentication

b. Symmetric Encryption

c. Hashing

d. Key Exchange

ו. כמה סרטיפיקטים נשלחו על-ידי השרת? מה ה-Common Names של הבעלים של הסרטיפיקטים הללו?

ז. האם הסרטיפיקט ששלח השרת תקף לכל שרת תחת הדומיין, או לשרת ספציפי?

ח. בין אילו תאריכים הסרטיפיקט בתוקף?

ט. מיהו ה-Root CA? האם הסרטיפיקט שלו נשלח?

י. הוסיפו את קובץ המפתחות. איזה משאב הלקוח מבקש בבקשת ה-GET הראשונה שלו? מהו ה- Status Code המוחזר מהשרת?

יא. מהו שם המשתמש והסיסמה שנשלחו מהלקוח אל השרת? לאחר שביצעתם Follow TCP Stream חפשו את המחרוזות Username, Password.

סיכום

התחלנו את הפרק בכך שהתקשורת בין השרת והלקוח עוברת מעל רשומות. סקרנו את המבנה של רשומה ואת סוגי הרשומות הקיימות.

מכאן התמקדנו ברשומות של Handshake. הבנו כי הדבר הראשון שהשרת והלקוח צריכים לתאם הוא ה- Cipher Suite. בחירה זו כוללת את השיטה הנבחרת להחלפת מפתחות סימטריים. ראינו כי ל-DH ול-RSA יש שתי גרסאות שונות במקצת של Handshake.

סקרנו כיצד לאחר בחירת שיטת החלפת המפתחות הצדדים יוצרים סוד משותף, ה-Pre Master Secret, ממנו נגזר ה-Master Secret, שנגזרים ממנו ארבעת המפתחות שנעשה בהם שימוש בסוקט המאובטח: מפתח הצפנה סימטרי לשרת, מפתח הצפנה סימטרי ללקוח, Hashing key לשרת, Hashing key ללקוח.

ראינו כיצד באמצעות ה-Finished השרת והלקוח מוודאים שאף גורם לא התערב ביניהם ושיחק ברשומות.

לבסוף, ניתחנו את כמות ה-RTT בתהליך ה-Handshake של גרסה 1.2. ראינו כי נדרש 2-RTT. ראינו כי באמצעות Session Ticket אפשר לקצר את התהליך ל-1-RTT.

על הדרך, ביצענו יצירה של קובץ מפתחות SSLKEYLOGFILE וייבאנו אותו לתוך Wireshark.

בפרק הבא נקפוץ עשור קדימה, מ-TLS 1.2 של שנת 2008 אל TLS 1.3 של 2018.

פרק 21: TLS 1.3

מבוא – מוטיבציות לגרסה 1.3

בפרק הקודם סקרנו את TLS 1.2, פרוטוקול שפורסם ב-2008. למרות הוותק הניכר של גרסה 1.2, היא עדיין נחשבת בטוחה לשימוש. עם זאת, יש כמה מוטיבציות לפיתוח גרסה חדשה יותר. לפני שניגש להסבר על גרסה 1.3, נסקור אותן בצורה מסודרת.

תאימות מופרזת לאחור של גרסה 1.2: תאימות לאחור, או Backward Compatibility, היא יכולת חשובה למדי של תוכנות. חישבו לדוגמה על שרת שמשתמש בגרסת תוכנה קצת יותר חדשה משל הלקוח. עדיין היינו רוצים שהלקוח יוכל לתקשר עם השרת. השאלה היא, כמה אחורה בדיוק צריכה להיות אותה תאימות לאחור?

השאלה הזו אינה היפותטית. בגרסה 1.2 של TLS יש תמיכה לאחור באלגוריתמי הצפנה סימטרים כגון DES, משנות השבעים. הזכרנו כבר שזהו אלגוריתם שנחשב כבר שנים רבות לא בטוח לשימוש.

חיסכון בזיכרון: לתאימות לאחור יש גם מחיר. תוכנה של לקוח TLS צריכה להכיל ספריות קוד של כל האלגוריתמים שאולי יהיה בהם שימוש. יותר ספריות קוד -יותר זיכרון שהתוכנה דורשת. כשמדובר במחשב אישי זו בעיה זניחה, אולם כיום נפוצים מכשירי חשמל קטנים שמחוברים לאינטרנט. היצרנים שלהם שואפים להשתמש בכמות קטנה ככל האפשר של זיכרון, כדי להוזיל את העלויות. ויתור חלקי על תאימות לאחור הוא הצעה כלכלית הגיונית עבורם.

חיסכון ב-RTT: ראינו כי לגרסה 1.2 יש 2-RTT. מרגע שהלקוח שולח את הפקטה הראשונה ועד שהוא יכול להתחיל לשלוח מידע "אמיתי", יש פעמיים הלוך ושוב בינו לבין השרת. מה אם נצליח לקצר את זה? בצעו פינג לשרת כרצונכם. אם נוריד RTT אחד, זה הזמן שתוכלו לחסוך בהמתנה לדפדפן שלכם שיעלה את האתר המבוקש. גם חצי שנייה היא זמן משמעותי כשמדובר בחוויה של משתמש, ויצרני הדפדפנים נאבקים להתהדר בתואר "הדפדפן המהיר ביותר".

בנוסף, גם השרת מושפע מהחסכון ב-RTT. ככל שיש פחות רשומות לשלוח ללקוח או לקבל מהלקוח, כך העומס עליו קטן. בשביל דומיינים גדולים, המשרתים מיליוני לקוחות ביום, חיסכון של רשומה אחת בלבד עשוי להיות מתורגם למספר קטן יותר של שרתים.

בפרק זה נסקור את השינויים ב-Cipher Suites, ב-Handshake וב-Session Resumption של גרסה 1.3. לא נסקור את תהליך יצירת המפתחות של גרסה 1.3, המורכב בהרבה מגרסה 1.2. הרקע שניתן בפרק על TLS 1.2 יאפשר למתעניינים לימוד עצמי.

שינויים ב-Cipher Suites

גרסה 1.3 מעדיפה פשטות על פני תמיכה לאחור. בגרסה 1.2 היה אפשר ליצור מאות שילובים של Cipher Suites, חלק ניכר מהם לא בטוחים לשימוש. גרסה 1.3 כוללת כמות מצומצמת של אלגוריתמים מכל סוג שמהם אפשר לבחור:

אלגוריתמי החלפת מפתחות – RSA כבר אינו קיים כאלגוריתם החלפת מפתחות, נשארות רק שתי שיטות מבוססות על DH.

השיטה **הראשונה** היא DHE, כאשר ה-E שנוסף, Ephemeral, פירושו זמני. כזכור ב-DH גם השרת וגם הלקוח בוחרים מספר, סוד פרטי, שמשמש ליצירת הסוד המשותף. כדי להבין מה נותן ה-Ephemeral, נתאר מה קורה אם הסוד אינו Ephemeral, כלומר השרת בוחר את אותו סוד פרטי לכל TLS Session. במקרה כזה, אם גורם כלשהו מצליח לפצח את הסוד הפרטי של השרת, הוא יכול להאזין לכל התשדורות של השרת, קדימה וגם אחורה (אם הן הוקלטו). נכון שגם הלקוח בוחר סוד פרטי, שעשוי להשתנות בין TLS Sessions, אך הדבר אינו עוזר אם הסוד הפרטי של השרת נפרץ. אפשר להשתמש בסוד הפרטי של השרת יחד עם המספר הגלוי שהלקוח מעביר, כדי לחשב את הסוד המשותף.

השיטה השניה המותרת היא ECDHE – קיצור של Elliptic Curve Diffie Hellman Ephemeral. את החלק של DHE כבר הבנו. על ה-Elliptic Curve ניתן לחשוב בתור שיטה גאומטרית לחישוב העלאה בחזקה מודולו N. דמיינו קו מפותל שמשורטט על מישור דו מימדי, אתם בוחרים נקודה על הקו. מכאן אתם מתקדמים באמצעות העברת קו משיק לנקודה שבחרתם, וחיתוך של המשיק עם הקו המפותל. אין צורך להעמיק בפרטים, אך למעוניינים הסרטון הבא מומלץ:

Elliptic Curves -Computerphile

RSA כבר אינו קיים כאלגוריתם החלפת מפתחות בגרסה 1.3 מכיוון שבדרך כלל אינו Ephemeral. שרתים בדרך כלל משתמשים באותו מפתח פרטי עבור סוקטים שונים. הסיבה לכך היא שכמות הזמן שלוקח ליצור מפתחות RSA היא גדולה יחסית. השרת צריך למצוא שני מספרים ראשוניים גדולים, חישוב המבוסס על ניסוי וטעיה ולכן לוקח זמן ניכר. השימוש החוזר במפתח הפרטי הוא סיכון אבטחה. אם מישהו פיצח את המפתח הפרטי של השרת, כל ה-TLS Sessions שלו הופכים להיות גלויים.

כיוון שהורדנו את RSA כאלגוריתם החלפת מפתחות, נותרנו רק עם שיטות שונות של DH. יש לכך משמעות רבה, שתתברר כשנסקור את ה-Handshake.

אלגוריתמי חתימה – נותרו DSA ו-RSA. בוטלו אלגוריתמים ישנים שלא סקרנו.

אלגוריתמי הצפנה סימטרית – נותרו AES128GCM, AES256GCM ואלגוריתם בשם ChaCha20 (השם המוזר הוא כיוון שאלגוריתם זה מחליף אלגוריתם ישן יותר בשם Salsa20...) שלא סקרנו. ירדה רשימה ארוכה של אלגוריתמים ותיקים שנמצאו לא בטוחים מספיק לשימוש.

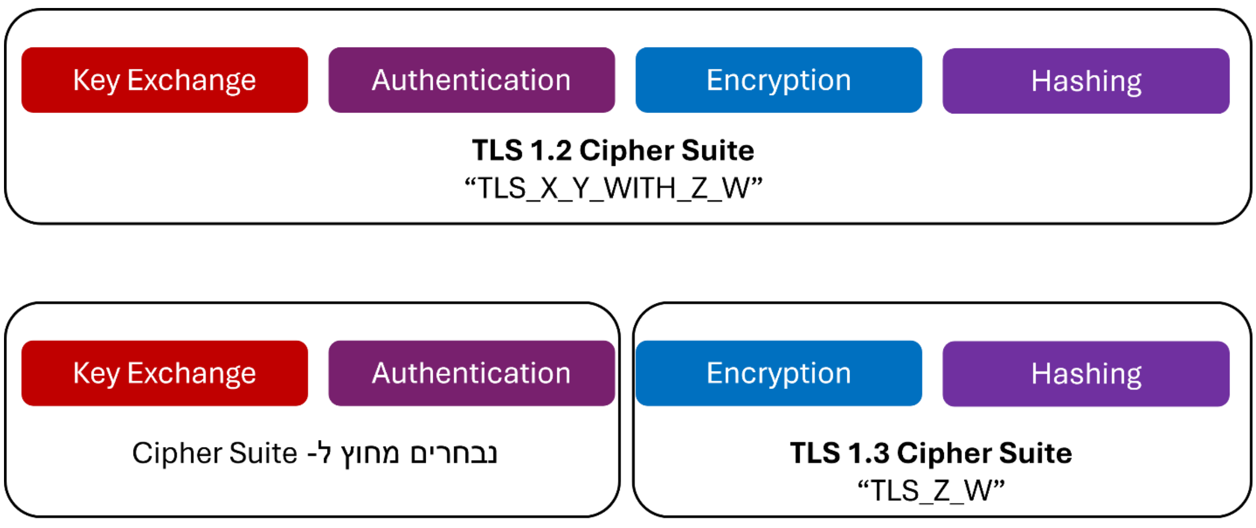
אלגוריתמי Hash – נותרו SHA256, SHA384 ואלגוריתם בשם POLY1305.

כפי שרואים, נותרנו עם כמות של 2–3 אלגוריתמים מכל סוג.

גם ה-Cipher Suites עצמם השתנו:

כזכור Cipher Suites של גרסה 1.2 כוללים את כל הרביעייה של האלגוריתמים. לעומת זאת, בגרסה 1.3 ה-Cipher Suite כוללים רק את ההצפנה הסימטרית וה-Hash. אלגוריתמי החלפת המפתחות והחתימה נבחרים באופן בלתי תלוי ל-Cipher Suite.

לכן Cipher Suite של גרסה 1.3 ייראה כך – TLS_Encryption_Hashing.



מבנה Cipher Suite של גרסה 1.3 לעומת 1.2

יתרה מכך, גרסה 1.3 מגדירה Cipher Suite יחיד בלבד שחייבים לתמוך בו, ועוד שניים מומלצים. כלומר כבר אין צורך אפילו לתמוך בכל עשרות האפשרויות שנותרו.

האפשרות שחייבים לתמוך בה היא:

TLS_AES_128_GCM_SHA256

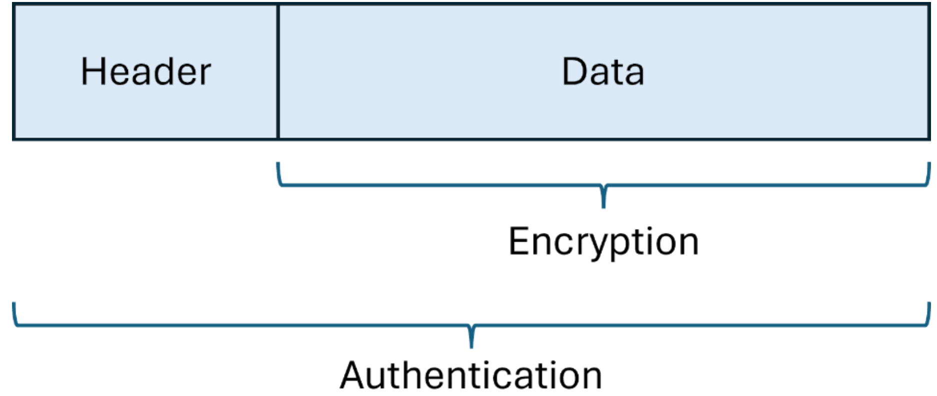
כלומר, יצרנים של ציוד חשמלי פשוט, שרוצים ליצור מכשיר זול שיידע להתחבר לאינטרנט, יכולים לתמוך רק באפשרות זו. זהו שיפור משמעותי יחסית למצב ששרר בגרסה 1.2, בו היה צורך לבזבז נפח אחסון על קוד שמשרת אפשרויות שאינן בשימוש.

שימוש ב-AEAD

עד כה, התייחסנו אל ההצפנה הסימטרית ואל ה-HMAC כאל שני תהליכים נפרדים. לוקחים מידע גלוי, מעבירים אותו את שני התהליכים ומקבלים מידע מוצפן עם HMAC. לא התייחסנו ל**סדר** של ביצוע הדברים. אפשר לבצע הצפנה סימטרית ואז להוסיף HMAC, מה שנקרא Encrypt-then-MAC, ואפשר קודם כל ליצור HMAC ואז להצפין הכל עם הצפנה סימטרית, מה שנקרא MAC-then-Encrypt.

ביצוע התהליכים בזה אחר זה יצר בעיות אבטחה, לדוגמה מתקפת POODLE ומתקפת BEAST, שהזכרנו בפרקים קודמים. בעקבות זאת פותחה משפחה של אלגוריתמים שהם AEAD, קיצור של Authenticated Encryption with Associated Data. באלגוריתמים אלו, ביצוע ההצפנה וחישוב ה-HMAC מתבצע **במקביל**. לכן החלק הקרוי "Authenticated Encryption".

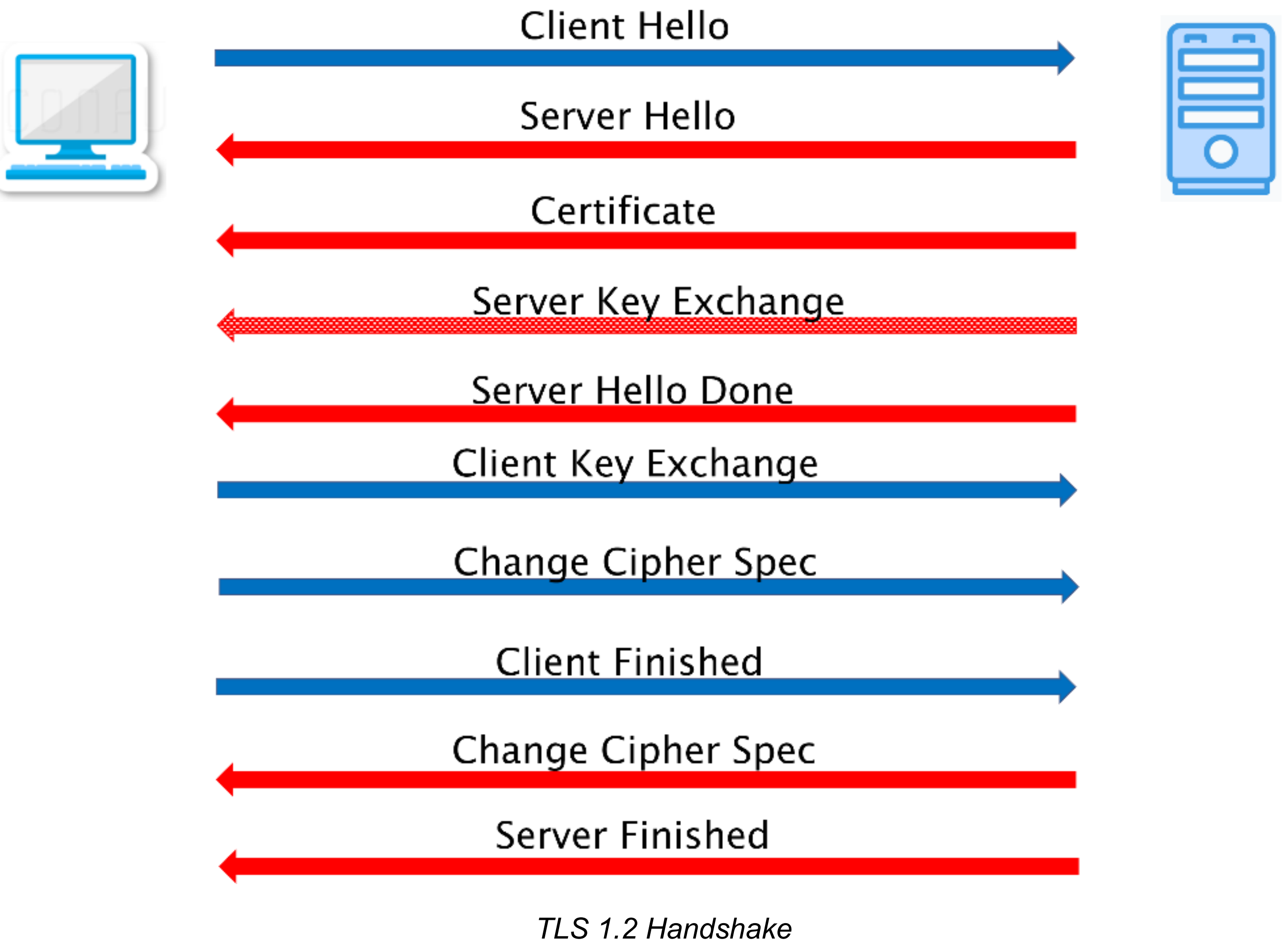
החלק הקרוי "Associated Data" מתייחס לאותנטיקציה של ה-Header. בניגוד למידע, שעובר גם הצפנה וגם משפיע על חישוב ה-HMAC, ה-Header משפיע רק על ה-HMAC. אפשר להקביל את התהליך למעטפה עם מכתב בתוכה (למי שמכירים את הטכנולוגיה הזו...). תוכן המכתב עובר גם הצפנה וגם אותנטיקציה, ואילו הכתובת שרשומה על המעטפה משפיעה על האותנטיקציה אך נשלחת באופן גלוי, כך שכל אחד יכול לקרוא אותה. יש לכך חשיבות במקרים שבהם רוצים להגן מתוקף שעלול לשנות את ה-Header (לדוגמה לשלוח את תוכן המכתב למישהו אחר) אך עדיין לאפשר לרכיבי רשת לקרוא את ה-Header (לדוגמה לאפשר לדוור לקרוא את כתובת היעד).



אלגוריתם AEAD לדוגמה הוא AES-GCM, שסקרנו בפרקים הקודמים. גרסת TLS 1.2 מאפשרת שימוש ב-AEAD אך זו אינה חובה. בגרסה 1.3, כל ההצפנות הסימטריות הן AEAD. לשינוי זה יתרון נוסף – מהירות העיבוד של AEAD גבוהה יותר מאשר מהירות העיבוד של הצפנה ו-HMAC בנפרד.

TLS 1.3 Handshake

כדי להמחיש את החידושים של גרסה 1.3, הנה תזכורת לתהליך בגרסה 1.2:



TLS 1.2 Handshake

לעומת זאת, ה-Handshake בגרסה 1.3 נראה כך:

בולט לעין שיש פחות רשומות שעוברות בין הצדדים. ושיש הפחתה של ה-RTT. עד שהלקוח מקבל את ה- Server Finished עובר RTT יחיד. זהו ה-RTT היחיד שהלקוח צריך להמתין לו, מכיוון שמיד לאחר ה-Client Finished הלקוח כבר יכול לשלוח פקטות של שכבת האפליקציה, כגון בקשות GET.

גילשו לאתר כרצונכם, שמשתמש ב-TLS 1.3. בדוגמה שלפנינו, www.idf.il. מומלץ לקרוא כל דוגמה ולבדוק את הדברים בעצמכם בתוך ההסנפה. ראשית, בצעו פילטר לפי tcp stream ושימרו רק פקטות מסוג TLS 1.2 או 1.3. הסיבה לכך שנשמור גם פקטות 1.2 היא שלעיתים Wireshark יפרש את ה-Client Hello בתור גרסה 1.2, הרי עד שמתקבל ה-Server Hello אין לדעת בדיוק איזו גרסה תיבחר.

נתחיל ללא שימוש בקובץ מפתחות, כדי שנוכל להתרשם מהחלקים ב-Handshake שעוברים גלוי לעומת מה שעובר מוצפן. תקבלו תוצאה כגון זו:

tcp.stream eq 37 and (_ws.col.protocol == TLSv1.3 or _ws.col.protocol == TLSv1.2)

No.	Source	Destinat	Protocol	Info
178	192.1...	45.60...	TLSv1.3	Client Hello (SNI=www.idf.il)
197	45.60...	192.1...	TLSv1.3	Server Hello, Change Cipher Spec, Application Data
204	45.60...	192.1...	TLSv1.3	Application Data, Application Data, Application Data
206	192.1...	45.60...	TLSv1.3	Change Cipher Spec, Application Data

כפי שרואים, מיד לאחר ה-Server Hello יש מעבר למוצפן. ה-Handshake בגרסה 1.3 מספק פחות פרטים לסקרנים שמאזינים לתעבורה בין השרת והלקוח.

Client Hello, Server Hello

אנחנו מכירים כבר את הרשומות של Client Hello ו-Server Hello בגרסה 1.2. נסקור את השינויים המרכזיים בגרסה 1.3.

אחד הפרטים החשובים ביותר שציינו לגבי ה-Cipher Suites, הוא שאין יותר החלפת מפתחות בשיטת RSA. רק DH. למעשה, בגלל השינוי הזה, ה-Cipher Suites של גרסה 1.3 כלל אינם כוללים שיטת החלפת מפתחות. השינוי הזה פותח אפשרות לטריק קטן שמבצע הלקוח – קיצור תהליך החלפת המפתחות. הלקוח עומד לנסות להעביר לשרת את הסוד המשותף עוד לפני שהשרת בחר את ה-Cipher Suite. נפתח את ה-Client Hello ונראה את זה מתרחש.

תחת Cipher Suites, הלקוח עדיין מציע לשרת אפשרויות הקשורות ל-1.2 ול-1.3:

```
˅ Cipher Suites (16 suites)
    Cipher Suite: Reserved (GREASE) (0xfafa)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
```

עם זאת, כיוון שהלקוח תומך ב-1.3 הוא טומן בתוך ה-Extensions הפתעה:

```
˅ Extension: key_share (len=1263) X25519MLKEM768, x25519
    Type: key_share (51)
    Length: 1263
  ˅ Key Share extension
      Client Key Share Length: 1261
    › Key Share Entry: Group: Reserved (GREASE), Key Exchange length: 1
    › Key Share Entry: Group: X25519MLKEM768, Key Exchange length: 1216
    › Key Share Entry: Group: x25519, Key Exchange length: 32
```

הלקוח אומר לשרת "שמע, אם במקרה תבחר בגרסה 1.3, אז שנינו יודעים שנחליף מפתחות בעזרת DH. בתוך התקן של TLS מוגדרים כמה Elliptic Curves של DH. אני *מנחש* שתעבוד עם אחד מה-Elliptic Curves הבאים. לכן, אני מראש מוסיף פה, שאם הבחירה שלך תהיה אחד מאלה, קבל כבר מעכשיו את החלק שלי בבחירה". הניחוש הזה, אם הוא מצליח, חוסך לגמרי את ה-Client Key Exchange שקיים בגרסה 1.2. אם הניחוש מצליח, השרת יכול לומר ללקוח "אוקיי, הנה החלק שלי בסוד המשותף, והנה אני כבר מתחיל להצפין".

היכנסו לתשובת השרת, ב-Server Hello:

```
˅ Extension: key_share (len=1124) X25519MLKEM768
    Type: key_share (51)
    Length: 1124
  ˅ Key Share extension
    ˅ Key Share Entry: Group: X25519MLKEM768, Key Exchange length: 1120
        Group: X25519MLKEM768 (4588)
        Key Exchange Length: 1120
        Key Exchange […]: a247cb67a0cab35fdd76310bbadacc62786d0fe7be77a6a4
```

בדוגמה זו השרת עונה ללקוח "אכן, בחרתי ב-Elliptic Curve שנקרא בתקן X25519MLKEM768. לכן החלק שלך של הסוד המשותף התקבל על-ידי בהצלחה. בבקשה, הנה החלק שלי בסוד המשותף".

שימו לב לכך שבניגוד לגרסה 1.2, השרת *אינו מצרף חתימה* על מפתח ה-DH שלו. החתימה שימשה כדי שהלקוח יוכל לוודא שלשרת יש את המפתח הפרטי המתאים למפתח הציבורי שבסרטיפיקט. האם בגרסה 1.3 זה מיותר? כמובן שלא. השרת מספק חתימה, אבל במקום אחר, שנגיע אליו בקרוב.

אך מה קורה אם הלקוח שגה בניחוש והשרת בחר Elliptic Curve שונה?

במקרה זה, השרת יודיע ללקוח שהניחוש שלו היה שגוי. הדרך שבה השרת מודיע זאת היא באמצעות שדה ה-Random. השרת ייקח את המחרוזת "HelloRetryRequest", יבצע לה SHA256, וישלח את התוצאה בתור Random.

תרגיל מודרך: מציאת HelloRetryRequest

באמצעות Powershell, מיצאו את ה-SHA256 של HelloRetryRequest והכניסו ל-Wireshark את הפילטר הבא:

tls.handshake.random == הערך שקבלתם

אם היו ניחושים לא מוצלחים, תוכלו לצפות בהם:

tls.handshake.random == CF:21:AD:74:E5:9A:61:11:BE:1D:8C:02:1E:65:B8:91:C2:A2:11:16:7A:BB:8C:5E:07:9E:09:E2:C8:A8:33:9C

No.	Source	Destinat	Protocol	Info
278	13.10…	192.1…	TLSv1.3	Hello Retry Request, Change Cipher Spec
588	13.10…	192.1…	TLSv1.3	Hello Retry Request, Change Cipher Spec
595	13.10…	192.1…	TLSv1.3	Hello Retry Request, Change Cipher Spec
1880	20.50…	192.1…	TLSv1.3	Hello Retry Request, Change Cipher Spec

מה קורה בעקבות ניחוש לא מוצלח? הרשומות הבאות נראות כך:

No.	Source	Destinat	Protocol	Info
275	192.1…	13.10…	TLSv1.3	Client Hello (SNI=amp.azure.net)
278	13.10…	192.1…	TLSv1.3	Hello Retry Request, Change Cipher Spec
279	192.1…	13.10…	TLSv1.3	Change Cipher Spec, Client Hello (SNI=amp.azure.net)
280	13.10…	192.1…	TLSv1.3	Server Hello, Encrypted Extensions
285	13.10…	192.1…	TLSv1.3	Certificate, Certificate Verify, Finished

כישלון הניחוש גרם לכך שהשרת הודיע ללקוח מה ה-Elliptic Curve שצריך לעבוד איתו, הלקוח שלח Client Hello חדש, הפעם עם ה"ניחוש" הנכון, ומשם המשך הרשומות היה כפי שכבר ראינו.

Extensions

כפי שראינו חלק מהמידע החשוב ב-Handshake עובר בתוך Extensions. לדוגמה, ה-Key Share. השם "Extension" רומז על כך שזה אינו ערך שחובה להכניס לרשומת ה-TLS. כך היה ב-TLS 1.2. לעומת זאת, בגרסה 1.3 חלק מה-Extensions הם חובה.

הפורמט של Extension מחולק לשלושה שדות: סוג – שני בתים; אורך – שני בתים; ומידע – לפי שדה האורך.

נמחיש באמצעות ה-Extension של SNI, עם סימון הבתים של השדות השונים:

```
⌄ Extension: server_name (len=15) name=www.idf.il
    Type: server_name (0)
    Length: 15
  ⌄ Server Name Indication extension
      Server Name list length: 13
      Server Name Type: host_name (0)
      Server Name length: 10
      Server Name: www.idf.il

00b0  00 00 2b 00 07 06 1a 1a  03 04 03 03 00 00 00 0f   ··+·····  ········
00c0  00 0d 00 00 0a 77 77 77  2e 69 64 66 2e 69 6c 00   ·····www  .idf.il·
```

השדה הראשון, סוג ה-Extension, שווה ל-0x0000. זה הערך של SNI. שדה האורך הוא 0x000F, כלומר 15 בתים. המידע עצמו מחולק בין כמה שדות, המוגדרים עבור SNI.

Encrypted Extensions

כיוון שהגענו לרשומות מוצפנות, נטען את מפתחות ההצפנה ונמשיך בפענוח. רשומות שלפני כן היו מוצפנות, מופיעות כרגע בשמן.

tcp.stream eq 37 and (_ws.col.protocol == TLSv1.3 or _ws.col.protocol == TLSv1.2)

No.	Source	Destinat	Protocol	Info
178	192.1…	45.60…	TLSv1.3	Client Hello (SNI=www.idf.il)
197	45.60…	192.1…	TLSv1.3	Server Hello, Change Cipher Spec, Encrypted Extensions
204	45.60…	192.1…	TLSv1.3	Certificate, Certificate Verify, Finished
206	192.1…	45.60…	TLSv1.3	Change Cipher Spec, Finished

הרשומה הראשונה מבין הרשומות המוצפנות היא Encrypted Extensions.

נדון ברשומת ה-ALPN החשובה, קיצור של Application Layer Protocol Negotiation.

```
˅ Handshake Protocol: Encrypted Extensions
    Handshake Type: Encrypted Extensions (8)
    Length: 15
    Extensions Length: 13
  › Extension: server_name (len=0)
  ˅ Extension: application_layer_protocol_negotiation (len=5)
      Type: application_layer_protocol_negotiation (16)
      Length: 5
      ALPN Extension Length: 3
    ˅ ALPN Protocol
        ALPN string length: 2
        ALPN Next Protocol: h2
```

כפי שאפשר לראות, השרת מכריז על שימוש ב-h2 (שם נרדף ל-HTTP/2). זוהי למעשה תשובה, שתאימה להצעה של הלקוח ב-Client Hello:

```
˅ Extension: application_layer_protocol_negotiation (len=14)
    Type: application_layer_protocol_negotiation (16)
    Length: 14
    ALPN Extension Length: 12
  ˅ ALPN Protocol
      ALPN string length: 2
      ALPN Next Protocol: h2
      ALPN string length: 8
      ALPN Next Protocol: http/1.1
```

הלקוח אמר לשרת, "אני תומך ב-h2. אם אתה לא, אפשר לעבוד עם HTTP/1.1". השרת ענה לו "h2 טוב".

מה תורם התיאום הזה? כאשר לקוח פונה לשרת הוא לא יכול להיות בטוח מה גרסת הפרוטוקול שהשרת תומך בה. זה לא חייב להיות HTTP כמובן, ויכול להיות כל פרוטוקול של שכבת האפליקציה. יש אפשרות שהלקוח יתחיל לעבוד מול השרת עם פרוטוקול בגרסה ישנה והם יבצעו תיאום, אך זה מבזבז RTT על תיאומים. לדוגמה, לקוח ישלח בקשת GET עם HTTP/1.1, יקבל תשובה שהשרת תומך ב-h2 וישלח את אותה בקשה שוב עם h2. ה-TLS Handshake מנוצל לטובת תיאום מקדים.

Certificate

בשלב זה, השרת שולח רשומת סרטיפיקט ולאחריה רשומת Certificate Verify.

את הסרטיפיקט אנחנו מכירים, הוא לא משתנה עם המעבר לגרסה 1.3. עם זאת, ההבדל הוא שב-1.3 הסרטיפיקט עובר מוצפן.

כאשר סקרנו את גרסה 1.2, הזכרנו את נושא הפרטיות והצנזורה. שדה ה-SNI ב-Client Hello מספק למי שיושב בין הלקוח והשרת מידע לאן בוצעה הגלישה, דבר שמאפשר לעקוב אחרי הרגלי הגלישה או לצנזר את המשתמש. הזכרנו שיש מחשבות לפתור את הבעיה הזו, לדוגמה על-ידי Encrypted Client Hello – ECH. רישומי ה-DNS של דומיין יכללו את המפתח הציבורי שלו, כך שניתן יהיה להצפין את ה-Client Hello שנשלח אליו. בין השדות המוצפנים יהיה גם שדה ה-SNI.

עם זאת, הצפנת ה-Client Hello לבדה לא תפתור את הבעיה, כיוון שבגרסה 1.2 הסרטיפיקט של השרת נשלח ללקוח בצורה גלויה וממנו אפשר להסיק את יעד הגלישה.

גרסה 1.3 היא התקדמות בכיוון לעבר פרטיות ומניעת צנזורה, שכן השרת שולח את הסרטיפיקט של עצמו מוצפן, וה-SNI בתגובת השרת עובר בתוך ה-Encrypted Extensions. כלומר, בגרסה 1.3 המקום היחיד שבו צופה מהצד יכול לראות מול איזה שרת בוצעה התקשורת הוא רק שדה ה-SNI שב-Client Hello. לכן, שילוב של TLS 1.3 עם ECH עשוי להיות פתרון לבעיה.

בפרק על סרטיפיקטים סקרנו את שלוש השיטות לביצוע Certificate Validation, כלומר לבדוק שהסרטיפיקט עדיין בתוקף. השיטות הן כזכור CRL, OCSP ו-OCSP Stapling. בגרסה 1.3 קיימת אפשרות שהלקוח יבקש מהשרת לעבוד בשיטת OCSP Stapling, ואם השרת תומך בכך הוא יענה בהתאם.

בידקו האם ב-Client Hello יש Status Request:

```
˅ Extension: status_request (len=5)
    Type: status_request (5)
    Length: 5
    Certificate Status Type: OCSP (1)
    Responder ID list Length: 0
    Request Extensions Length: 0
```

כפי שרואים במקרה זה, הלקוח ביקש OCSP, למעשה הכוונה ל-OCSP Stapling.

במקרה שזו היתה בקשת הלקוח, פיתחו את הסרטיפיקט, וגם שם תמצאו Status Request. כפי שרואים, השרת צירף אישור חתום על-ידי שרת OCSP לכך שהסרטיפיקט שלו בתוקף, הכולל את המספר הסידורי של הסרטיפיקט ואת התאריכים שהאישור בתוקף:

```
˅ Extension: status_request (len=604)
    Type: status_request (5)
    Length: 604
    Certificate Status Type: OCSP (1)
    OCSP Response Length: 600
  ˅ OCSP Response
      responseStatus: successful (0)
    ˅ responseBytes
        ResponseType Id: 1.3.6.1.5.5.7.48.1.1 (id-pkix-ocsp-basic)
      ˅ BasicOCSPResponse
        ˅ tbsResponseData
          › responderID: byKey (2)
            producedAt: Feb 19, 2026 12:33:59.000000000 Jerusalem Standard Time
          ˅ responses: 1 item
            ˅ SingleResponse
              › certID
              › certStatus: good (0)
                thisUpdate: Feb 19, 2026 12:33:59.000000000 Jerusalem Standard Time
                nextUpdate: Feb 26, 2026 12:33:58.000000000 Jerusalem Standard Time
```

Certificate Verify

כאשר סקרנו את ה-Server Hello ראינו שהשרת שולח את ה-DH הציבורי שלו, אך אינו מצרף אליו חתימה.

השם "רשומת ה-Certificate Verify" הוא מטעה מעט. למעשה, זוהי רשומה שחותמת על *כל* הרשומות ששלח השרת עד כה. לא רק הסרטיפיקט אלא גם ה-Server Hello. ה-Key Share הוא חלק מה-Server Hello, כך שבעצם החתימה כוללת גם אותו.

בגרסה 1.2 אין רשומה כזו כלל. ניזכר מה היה לנו ב-1.2.

הדרך שבה הלקוח מוודא שהשרת הוא הבעלים של המפתח הפרטי המתאים לציבורי היא או באמצעות חתימה של השרת על החלק שלו ביצירת הסוד המשותף (אם החלפת המפתחות היא DH) או שהלקוח מצפין את החלק שלו ביצירת הסוד המשותף באמצעות המפתח הציבורי של השרת (אם החלפת המפתחות היא RSA).

שיטת החלפת המפתחות באמצעות RSA אינה רלבנטית לגרסה 1.3, לכן נותרנו רק עם חתימה של השרת על החלק שלו בסוד המשותף. רשומת ה-Certificate Verify מרחיבה את החתימה. ב-1.2 החתימה היתה רק על ה-Pubkey, ואילו ב-1.3 החתימה היא על כל הרשומות עד כה.

הלקוח ישתמש בחתימה הזו כדי לוודא שהשרת הוא אכן הבעלים של המפתח הציבורי.

פיתחו את ה-Client Hello, ותחת Extensions תמצאו את ה-Signature Algorithms. אלו אלגוריתמים של חתימה ו-Hash שהלקוח מציע במיוחד בשביל הרשומה של Certificate Verify. אפשר לראות בה סכמות שונות של RSA, כגון pkcs1 שהזכרנו כשלמדנו איך השרת מוכיח ללקוח שהוא בעל המפתח הפרטי המתאים לסרטיפיקט, בפרק על TLS 1.2.

```
˅ Extension: signature_algorithms (len=18)
    Type: signature_algorithms (13)
    Length: 18
    Signature Hash Algorithms Length: 16
  ˅ Signature Hash Algorithms (8 algorithms)
    › Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
    › Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
    › Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
    › Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
    › Signature Algorithm: rsa_pss_rsae_sha384 (0x0805)
    › Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
    › Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
    › Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
```

הלקוח מודיע לשרת באילו אלגוריתמי חתימה הוא תומך.

נמצא את המענה של השרת בתוך ה-Certificate Verify:

```
˅ Handshake Protocol: Certificate Verify
    Handshake Type: Certificate Verify (15)
    Length: 260
  › Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
    Signature length: 256
    Signature […]: 86f4e7ff31a5a687a6c797a1e2808176b81
```

לסיכום, הלקוח הציע אלגוריתמי חתימה משולבי Hash כחלק מה-Client Hello, בתוך Extension. השרת בחר את אחד האלגוריתמים שהלקוח הציע וחתם באמצעותו על כל הרשומות שהשרת שלח עד כה.

סיכום תהליך ה-Handshake

לאחר ה-Certificate Verify מועברות רשומות ה-Finished המוכרות לנו מגרסה 1.2. לפני שהלקוח שולח את ה-Finished שלו הוא מודיע על מעבר למוצפן. בכך מסתיים תהליך ה-Handshake. נסכם את הדברים המרכזיים:

- ה-Cipher Suite של 1.3 כולל רק אלגוריתם הצפנה סימטרית ו-Hash. כמו כן, הופסק השימוש ברבים מהאלגוריתמים שגרסה 1.2 תמכה בהם, לטובת תאימות לאחור. אחת המשמעויות היא שאלגוריתם החלפת המפתחות הוא תמיד DH.
- חסכנו RTT. החסכון מושג באמצעות תוספת ל-Client Hello, שבה הלקוח מנחש מה עקום ה-DH שהשרת ייבחר, ושולח לו מראש את מפתח ה-DH הציבורי שלו. אם הניחוש נכשל, השרת יודיע על כך באמצעות HelloRetryRequest. למרות תשלום של RTT נוסף, התהליך יושלם.
- בעקבות הצפנה של הסרטיפיקט, המקום היחידי שבו צופה מהצד יכול לדעת לאן הלקוח פנה הוא שדה ה-SNI ב-Client Hello. יש עבודה על תקן שמטרתו להשלים גם את ההצפנה של שדה זה.
- באמצעות רשומת ה-Certificate Verify ששולח השרת, הלקוח יכול לוודא שני דברים. א' – לשרת יש המפתח הפרטי המתאים למפתח הציבורי שבסרטיפיקט; ו-ב' – שאף גורם לא שינה את הרשומות הקודמות ששלח השרת.

Session Resumption

לגרסה 1.3 יש אפשרות למה שנקרא 0-RTT. כלומר, הלקוח אינו מחכה כלל לתגובת השרת ומתחיל מיד לשלוח מידע של שכבת האפליקציה.

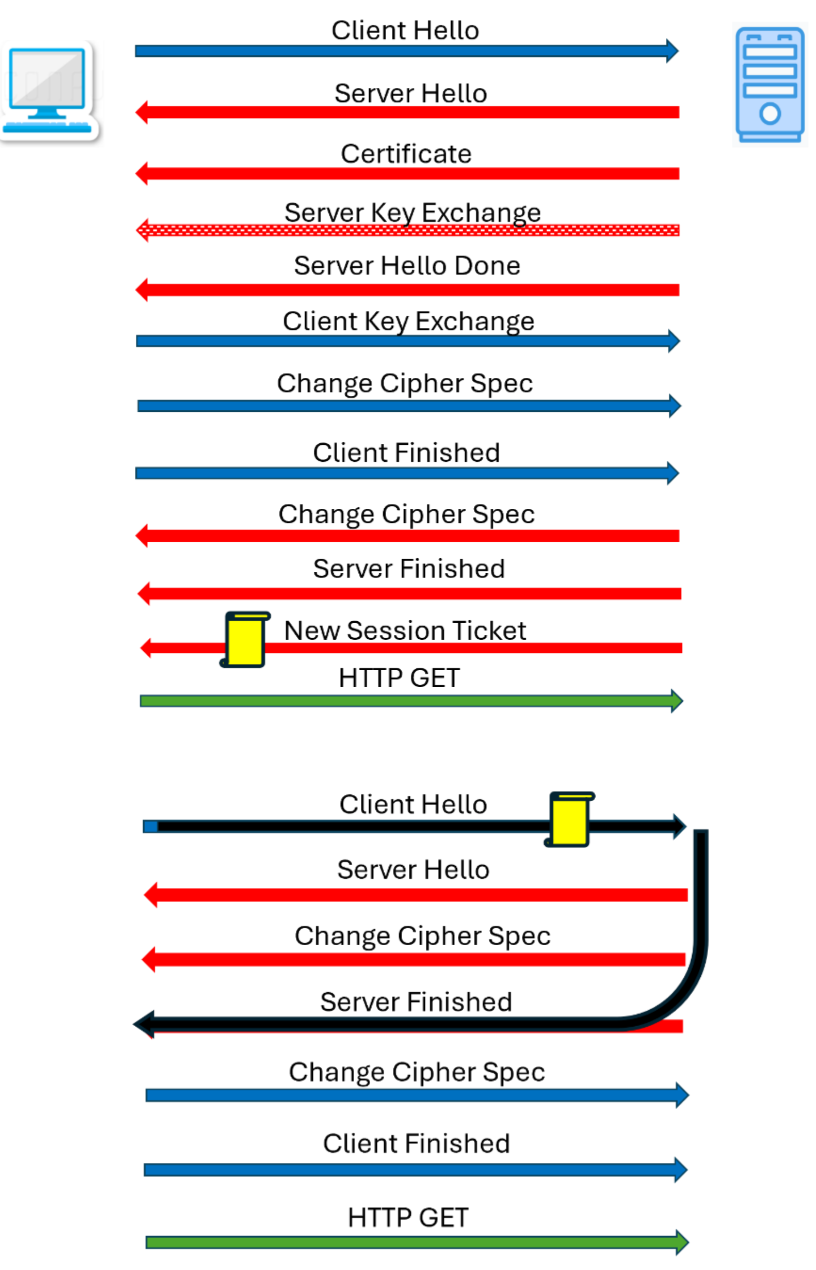
כדי להבין כיצד זה אפשרי, ניזכר ברעיון של Session Resumption שלמדנו בגרסה 1.2. לאחר ביצוע Handshake רגיל, השרת שלח ללקוח רשומה שנקראת New Session Tickets.

ה-Session Ticket כולל:

- פרק זמן שבו הוא בתוקף
- גרסת ה-TLS שתואמה בין הצדדים
- ה-Cipher Suite שתואם
- ה-Master Secret שתואם
- ה-Randomים של השרת והלקוח

מהמידע הזה אפשר לחשב מחדש את כל המפתחות של השרת והלקוח, והוא נשלח מוצפן באמצעות המפתח הציבורי של השרת, כך שרק השרת יכול לפענח אותו.

אם הלקוח רוצה להתחבר לשרת שוב, הוא יכול לשלוח את ה-Session Ticket בתוך ה-Client Hello ולקבל חיבור מקוצר:



שימוש ב-Session Ticket בגרסה 1.2

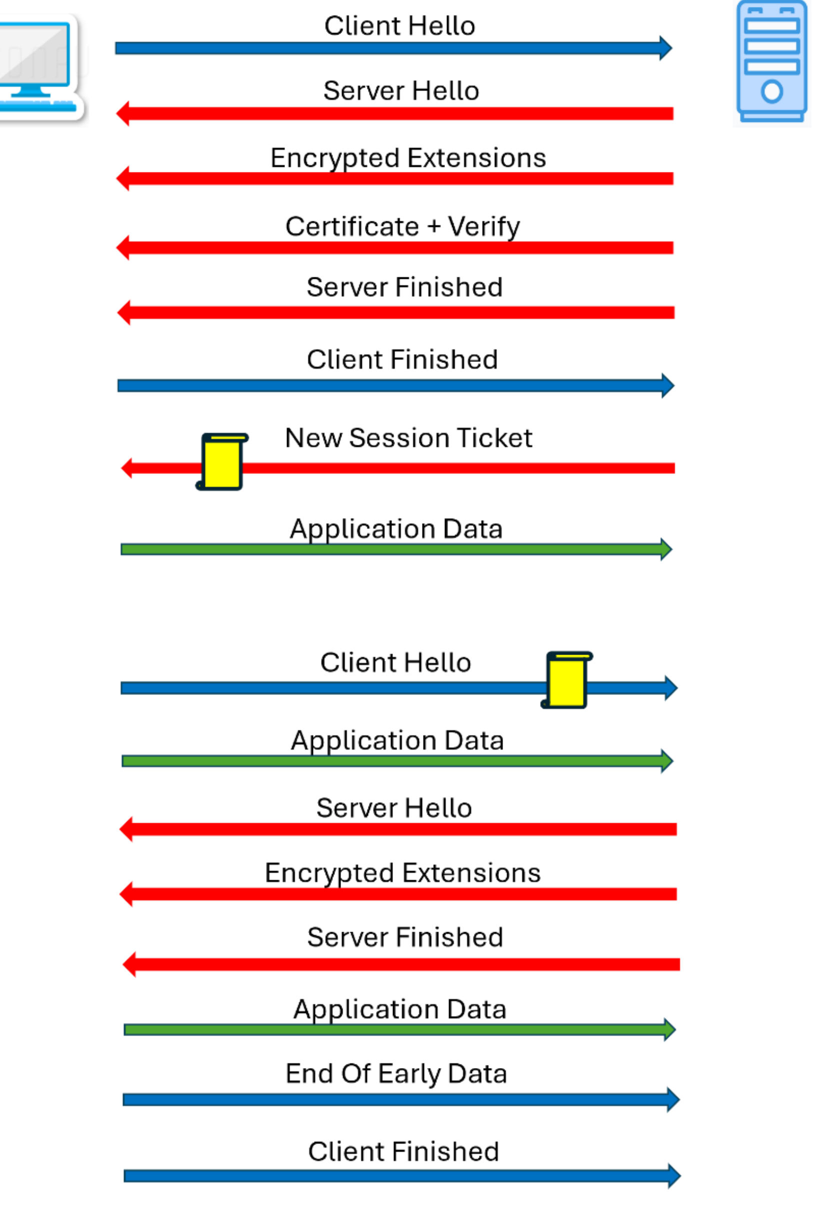
גם בגרסה 1.3 יש שימוש ב-Session Ticket. אך כיוון שבגרסה 1.3 יש ממילא 1-RTT, הירידה היא ל-0-RTT. נסקור את תהליך ה-0-RTT לפרטים.

בהקמת הקשר הראשונית, הלקוח יציין ב-Client Hello שהוא תומך ב-PSK Key Exchange Mode. המונח PSK הוא ראשי תיבות של Pre-Shared Key, כלומר מפתח שתואם מראש בין הצדדים.

בעקבות זאת, השרת עשוי לשלוח ללקוח רשומה של New Session Ticket וכן להוסיף Extension בשם Early Data. תוספת זו מציינת שהשרת תומך ב-0-RTT. התוספת תכלול את כמות הבתים המקסימלית שהשרת מוכן לקבל מהלקוח לפני שהשרת ידרוש מהלקוח החלפת מפתחות רגילה. לדוגמה, השרת יכול להחליט שהלקוח יכול להשתמש במפתח שנקבע ביניהם מראש לשליחת מידע עם 0-RTT, אך לא יותר מאשר 64KB. כעת השרת והלקוח סיימו את ההתקשרות ביניהם.

בהתקשרות הבאה של הלקוח, הוא שולח Client Hello, ובלי לחכות לתגובת השרת – מידע של שכבת האפליקציה, מה שקראנו Early Data. השרת יתחיל את תהליך ה-Handshake כרגיל, תוך כדי שהוא מקבל את ה- Early Data של הלקוח.

כאשר הלקוח רוצה לעבור למפתחות ההצפנה החדשים, או שהוא נאלץ לעשות זאת בלית ברירה מכיוון שהגיע למקסימום הבתים שהשרת הרשה לו לשלוח על גבי Early Data, הלקוח שולח רשומת End of Early Data. לאחריה יבוא Client Finished ואז שני הצדדים יעברו למפתחות החדשים שתואמו ביניהם.



TLS 1.3 0-RTT

21.1 תרגיל ניתוח הסנפה

הורידו את הקובץ הבא:

https://data.cyber.org.il/networks/TLS13files.rar

תחת הקובץ נמצאים קובץ הסנפה וקובץ מפתחות. מיצאו את הפקטות המשוייכות לתקשורת עם jct.ac.il.

Client Hello

א. מהי הגרסה של ה-TLS לפי ה-Header של הרשומה? מהי הגרסה של ה-TLS לפי ה-Client Hello?

ב. כמה Cipher Suites הציע הלקוח?

ג. אילו Cipher Suites הן TLS 1.3?

ד. אילו DH Elliptic Curves מציע הלקוח?

ה. בשלב זה, האם הלקוח יודע אילו DH Elliptic Curves נתמכים על-ידי השרת?

Server Hello

ו. מהי הגרסה של ה-TLS לפי ה-Header של הרשומה? מהי הגרסה של ה-TLS לפי ה-Server Hello?

ז. איזה Cipher Suite בחר השרת?

ח. האם הלקוח בחר נכון? מה היה קורה אם לא?

סרטיפיקט

ט. מי הבעלים של הסרטיפיקט הראשון?

י. מי חתם על הסרטיפיקט לבעלים של הסרטיפיקט הראשון?

יא. מה התפקיד של שדה ה-Status Request? איזה פרוטוקול בשימוש?

יב. מהו ה-Serial Number של הסרטיפיקט?

יג. מהו ה-Serial Number שה-Status Request בתגובת השרת מתייחס אליו?

יד. מי הבעלים של הסרטיפיקט השני?

טו. מי חתם על הסרטיפיקט לבעלים של הסרטיפיקט השני?

סיכום

בפרק זה התמקדנו בשני שינויים מרכזיים של גרסה 1.3. השינויים ב-Cipher Suites יוצרים פשטות רבה יותר ומאפשרים גם למכשירים זולים לתמוך ב-TLS בלי להכביד מדי בעלויות הזיכרון.

לאחר מכן סקרנו את ה-Handshake. הוויתור על RSA כאלגוריתם החלפת מפתחות מאפשר ללקוח לנחש מראש באיזו שיטה של DH השרת יבחר, וכך לחסוך RTT.

בשלב זה אנחנו יכולים לעלות לשכבת האפליקציה ולראות מה קרה לשני פרוטוקולים נפוצים, HTTP ו-DNS, בעקבות המעבר של דפדפנים ל-TLS. בעשור האחרון התרחשה מהפכה, ונגלה שלא מעט מהדברים שלמדנו על פרוטוקולים אלו השתנו.

פרק 22: DNS ו-HTTP מעל TLS

הקדמה

עד כה תיארנו את ההתפתחות הטכנולוגית של סוקטים. אי שם בשנות התשעים, סיום של לחיצת יד משולשת היה מספיק בשביל להתחיל להעביר פקטות של שכבת האפליקציה. היום, אנחנו אחרי שש גרסאות של SSL ו-TLS. מה לגבי שכבת האפליקציה? האם השינויים פסחו על הפרוטוקולים הוותיקים הללו?

ובכן, עקרונית אין קשר בין הדברים. טכנית ניתן להעביר מעל סוקט של TLS 1.3 את גרסה 1.1 של פרוטוקול HTTP. גם פרוטוקול DNS עובד היטב, כפי שאפשר לגלות בקלות מביצוע NSLOOKUP.

עם זאת, סביר שבהסנפה של גלישת אינטרנט לא תזהו לא את גרסה 1.1 של HTTP ולא פקטות DNS. בפרק זה אנו שבים לבקר את גיבורי הילדות שלנו משנות התשעים ושואלים "מה קרה להם מאז?"

HTTP/2

נסקור ארבעה שינויים שחלו בגרסה 2 של HTTP (הקרויה HTTP/2 או H2):

- Stream ID
- HTTP PING
- Packed Header
- Server Push

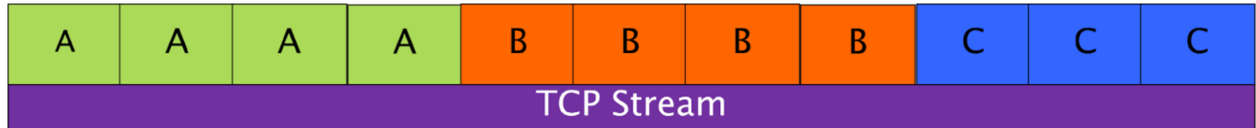
Stream ID

לפני שניכנס לפרטים על אודות Stream ID ב-HTTP/2, ניזכר באופן הפעולה של גרסה 1.1.

גרסה 1.1 של HTTP היא גרסה שמעבירה משאבים בצורה טורית. כלומר, משאב ב' מועבר רק לאחר שהסתיימה ההעברה של משאב א'.

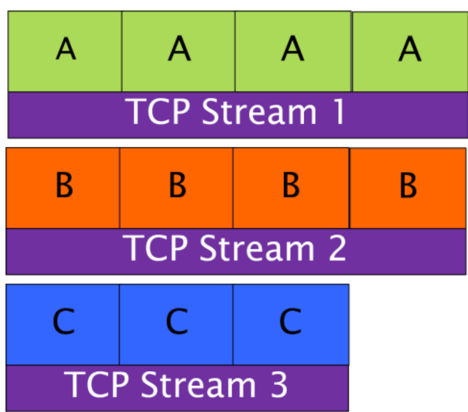
כל משאב מחולק לפקטות לפי הצורך, והן נשלחות מעל פרוטוקול TCP. לכל פקטת TCP יש SEQ מתאים, אשר מתקדם מפקטה לפקטה.

לדוגמה, אם לשרת שלושה משאבים, שיסומנו A, B, C, אז זרם הפקטות מהשרת ללקוח עשוי להראות כך:



זוהי צורת העברה לא אופטימלית, מכיוון שאם חל עיכוב בשליחת אחד המשאבים, אז כל יתר המשאבים הבאים יחכו. לדוגמה, אם משאב B הוא קובץ וידאו כבד ואילו משאב C הוא קובץ טקסט קטן יחסית, לא ניתן יהיה להציג למשתמש את הטקסט עד שההורדה של הוידאו תסתיים. התוצאה עלולה להיות חווית גלישה שמצריכה סבלנות וגולשים שנוטשים את האתר.

דפדפנים פתרו את הבעיה באופן חלקי באמצעות פיצול התעבורה למספר סוקטים של TCP. כך, זרם המידע שבדוגמה הקודמת עשוי להראות כך:



חלוקת זרם המידע למספר זרמים שאינם תלויים זה בזה

הידד. האם הבעיה נפתרה?

התשובה היא "כן, אבל". פרוטוקול TCP הוא אופטימלי להעברה של קבצים ארוכים. הסיבה לכך היא שיש לו התחלה איטית, וקצב התעבורה עולה. נושא זה לא כוסה בחלקו הראשון של ספר רשתות, לכן נסקור אותו בקצרה. דמיינו שביד שלכם יש כדור פינג פונג. אתם מדביקים עליו מילה ומוסרים לחברכם. החבר מחזיר לכם את הכדור הריק, לשימוש חוזר. רק לאחר שהכדור שוב בידיכם אתם יכולים להדביק עליו מילה ולמסור לחבר. כלומר כל מילה שאתם רוצים לשלוח מצריכה ממכם המתנה של RTT אחד.

מצבכם ישתפר אם תהיה לכם כמות גדולה יותר של כדורי פינג פונג. הכדור הראשון שמסרתם עדיין באויר, ואתם כבר שולחים את הכדורים הנותרים. אם בידיכם חמישה כדורים לדוגמה, אז אתם יכולים לשלוח בבת אחת חמישה ולקבל בבת אחת חמישה כדורים ריקים לשימוש חוזר. כעת ב-RTT יחיד אפשר לשלוח חמש מילים.
ובכן למה שלא נגדיל את כמות הכדורים? אולי אתם יכולים להסתדר עם עשרה כדורים שנמצאים באויר בו זמנית. אולי יותר. אך מתישהו אתם או חברכם לא תצליחו לעמוד בקצב, וכדורים יפלו ויילכו לאיבוד. ובכן, האם יש כמות אופטימלית של כדורים שתוכלו לעמוד בה? ואם כן – כיצד מוצאים אותה?

כדי למצוא את מספר הכדורים האופטימלי, תוכלו לסכם עם חברכם כך: שימו לידכם ארגז של כדורים. בתחילת התקשורת ביניכם לבין החבר, תוציאו מהארגז כדור יחיד. אם קבלתם אותו חזרה – קחו כדור נוסף מהארגז. כלומר בפעם השנייה תמסרו לחבר שני כדורים. בפעם השלישית תמסרו לחבר ארבעה כדורים (זיכרו, אתם לא מוסיפים רק כדור אחד בכל פעם אלא מוסיפים כדור על כל כדור שהגיע אליכם חזרה). כמות הכדורים בידיכם תעלה לפי חזקות של 2.

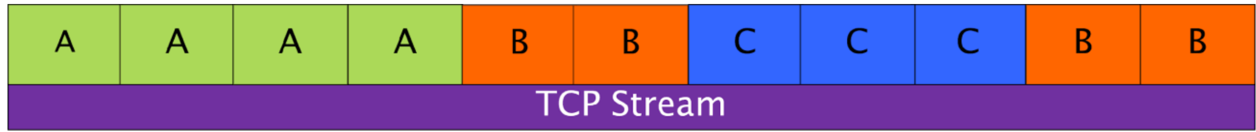
לעומת זאת, אם כדור שמסרתם לא חזר, אתם מאבדים מחצית מכמות הכדורים שבידכם ועליכם להחזיר אותם לארגז. אך אל דאגה – אם תמסרו אותם והם יוחזרו אליכם בהצלחה, כבר במסירה הבאה תחזירו לידיכם את כל הכדורים האבודים.

אלגוריתם זה דומה לאלגוריתם שמשתמשים בו בקישורי TCP. לא לחינם האלגוריתם זכה לשם "TCP Slow Start". משחק הפינג פונג לימד אותנו שבתחילת הקמת הקשר, קצב התקשורת בין הצדדים הוא איטי, בסך הכל פקטה אחת לכל RTT, אולם קצב התקשורת עולה מעריכית עד שהוא מתייצב על המקסימום האפשרי. אלגוריתם TCP Slow Start אינו האלגוריתם היחיד למציאת קצב התקשורת האופטימלי ב-TCP, אך מה שמשותף לכל האלגוריתמים הוא שמציאת הקצב דורשת זמן. אין קישור שברגע שהוא מוקם הוא כבר בקצב אופטימלי.

לאחר שהבנו את הרעיון של אלגוריתם TCP Slow Start, אפשר לבחון שנית את הפתרון של חלוקת זרם ה-TCP היחיד למספר זרמי מידע. הבעיה שניסינו לפתור בחלוקה לזרמי מידע, היא משאב קטן שהשליחה שלו מתעכבת כיוון שנמצא אחרי משאב כבד. לפתוח למשאב הקטן זרם TCP משלו נשמע בתחילה פתרון טוב, אולם זרם המידע החדש יתחיל בקצב איטי. כל מי שעמדו פעם בתור מכירים את התופעה שיש שני תורים, האחד ארוך אבל מתקדם מהר והשני קצר אבל מתקדם לאט. מי שבוחרים את התור הקצר עלולים לגלות שבעצם הפסידו.

גרסה 2 של HTTP חוזרת לזרם TCP יחיד, כדי לא להפסיד מה-TCP Slow Start. יחד עם זאת, שדה חדש בפרוטוקול – Stream ID – מאפשר להקצות לכל משאב זרם מידע מזוהה משלו, ברמת שכבת האפליקציה. הדבר מאפשר לשרת וללקוח לזהות מתוך כלל תעבורת ה-TCP פקטות ספציפיות ששייכות למשאב כזה או למשאב אחר.

איך דבר זה מסייע לפתרון הבעיה של משאב קטן, כמו טקסט, שהשליחה שלו מתעכבת בגלל משאב כבד, כמו וידאו? ובכן, ה-Stream ID אינו מסייע לבעיה זו. אבל, במקרים רבים שליחה של פקטות מתעכבת מכיוון שהמשאב אינו מוכן לשליחה בשרת. המשאב עשוי להיות תוצאה של חישוב, פנייה למאגר מידע וכו'. נניח שמשאב B הוא משאב כזה. כפי שאפשר לראות, השליחה של B החלה כבר, אך אז חל עיכוב מסויים. נוצר בין השרת והלקוח זמן מת, שבו אפשר להתחיל להכניס פקטות של משאב נוסף, משאב C. מה שמאפשר את ניצול הזמן הזה הוא שהצד המקבל יכול לדעת שהפקטה שהתקבלה היא של משאב נפרד ואינה המשך של המשאב המושהה, B.

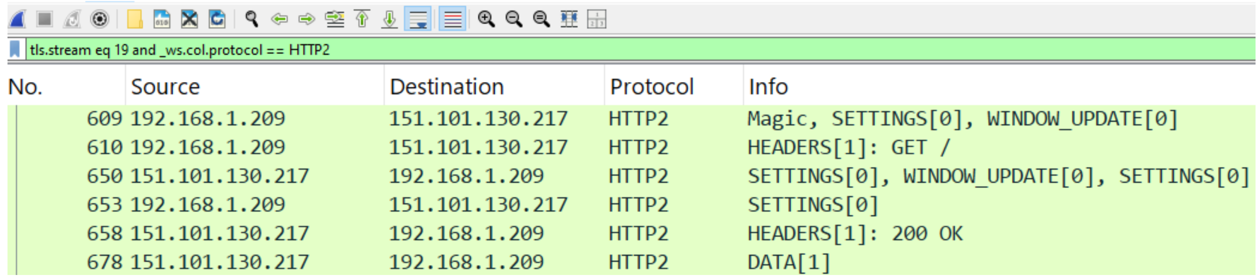


הבה נצפה ב-Stream ID בפעולה.

בצעו הסנפה כלשהי. לא לשכוח לצרף קובץ מפתחות ל-Wireshark, כיוון ש-HTTP/2 מוצפן על-ידי TLS.

בהנחה שההסנפה שלכם כוללת יותר מאשר תקשורת מול שרת יחיד שעובד HTTP/2, רצוי שתפלטרו פקטות HTTP/2 מעל זרם TCP ספציפי. התחילו מחיפוש Client Hello, המשיכו עם Follow TCP Stream, ואם תמצאו פקטות HTTP/2 דייקו את הפילטר כך שיכלול את ה-TCP Stream ID ופקטות מסוג HTTP/2 בלבד:

tcp.stream eq … and _ws.col.protocol == HTTP2



No.	Source	Destination	Protocol	Info
609	192.168.1.209	151.101.130.217	HTTP2	Magic, SETTINGS[0], WINDOW_UPDATE[0]
610	192.168.1.209	151.101.130.217	HTTP2	HEADERS[1]: GET /
650	151.101.130.217	192.168.1.209	HTTP2	SETTINGS[0], WINDOW_UPDATE[0], SETTINGS[0]
653	192.168.1.209	151.101.130.217	HTTP2	SETTINGS[0]
658	151.101.130.217	192.168.1.209	HTTP2	HEADERS[1]: 200 OK
678	151.101.130.217	192.168.1.209	HTTP2	DATA[1]

כפי שרואים, Wireshark מזהה הן את ה-Stream ID, שנמצא בסוגריים מרובעים, והן את סוג ה-Stream. לדוגמה, רואים שני Stream ID שערכם "1", אך אחד מהם הוא HEADERS והאחר DATA.

מפתיחת הפקטה, אפשר לראות שהשדות הללו מוגדרים בפרוטוקול HTTP/2.

שדה ה-Type, בגודל בית יחיד:

```
˅ HyperText Transfer Protocol 2
  ˅ Stream: HEADERS, Stream ID: 1, Length 2763, GET /
      Length: 2763
      Type: HEADERS (1)
    > Flags: 0x25, Priority, End Headers, End Stream
      0... .... .... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1

0000  00 0a cb 01 25 00 00 00  01 80 00 00 00 ff 82 41   ···.%··· ·······A
0010  8d f1 e3 c2 e9 9c b4 8e  ce a5 b1 72 1e 9f 87 84   ········ ···r····
```

שדה ה-Stream ID, בגודל 4 בתים:

```
˅ HyperText Transfer Protocol 2
  ˅ Stream: HEADERS, Stream ID: 1, Length 2763, GET /
      Length: 2763
      Type: HEADERS (1)
    > Flags: 0x25, Priority, End Headers, End Stream
      0... .... .... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1

0000  00 0a cb 01 25 00 00 00  01 80 00 00 00 ff 82 41   ····%··· ·······A
0010  8d f1 e3 c2 e9 9c b4 8e  ce a5 b1 72 1e 9f 87 84   ········ ···r····
```

HTTP PING

אם תחפשו בין פקטות ה-HTTP/2 שפילטרתם, סביר שתמצאו שם פקטות שה-Type שלהן הוא PING.

No.	Source	Destination	Protocol	Info
26541	192.168.1.209	151.101.130.217	HTTP2	PING[0]
26554	151.101.130.217	192.168.1.209	HTTP2	PING[0]

פינג? מה הקשר ל-HTTP? ובכן, למרות שהפקטות הללו נקראות פינג, הן אינן בדיוק פינג אלא מה שנקרא ב-TCP פקטות "Keep Alive".

לעיתים, צד לתקשורת רוצה לבדוק שהקישור בינו לצד השני עדיין קיים. ניתוקים עלולים להתרחש כיוון שרכיב רשת כגון NAT או Firewall החליט לנתק את התקשורת, או כיוון שאחד הצדדים פשוט נסגר בצורה אלימה ולא הספיק לסגור את הסוקט.

ייתכן גם שהתקשורת בין הצדדים היתה דוממת זמן רב ומערכת ההפעלה החליטה לבדוק שהצד השני עדיין פעיל, לפני שהיא סוגרת את הסוקט. מערכות ההפעלה Linux ו-Windows יבדקו שהצד השני עדיין פעיל לאחר שעתיים ללא תקשורת. מכל הסיבות הללו, עוברות לעיתים פקטות Keep Alive ברשת.

כדי להבין מדוע נדרש HTTP PING, נסקור קודם כל איך עובד Keep Alive של TCP, שקדם ל-HTTP PING.

ב-TCP Keep Alive, הצד השולח ישלח פקטת ACK שה-SEQ שלה נמוך באחד ביחס ל-SEQ שאמור להיות בפקטה הבא. לדוגמה, אם ה-SEQ של הפקטה הבאה אמור להיות 100, תישלח פקטת ACK עם SEQ של 99.

הצד המקבל לא יעביר את הפקטה לשכבת האפליקציה, כיוון שמספר ה-SEQ אינו חדש ולכן מבחינת האפליקציה מדובר בשכפול של מידע שהתקבל כבר. עם זאת, המקבל יענה עם TCP ACK על הפקטה שהתקבלה, כאשר הוא מאשר שה-SEQ החדש הוא מה שצפוי להיות, בדוגמה שלנו 100.

הקושי עם פקטות ה-Keep Alive הוא שלעיתים רכיבי רשת עלולים לא להעביר אותם. כמו כן, עלול להיות מצב שבו עד שמערכת ההפעלה מזהה שאין תקשורת בין הצדדים ומבקשת לשלוח TCP Keep Alive, רכיבי רשת מזמן החליטו שעקב חוסר תקשורת הם מנתקים את הקישור. לדוגמה, NAT של ראוטר ביתי יכול למחוק את הרשומה בטבלה שמאפשרת לקשר בין תעבורה נכנסת ויוצאת אל מחשב שמאחרי ה-NAT.

הפתרון שנמצא לבעיות אלו הוא שימוש ב-Keep Alive כחלק מפרוטוקול HTTP/2.

נפתח את הפקטה ששלח הלקוח אל השרת:

```
HyperText Transfer Protocol 2
⌄ Stream: PING, Stream ID: 0, Length 8
    Length: 8
    Type: PING (6)
  › Flags: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0000 0000 = Stream Identifier: 0
    Ping: 0000000000000001
```

כפי שרואים, ה-Type שלה הוא PING, ואחריו יש שמונה בתים שהלקוח בוחר. לבתים אלו אין משמעות פרט לכך שהשרת יחזיר אותם, בדומה לאופן שבו בפרוטוקול ICMP הצד המקבל פינג שולח חזרה את הבתים שהוא קיבל. תגובת ה"פונג" נבדלת מהפינג רק בכך שדגל ה-ACK שלה דולק. לא להתבלבל – זהו דגל ACK שנמצא בפרוטוקול HTTP/2, לא דגל ה-ACK המוכר לנו מ-TCP:

```
HyperText Transfer Protocol 2
⌄ Stream: PING, Stream ID: 0, Length 8
    Length: 8
    Type: PING (6)
  › Flags: 0x01, ACK
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0000 0000 = Stream Identifier: 0
    Pong: 0000000000000001
```

Packed Header

פרוטוקול HTTP הוא פרוטוקול טקסטואלי. כלומר נשלחים תווים דפיסים, בקידוד ASCII או UNICODE, שניתן לקרוא אותם. ה-Header של פרוטוקול HTTP, בו דנו בהרחבה בפרק 4, מכיל ערכים שבדרך כלל חוזרים על עצמם. לדוגמה, הצירוף "index.html".

פרוטוקול HTTP/2 משתמש באלגוריתמי דחיסה כדי לצמצם את כמות הבתים שנשלחת. להסבר מלא על דחיסה מומלץ בחום לקרוא את המאמר הבא, שמסביר לא רק על אלגוריתמי דחיסה אלא גם על איך שימוש בדחיסה יצר פרצות אבטחה, שתרמו לפיתוח גרסאות TLS מתקדמות יותר:

[כשדחיסה היא crime ויוצרת breach באבטחה]

להלן הסבר קצרצר דרך פעולתם של אלגוריתמי דחיסה.

הרעיון הכללי הוא לזהות מחרוזות שחוזרות על עצמן בתדירות גבוהה ולהקצות להן קוד קצר. לעומת זאת, מחרוזות שחוזרות על עצמן לעיתים רחוקות יקבלו קוד ארוך. לדוגמה, אילו היינו רוצים לדחוס את הטקסט בספר זה, סביר שהיינו רוצים להקצות למילה "פקטה" קוד קצר, לדוגמה "אא". חסכנו שתי אותיות על כל פעם שהמילה "פקטה" כתובה. ככל שנשתמש יותר בקוד הקצר, כך החסכון יותר גדול. אא אא אא!

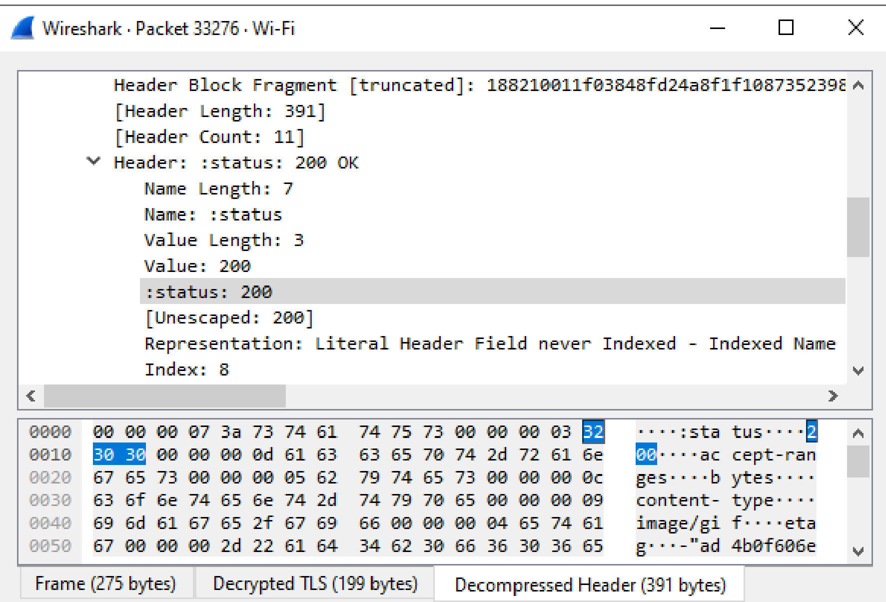
להלן דוגמה לקידוד של ה-Header של HTTP/2. הצירוף "/index.html" כולו מקבל את הקוד "5".

Index	Header name	Header value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html

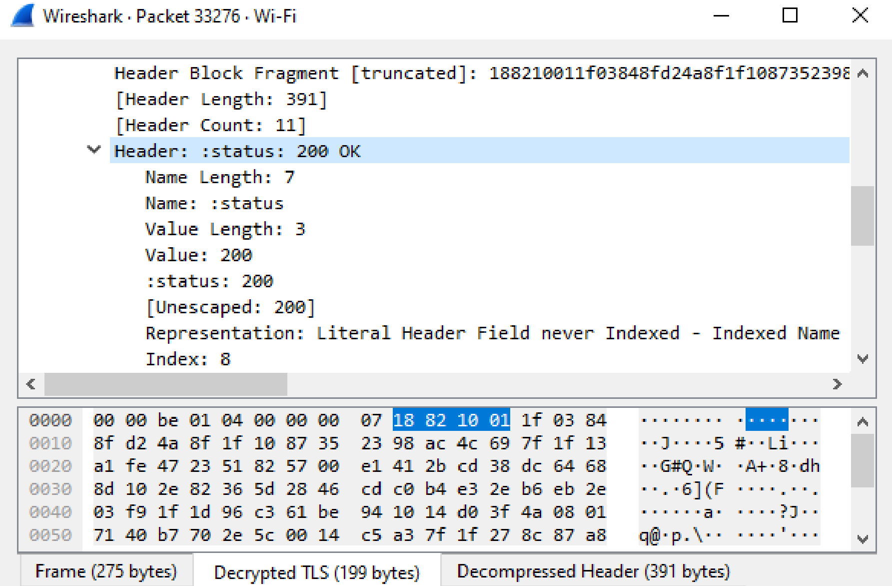
לדחיסת ה-Header של HTTP/2 קוראים "HPACK", קיצור של Header ו-Pack.

כדי לצפות ב-HPACK, בחרו פקטה כלשהי והעבירו בין תצוגה במצב Decrypted TLS למצב Decompressed Header. הבתים בתצוגת Decrypted TLS הם המידע שעבר לאחר שהוסרה ממנו ההצפנה, אך הוא עדיין דחוס.

הנה כך נראית המחרוזת “status 200” במצב Decompressed Header. שימו לב שבין המילים "status" ו-"200" יש ארבעה בתים שערכם 03 00 00 00, שהם שדה האורך של הסטטוס. כלומר סך הכל נדרשים 13 בתים להעברת המידע:



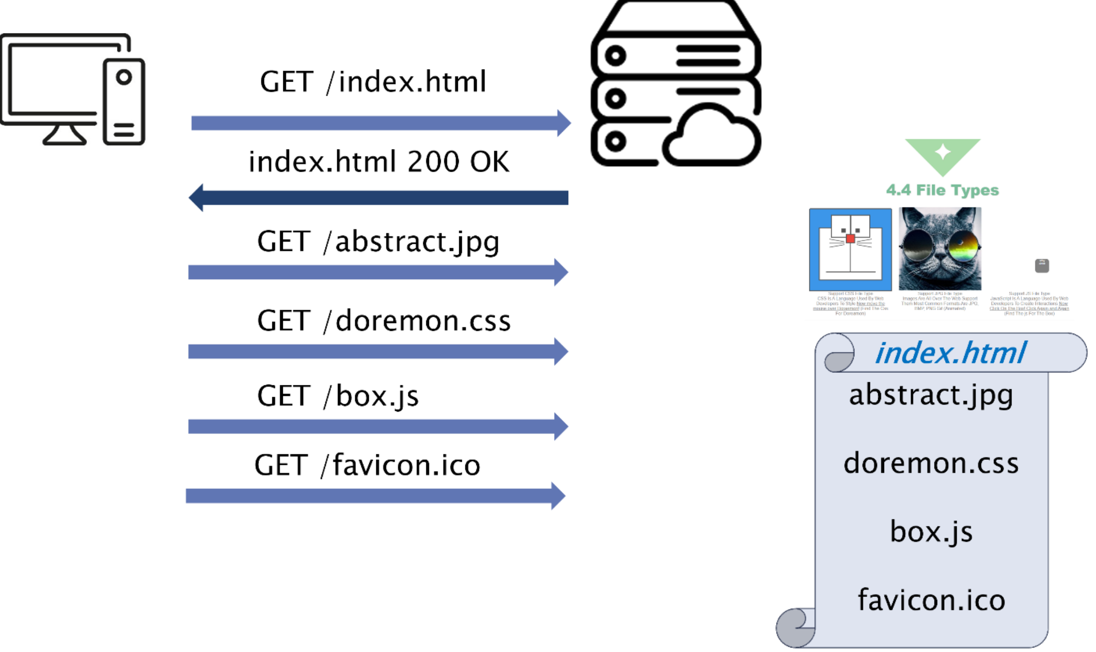
וכעת אותו ה-Header בצורתו הדחוסה:



אותם 13 בתים נדחסו כעת ל-4 בתים בלבד. סך הכל, ה-Header תופס במקרה זה 391 בתים בצורתו הלא המקורית ו-199 בתים בצורתו הדחוסה. החיסכון הזה אולי נראה קטנטן אבל הוא מוכפל בכמות פקטות ה-HTTP שהמחשב שלכם, וכל המחשבים האחרים בעולם, שולחים.

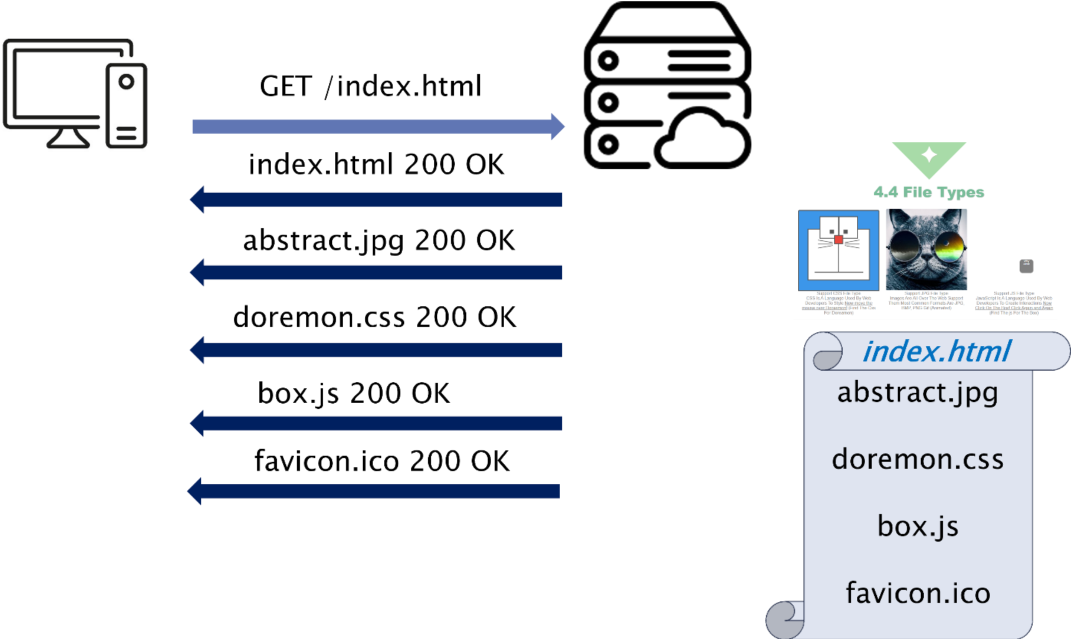
Server Push

בגרסה 1.1, שליחת המידע מהשרת ללקוח מתבצעת רק כמענה לבקשות הלקוח. ניקח לדוגמה את אתר האינטרנט הלימודי ששימש אותנו לכתיבת שרת HTTP בחלקו הראשון של ספר רשתות, תרגיל 4.4. הלקוח פונה אל השרת ומבקש את המשאב index.html. עם הורדת המשאב, הלקוח מגלה שיש לו צורך בארבעה משאבים נוספים והוא מוציא גם אליהם בקשת GET:



גרסה 2 מייעלת את התהליך. השרת הרי יודע מראש שאם הוא שלח את index.html, הלקוח עומד מיד לגלות שהוא צריך ארבעה משאבים נוספים ולבקש אותם. למה לא לחסוך את הבקשות ואת ה-RTT שהן ידרשו?

בגרסה 2 יש שימוש ב-Server Push. השרת, שצופה כבר מראש מה הלקוח יבקש, "דוחף" לו את המשאבים עוד טרם הבקשה:



עד כאן לגבי השינויים המרכזיים בין HTTP/1.1 ל-HTTP/2. אך עדיין יש מספר בעיות מרכזיות שלא טופלו, הפ ונתייחס אליהן בהמשך, בפרק אודות HTTP/3.

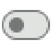
DNS Over HTTPS

הסנפה של DNS רגיל

לפני שנסביר כיצד הדפדפן שלנו מבצע בקשת DNS כיום, נגרום לו לחזור לעמט אחורה בזמן ולהוציא בקשות DNS רגילות. הכנסו להגדרות הדפדפן ובטלו את האפשרות לבצע Secure DNS.

בכרום – settings, ואז privacy and security, ומשם Use Secure DNS, ובטלו את האופציה.

Use secure DNS
Make it harder for people with access to your internet traffic to see which sites you visit. Chrome uses a secure connection to look up a site's IP address in the DNS (Domain Name System).



הפעילו Wireshark וגלשו לאתר כלשהו. תוכלו לראות את בקשת ה-DNS בהסנפה:

Source	Destination	Protocol	Info
192.168.1.209	192.168.1.1	DNS	Standard query 0xcb3d A www.kan.org.il
192.168.1.1	192.168.1.209	DNS	Standard query response 0xcb3d A www.kan.org.il A 172.66.43.92 A 172.66.40.164

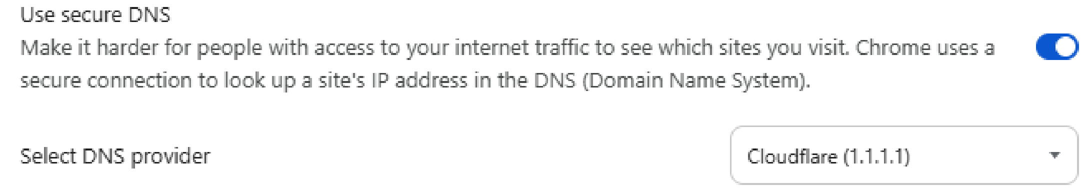
שימו לב, אם אינכם מוצאים את הפקטות בהסנפה ייתכן כי כתובת ה-IP של האתר עדיין נמצאת ב-Cache של הדפדפן. בחרו אתר שלא גלשתם אליו לפני זמן קצר.

למדנו מהניסוי הקטן שלנו שכאשר אנו גולשים לאתר כלשהו, הדפדפן מוציא אוטומטית שאילתת DNS על שרת ה-web של האתר.

כעת נתקדם בניסוי הקטן שלנו אל העבר הקרוב יותר.

הסנפה של DNS מעל HTTP

החזירו בדפדפן את האפשרות של Use Secure DNS. תוכלו לבחור באחד מספקי השירות, כגון שרת ה-DNS של גוגל, 8.8.8.8, שרת ה-DNS של Cloudflare, 1.1.1.1. נכון למועד כתיבת שורות אלו, מומלץ להשתמש בשירות של Cloudflare, שמשלמת לביקורת חיצונית כדי לוודא שאין שימוש לרעה בנתוני הגלישה שלכם.



כמו כן, בטלו את האפשרות של פרוטוקול QUIC. נלמד עליו בהמשך, בשלב זה איננו רוצים שיופיעו בהסנפה פקטות שאיננו יודעים מה הן.

ביטול QUIC בדפדפן כרום:

chrome://flags/#enable-quic

והעבירו את הבחירה למצב disabled:

Experimental QUIC protocol
Enable experimental QUIC protocol support. – Mac, Windows, Linux, ChromeOS, Android
#enable-quic

החלו בהסנפה חדשה וגילשו לאתר כרצונכם.

כעת, לא תמצאו את שאילתות ה-DNS שלכם בהסנפה, מכיוון שהשאילתות כבר אינן עוברות מעל DNS. אבל, אם הוספתם ל-Wireshark את קובץ המפתחות TLS, תוכלו למצוא פקטות שהפרוטוקול שלהן הוא DoH, הלא הוא DNS over HTTPS.

צפייה ב-DoH

תוכלו להשתמש בהסנפה שביצעתם, או בהסנפה הבאה:

https://data.cyber.org.il/networks/DoHfiles.rar

חפשו פקטות DoH

_ws.col.protocol == "DoH"

ובצעו Follow HTTP stream:

Source	Destination	Protocol	Info
192.168.1.209	162.159.61.3	HTTP2	HEADERS[51]: POST /dns-query
192.168.1.209	162.159.61.3	DoH	Standard query 0x0000 A www.kan.org.il OPT
162.159.61.3	192.168.1.209	HTTP2	HEADERS[51]: 200 OK
162.159.61.3	192.168.1.209	DoH	Standard query response 0x0000 A www.kan.org.il A 172.66.43.92 A 172.66.40.164 OPT

בקשת ה-DNS שאנו רואים מחולקת למספר פקטות. כפי שאפשר לראות, המידע שעובר בין השרת והלקוח נראה כמו תקשורת HTTP רגילה.

Wireshark · Follow HTTP2 Stream (tcp.stream eq 96 and http2.streamid ...

```
:method: POST
:authority: chrome.cloudflare-dns.com
:scheme: https
:path: /dns-query
content-length: 128
accept: application/dns-message
accept-language: *
user-agent: Chrome
accept-encoding: identity
content-type: application/dns-message
priority: u=0, i

............www.kan.org.il..A....).......U...Q..........................
..................................................

:status: 200
server: cloudflare
date: Fri, 31 Oct 2025 13:45:13 GMT
content-type: application/dns-message
access-control-allow-origin: *
x-content-type-options: nosniff
content-length: 468
cf-ray: 9973977398c79b09-TLV
alt-svc: h3=":443"; ma=86400

............www.kan.org.il..A.....A.......:........h2.....B(..B+\... &.G.
1........B(.&.G.1........B+\..).......c..._.............................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
......................
```

פקטה ראשונה – ביצוע POST אל השרת. ניזכר באופן הפעולה של HTTP: בקשות POST, כמו GET, כוללות את ה-Host שפונים אליו ואת ה-URL המבוקש מה-Host. במקרה זה אנו פונים ל--chrome.cloudflare-dns.com/dns-query, והפנייה מחולקת בין שדה ה-authority ושדה ה-path.

פקטה שניה – שאילתת ה-DNS עצמה. השאילתא מופיעה כ-DNS מעל HTTP 😊 הסוג של השאילתא הוא A, כלומר IPv4.

```
HyperText Transfer Protocol 2
  Stream: DATA, Stream ID: 51, Length 128
  Domain Name System (query)
    Transaction ID: 0x0000
    Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 1
    Queries
      www.kan.org.il: type A, class IN
```

פקטה שלישית – השרת משיב 200 OK, בהתאם לפרוטוקול HTTP.

פקטה רביעית – השרת מספק את התשובה.

```
HyperText Transfer Protocol 2
  Stream: DATA, Stream ID: 51, Length 468
  Domain Name System (response)
    Transaction ID: 0x0000
    Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 2
    Authority RRs: 0
    Additional RRs: 1
    Queries
      www.kan.org.il: type A, class IN
    Answers
      www.kan.org.il: type A, class IN, addr 172.66.43.92
        Name: www.kan.org.il
        Type: A (1) (Host Address)
        Class: IN (0x0001)
        Time to live: 23 (23 seconds)
        Data length: 4
        Address: 172.66.43.92
      www.kan.org.il: type A, class IN, addr 172.66.40.164
        Name: www.kan.org.il
        Type: A (1) (Host Address)
        Class: IN (0x0001)
        Time to live: 23 (23 seconds)
        Data length: 4
        Address: 172.66.40.164
```

אך הלקוח לא מסתפק בכך והוא מוציא במקביל גם בקשת DNS מסוג שלא הכרנו עד כה – "HTTPS". ניתן לראות בפקטה השניה, שסוג הבקשה הוא HTTPS ולא A:

Source	Destination	Protocol	Info
192.168.1.209	162.159.61.3	HTTP2	HEADERS[49]: POST /dns-query
192.168.1.209	162.159.61.3	DoH	Standard query 0x0000 HTTPS www.kan.org.il OPT
162.159.61.3	192.168.1.209	HTTP2	HEADERS[49]: 200 OK
162.159.61.3	192.168.1.209	DoH	Standard query response 0x0000 HTTPS www.kan.org.il HTTPS OPT

בקשה זו נועדה לבדוק מהו הפרוטוקול של שכבת האפליקציה שהשרת המיועד תומך בה. התשובה תינתן בשדה ה-ALPN – קיצור של Application Layer Protocol Negotiation:

```
⌄ Answers
  ⌄ www.kan.org.il: type HTTPS, class IN
      Name: www.kan.org.il
      Type: HTTPS (65) (HTTPS Specific Service Endpoints)
      Class: IN (0x0001)
      Time to live: 284 (4 minutes, 44 seconds)
      Data length: 58
      SvcPriority: 1
      TargetName: <Root>
    > SvcParam: alpn=h2
    > SvcParam: ipv4hint=172.66.40.164,172.66.43.92
    > SvcParam: ipv6hint=2606:4700:3108::ac42:28a4,2606:4700:3108::ac42:2b5c
```

תשובת השרת מראה לנו כי תומך בפרוטוקול HTTP/2, או בשמו המקוצר "H2".

בנוסף, השרת נתן לנו מידע על כתובות ה-IP של השרת שאליו אנחנו פונים. במקרה זה, הכתובות זהות לכתובות ה-IP שקיבלנו בשאילתא מסוג A. עם זאת, המידע הזה יכול להיות לא מדויק ברשומה מסוג HTTPS ולכן הדפדפן שלנו הוציא גם בקשת A.

DoH עם Base64

בהספה שניתחנו, כדי להוציא בקשה, הלקוח שלח שתי פקטות. תחילה HTTP עם מתודת POST ולאחר מכן פקטת DoH.

קיימת אופציה נוספת של DoH, בה נשלחת פקטה יחידה. הפקטה תראה כך:

GET /dns-query?dns=<base64url-encoded-message>

שרת ה-DNS של גוגל משתמש באפשרות זו. במקרה זה, כבר לא ניתן להפעיל פילטר של DoH מכיוון שכל הבקשות ייראו כ-HTTP. כמו כן, Wireshark כבר אינו מציג את שם הדומיין עליו התבצעה השאילתא, אלא מציג את הדומיין מקודד Base64.

כדי לבצע את ההסנפה בלי להתקל בפרוטוקול שטרם דנו בו, שימו לב שעדיין מבוטל השימוש ב-QUIC בדפדפן.

גילשו לאתר לבחירתכם, תוך כדי הסנפה (וכמובן קובץ מפתחות מוזן לתוך Wireshark).

נבחן את הפקטה הבאה שנשלחה לכתובת 8.8.4.4, היא אחת מכתובות ה-IP של google.dns:

Source	Destination	Protocol	Info
192.168.1.103	8.8.4.4	HTTP2	HEADERS[33]: GET /dns-query?dns=AAABAAABAAAAAAABBGh1amkCYWMCaWwAAE

כפי שרואים המשאב המבוקש בבקשת ה-GET הוא /dns-query. לאחר מכן, יש פרמטר (היזכרו בפרק HTTP, "בקשת GET עם פרמטרים") בשם dns, שערכו הוא מחרוזת ארוכה.

להלן המחרוזת במלואה. העתיקו אותה לתוך Base64 decoder כלשהו מהרשת:

AAABAAABAAAAAAABBGh1amkCYWMCaWwAAEEAAQAAKRAAAAAAAABZAAwAVQAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

בפענוח, תוכלו לראות את שם הדומיין עליו מתבצעת השאילתא – huji.ac.il, וכן את סוג השאילתא – A, שהוא כזכור מציין IPv4.

נעבור לתגובת השרת:

Source	Destination	Protocol	Info
8.8.4.4	192.168.1.103	DoH	Standard query response 0x0000 A huji.ac.il A 128.139.7.7 OPT

איזה תענוג. מעל TLS יש HTTP/2, בתוכו יש DNS כמו שמוכר לנו:

```
› Transport Layer Security
⌄ HyperText Transfer Protocol 2
  › Stream: DATA, Stream ID: 35, Length 0
  ⌄ Domain Name System (response)
      Transaction ID: 0x0000
    › Flags: 0x8180 Standard query response, No error
      Questions: 1
      Answer RRs: 1
      Authority RRs: 0
      Additional RRs: 1
    ⌄ Queries
      › huji.ac.il: type A, class IN
    ⌄ Answers
      › huji.ac.il: type A, class IN, addr 128.139.7.7
```

ושוב הדפדפן שלנו מקבל את כתובת ה-IP של השרת המבוקש.

22.1 תרגיל: DoT

עד כה, דיברנו על DNS מעל HTTP, שנקרא גם DoH. בפרוטוקול DoT מעבירים DNS *ישירות* מעל TLS, בלי צורך בתיווך של HTTP. דבר זה חוסך את ה-HTTP Header.

מעל TCP נמצא TLS. מעל TLS יש שני שדות בלבד: שדה אורך, ובקשת DNS.

בתרגיל זה נתכנת DoT.

<u>הסבר כללי</u>

בתרגיל זה נתבסס על התרגיל המסכם "סוקטים מאובטחים" שבסוף פרק 18, ונוסיף לו יכולת שימושית – ביצוע DNS. בתרגיל סוקטים מאובטחים יצרתם שרת ולקוח, שביצעו החלפת מפתח סודי. בסיום התרגיל, הלקוח שלח לשרת מסר כלשהו, שעבר הצפנה ונוסף לו HMAC.

שאלה זו מתחילה היכן שהקמת הסוקט המאובטח מסתיימת. כעת, השרת אינו סתם שרת אלא שרת שיודע לבצע העברת שאילתות ותגובות DNS (כלומר DNS Relay). השרת מקבל מהלקוח בקשה לדומיין מסויים ומחזיר ללקוח את כתובות ה-IP של הדומיין.

<u>הלקוח</u>

יקים סוקט מאובטח עם השרת. לטובת בקרה הדפיסו למסך את הסוד המשותף שתואם עם השרת.

לאחר הקמת הסוקט המאובטח, הלקוח יבקש מהמשתמש להזין קלט – שם דומיין, או EXIT על מנת לסיים.

אם המשתמש ביקש EXIT, ההודעה תועבר לשרת והלקוח יסיים.

אחרת, הלקוח שולח לשרת בהודעה אחת:

- אורך המידע שנמצא בהמשך. שדה האורך יהיה טקסטואלי, ויכלול 3 ספרות. לדוגמה "213" יציין שיש 213 בתים בהמשך.
- מחרוזת הכוללת את הקלט של הלקוח
- סוג בקשת ה-DNS (בתרגיל זה הבקשה תמיד תהיה מסוג A)

הודעה זו כאמור תהיה מוצפנת ותכלול HMAC.

כעת, הלקוח מאזין לתשובת השרת. כשהשרת משיב, הלקוח מקבל את תגובת השרת ומדפיס למסך את כל כתובות ה-IP שקיבל.

השרת

השרת עובד מול לקוח אחד. אין צורך להשתמש ב-select, ריבוי תהליכים וכו'. השרת מאזין ללקוח, מקבל את ההודעות שלו בלולאה ומסיים ריצה כאשר מתקבל EXIT.

השרת יקים סוקט מאובטח עם הלקוח. לטובת בקרה יש להדפיס למסך את הסוד המשותף שתואם עם הלקוח.

עבור כל הודעה שנשלחת אליו, השרת:

- פותח את ההצפנה, מדפיס את ה-HMAC ומוודא HMAC תקין (חלק זה ממילא בוצע בתרגיל "סוקטים מאובטחים"). לא נדרש טיפול במצב שה-HMAC אינו תקין, רק דיווח למשתמש.
- מחלץ את התוכן מתוך הבקשה (שם דומיין וסוג בקשה, או EXIT).
- במידה וקיבל EXIT, השרת מסיים ריצה.
- יוצר באמצעות scapy בקשת DNS ושולח אותה אל שרת DNS כלשהו.
- מקבל את תשובת שרת ה-DNS, מחלץ את כתובות ה-IP.
- **שולח ללקוח הודעת תשובה הכוללת את כתובות ה-IP מסוג A של הדומיין, ורק אותן. שימו לב שיכולה להיות יותר מכתובת אחת.**

גם תגובת השרת **תהיה מוצפנת ותכלול HMAC**.

טיפול במצבים מיוחדים:

א. אם הדומיין אינו קיים, יש להעביר ללקוח כתשובה את המחרוזת "No such domain".

ב. אם השרת לא קיבל כלל תגובה משרת ה-DNS, יש לזהות זאת ולשלוח ללקוח כתשובה את המחרוזת "No response".

בהצלחה!

דוגמאות הרצה:

1. דומיין שיש לו כתובת IP יחידה – www.jct.ac.il:

```
Shared secret is: 9222
Enter domain, or EXIT
www.jct.ac.il
Message HMAC:  86f09ab7ba44dd3907167c6e75687ee88b15e243
Computed HMAC:  86f09ab7ba44dd3907167c6e75687ee88b15e243
HMAC matches

Server response:
 185.186.66.220
```

2. דומיין עם מספר כתובות IP – www.youtube.com:

```
Enter domain, or EXIT
www.youtube.com
Message HMAC:  0a17e3c6379defc438b5f0025d4ac3c78c8efd44
Computed HMAC:  0a17e3c6379defc438b5f0025d4ac3c78c8efd44
HMAC matches

Server response:
 142.250.75.174
142.250.75.46
142.250.75.206
142.250.75.78
142.250.75.110
142.250.75.142
```

3. דומיין לא קיים – blabla.jct.ac.il:

```
Enter domain, or EXIT
blabla.jct.ac.il
Message HMAC:  a235ca76bc29983dc3fbe1511a6c1375dbffe2cc
Computed HMAC:  a235ca76bc29983dc3fbe1511a6c1375dbffe2cc
HMAC matches

Server response:
 Domain not found
```

4. לא התקבלה תשובה בזמן מוגדר (ניתקו את השרת מהרשת, הקטינו את ה-Timeout לקבלת התגובה משרת ה-DNS או שלחו את בקשת ה-DNS לשרת שאינו קיים):

```
Enter domain, or EXIT
www.amazon.com
Message HMAC:  acd397b932420cc5ff5ed5213c99af19152e9c89
Computed HMAC:  acd397b932420cc5ff5ed5213c99af19152e9c89
HMAC matches

Server response:
 No response
```

5. סיום יפה:

```
Enter domain, or EXIT
EXIT
Closing


Process finished with exit code 0
```

סיכום

בפרק זה סקרנו את השינויים בשני פרוטוקולים מרכזיים – HTTP ו-DNS.

ראינו כי HTTP עבר גלגולים שונים, וכי גרסה 2 מאיצה את הגלישה בשיטות שונות: שימוש ב-Stream ID, ביצוע Server Push, דחיסה של ה-HTTP Header. כמו כן פרוטוקול HTTP/2 לוקח עליו את אחת המשימות של TCP ומבצע Keep Alive. בפרק הבא ניתקל בפרוטוקול שלא רק שלוקח לעצמו משימות של TCP אלא למעשה מייתר אותו לחלוטין.

חקרנו איך נראית בקשת DNS עכשווית באמצעות הסנפה של שתי האופציות של DoH. בשתיהן הדפדפן שלנו קיבל את כתובת ה-IP בצורה מאובטחת. מי שמאזין לתקשורת שלנו יוכל לראות רק פקטות שמיועדות לשרת ה-DNS של גוגל, בלי יכולת להבין לאיזה אתר אנחנו רוצים לגלוש. הפרוטוקולים החדשים שלמדנו עושים כל מאמץ להקשות על מי שמאזין לנו להוציא פרטי מידע על הגלישה שלנו.

במהלך הפרק, כדי שנוכל להתמקד בחומר הנלמד בלי להתבלבל עם פרוטוקול לא מוכר, ביטלנו אופציה של שימוש בפרוטוקול QUIC. זה יהיה הנושא של הפרק הבא.

פרק 23: QUIC

הקדמה

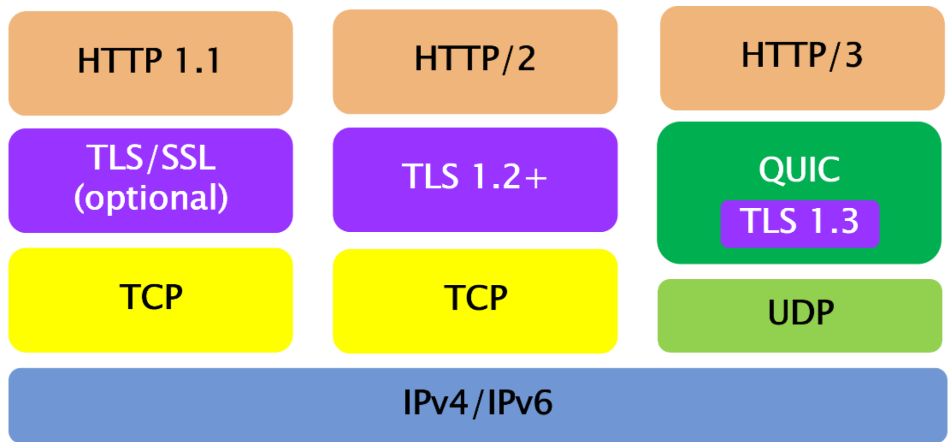
בפרקים הקודמים כיסינו את ההתקדמות הטכנולוגית שהיתה בין אמצע שנות ה-90, כאשר תעבורת האינטרנט היתה לא מאובטחת, ועד 2018, השנה שבה פורסם תקן TLS בגרסה 1.3.

בפרק זה נתקדם עוד כמה שנים קדימה, לשנת 2021 ולפרוטוקול חדש שפורסם ואשר נכון למועד כתיבת שורות אלה הוא הפרוטוקול העדכני והמתקדם ביותר. כאשר נסיים ללמוד אותו, נהיה בנקודה שבה יש לנו הבנה בפרוטוקול החדש ביותר שיוצר סוקטים מאובטחים.

שם הפרוטוקול – QUIC. ראשי תיבות של Quick UDP Internet Connections.

רגע אחד! UDP? הרי UDP אינו פרוטוקול אמין. כל מה שלמדנו עד כה הביא אותנו לתובנה שכל פרוטוקול שדורש אמינות חייב להיות מעל TCP. ובכן, נכון, פרוטוקול QUIC שובר הנחת יסוד משמעותית שלמדנו בכך שהוא עובד מעל UDP. ולא רק זאת, אלא ש-QUIC גם שובר את מודל השכבות המוכר לנו. פרוטוקול שנמצא מעל שכבת התעבורה, סופח אליו כמעט את כל התפקידים שהיו עד כה שמורים ל-TCP.

האיור הבא נותן מושג על השינויים שעברנו ועל השינוי שאנחנו עומדים לעסוק בו בפרק זה.



מצד שמאל – האינטרנט הלא מאובטח, או המאובטח בגרסת TLS ישנה כלשהי. רואים יפה את מודל השכבות המוכר, כאשר מעל IP יש TCP, מעליו יש TLS ישן כלשהו (או אין, תלוי כמה אחורה הולכים בזמן), ומעליו HTTP גרסה 1.1.

באמצע – המעבר ל-TLS גרסה 1.2 או 1.3, ויחד איתו גרסה 2 של HTTP. עדיין השכבות למטה אינן משתנות: IP ומעליו TCP.

מצד ימין – תוהו ובוהו. לקחנו את המבנה הקודם, ששירת אותנו עשרות שנים, ושברנו אותו. החלפנו את TCP ב-UDP. מעליו יש משהו שנקרא QUIC, שבאופן מוזר מאוד כולל **בתוכו** את TLS 1.3. מעליו – גרסה חדשה של HTTP, גרסה 3. הפרוטוקול היחיד שנותר מערימת הפרוטוקולים המקורית הוא IP.

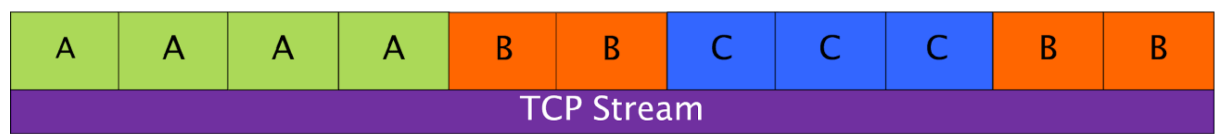
במהלך הפרק נענה על אוסף שאלות מעניינות שיחולקו למספר חלקים. בחלקו הראשון של הפרק, נבין מה היו החסרונות והבעיות של HTTP/2 מעל TLS 1.3 מעל TCP. כך נבין מדוע המעבר ל-UDP הוא מוצלח ואף הכרחי. חלקו השני של הפרק ידון בפרוטוקול TCP, מכיוון שאם רוצים לבטל את TCP אז צריך להבין מה היכולות של TCP שנצטרך למצוא להן תחליף. בחלק השלישי נסקור את QUIC בליווי הסנפה, ונראה כיצד מיושמים העקרונות של TCP בלי שימוש ב-TCP. נעבור על כל המוטיבציות לשינוי מהמצב ששרר לפני QUIC ונראה כיצד השינויים עונים על הבעיות. בחלק הרביעי נסקור את השינויים שחלו ב-HTTP בין גרסה 2 לגרסה 3, ונסיים בצפייה ב-DNS מעל QUIC.

HTTP/2 – מקומות לשיפור

בפרק הקודם הצגנו את ההתקדמות המשמעותית שהיתה בין HTTP/1.1 ל-HTTP/2. כעת, נסקור את הדברים המרכזיים שצריך לשפר ב-HTTP/2, אשר רובם המכריע אינם קשורים ישירות ל-HTTP/2 אלא לפרוטוקול TCP, שנמצא בחבילת הפרוטוקולים עליהם HTTP/2 מתבסס.

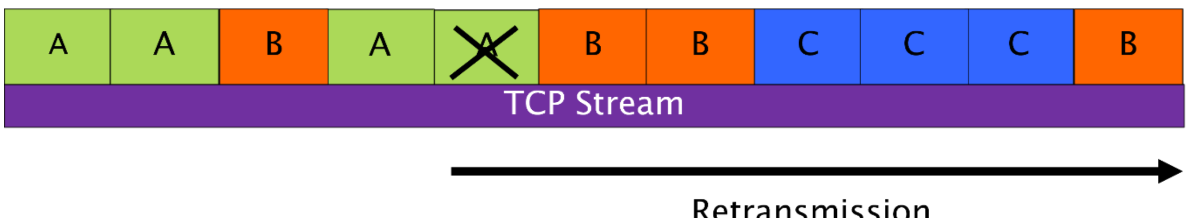
בעיית ה-Head Of Line Blocking

HTTP/2 משתמש, כפי שראינו, בקישור TCP יחיד שעליו מועברים זרמי מידע שונים. כדי להפריד בין זרמי המידע, לכל זרם יש stream ID שהוא חלק מהפרוטוקול, ואשר מאפשר לצד המקבל לאסוף יחד פקטות ששייכות לאותו זרם. דבר זה מאפשר לנצל בצורה מיטבית את הערוץ, מכיוון שגם אם משאב אחד "תקוע", או נמצא בהמתנה, אפשר לנצל את הערוץ כדי להעביר משאבים אחרים.



נראה שהשגנו ניתוק בין הזרמים השונים, כל זרם מידע יכול לעבור בלי תלות בזרמי המידע האחרים. אך האמנם זה באמת המצב?

דמיינו תור של אנשים הממתינים לשירות של קופאי בבנק. הקופאי מטפל בלקוח, אבל בגלל בעיה טכנית הפעולה מתארכת. כל הממתינים בתור לקופאי צריכים לחכות, למרות שיכול להיות שהם צריכים שירותים אחרים, שאין בעיה טכנית לבצע אותם. מי שנמצא בראש התור יוצר "פקק" לכל התור. דבר זה נקרא Head of Line Blocking.
וכעת נחזור לעולם התקשורת. האיור הבא ממחיש מה קורה כאשר פקטת TCP אחת נפלה.



למרות שכל הפקטות שאחריה הגיעו בצורה תקינה ליעד, הן עלולות להיזרק והשולח יצטרך לשלוח אותן מחדש. כן, גם פקטות שנושאות Stream ID של משאב B או של משאב C, שאין שום בעיה לשמור אותן ולהשתמש בהן, עלולות להיזרק. אם הדבר אינו ברור לכם, בהמשך הפרק נסביר על ה-Cumulative ACK של TCP ונבין מדוע אובדן של פקטה מעכב את הטיפול בפקטות הבאות. פרוטוקול TCP מייצר תלות בסיסית בין זרמי המידע, בצורה שגם הפרדה מאוחרת יותר בשכבת האפליקציה אינה יכולה לתקן. פקטה שאבדה היא כמו לקוח שנתקע אצל הקופאי, יוצרת Head Of Line Blocking.

TCP Ossification

האם אפשר איכשהו לתקן את TCP, או להוסיף לו שיפור כלשהו, שיפתור את בעיית ה-Head of Line Blocking? תאורטית כן, מעשית – בעיה גדולה. כדי להסביר מדוע זה קשה מאוד, נסקור את תהליך ה-Ossification של TCP וניתן דוגמה לשיפור יפה של TCP, שיכול היה להיות קיים, אך פקטות שכללו את השיפור הזה לא הגיעו ליעדן.

המונח Ossification פירושו "התגרמות", מהמילה "גֶּרֶם", שפירושה "עצם". לדוגמה, מישהו שהוא גרום הוא מישהו רזה במידה שהעצמות בולטות ממנו. תהליך ההתגרמות הוא תהליך טבעי שבו רקמת סחוס הופכת לעצם. תינוקות נולדים עם עצמות קטנות והרבה סחוס ביניהן. בתהליך הגדילה וההתבגרות הסחוס מתגרם לעצם, והתהליך מסתיים בתום ההתבגרות כאשר אין כמעט רווח בין העצמות.

הכוונה בפרוטוקול שעבר Ossification היא שהוא הגיע למצב של בגרות כזו, שאיבד את הגמישות שלו וקשה להכניס לתוכו תוספות או שינויים.

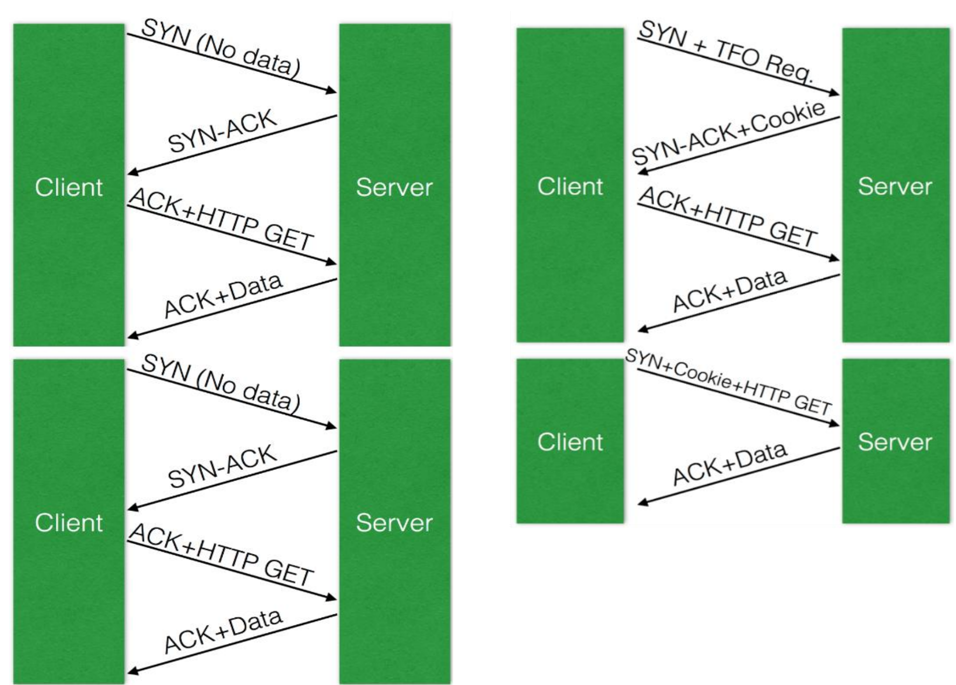
כדי להבין למה ואיך פרוטוקול TCP עבר Ossification, צריך להפנות את המבט אל הרכיבים שנמצאים בין השרת והלקוח, ה-Middleboxes. כל הרכיבים כמו Firewall, NAT, רכיבים שמבצעים תרגום בין פרוטוקולים, רכיבי DPI – Deep packet inspection, ועוד ועוד. רכיב כזה עשוי לעשות את עבודתו במשך שנים רבות ואז להיתקל

בפקטה שכוללת פרוטוקול לא מוכר, או איזושהי המצאה חדשה שמבוססת על פרוטוקול מוכר אך עובדת קצת אחרת. במקרה זה, הרכיב עשוי להעביר את הפקטה כמו שהיא, אך הוא עלול גם להשליך אותה או לשנות אותה. וזה עלול להיות קטלני להמצאות חדשות.

כבר כאשר דנו בשדות של TLS record, ראינו שאחד השדות הוא גרסת ה-TLS וראינו שהערך של שדה זה אינו תמיד תואם את הגרסה האמיתית של פרוטוקול ה-TLS שהוא מעביר. זו היתה דוגמה להתחלה של Ossification של פרוטוקול TLS.

פרוטוקול TCP הוא ותיק בעשרות שנים מ-TLS ואותם Middleboxes למדו אותו היטב, מה יש בו ומה אין בו. וכאשר מנסים לשלוח פקטת TCP עם משהו חדש, המשהו החדש הזה אינו תואם לידע של מה יש ומה אין בפרוטוקול. לכן, יש סיכוי לא מבוטל ששינויים וחידושים ב-TCP ייכשלו במבחן המעשי של זרימה ברשת האינטרנט. דוגמה קלאסית לכך הוא מה שנקרא TCP Fast Open, או בקיצור TFO. הרעיון של TFO הוא לחסוך RTT במקרה של הקמת סוקט שנסגר לאחרונה, באמצעות קיצור ה-Three Way Handshake.

האיור הבא מדגים את התהליך. מצד שמאל המצב כיום, הקמת קישור, סגירה שלו, הקמת קישור חדש. מימין – TFO. הקמת קישור ואז קישור מקוצר.



מקור: reproducingnetworkresearch

המצב ללא TFO הוא שלקוח מתחבר לשרת לאחר שמקיים Three Way Handshake, בעל RTT של 1. נניח שהלקוח התנתק ורוצה ליצור קישור חדש, עליו לחזור על ה-Three Way Handshake ולשלם ב-RTT נוסף.

ב-TFO, ה-Three Way Handshake הראשון עדיין נמשך RTT אחד, אבל הוא שונה במקצת. **על גבי** פקטת ה-TCP Syn, הלקוח יוסיף מידע שמשמעותו "בקשת TFO". אם השרת תומך ב-TFO, השרת יענה עם TCP Syn-Ack הכולל מחרוזת כלשהי, שהינה ייחודית לכל לקוח ולכל הקמת קישור. מחרוזת זו נקראת TFO Cookie.

כעת הלקוח התנתק ורוצה ליצור קישור מחדש. הוא שולח בקשת TCP Syn שכוללת את ה-TFO Cookie שהשרת שלח. ה-Cookie מציין "היי שרת, אנחנו כבר מכירים, בוא נדלג על ה-Three Way Handshake". בנוסף, הלקוח כבר שולח על ה-TCP Syn מידע של שכבת האפליקציה, כגון HTTP GET. אם הקמת הקשר המקוצר מצליחה, אז הפקטה הבא שהשרת ישלח תהיה TCP Ack שכבר כוללת את התגובה לבקשה של הלקוח. קבלנו 0-RTT (ניזכר כי הכוונה שיש אפס הלוך ושוב לפני שהלקוח יכול לשלוח מידע של שכבת האפליקציה).

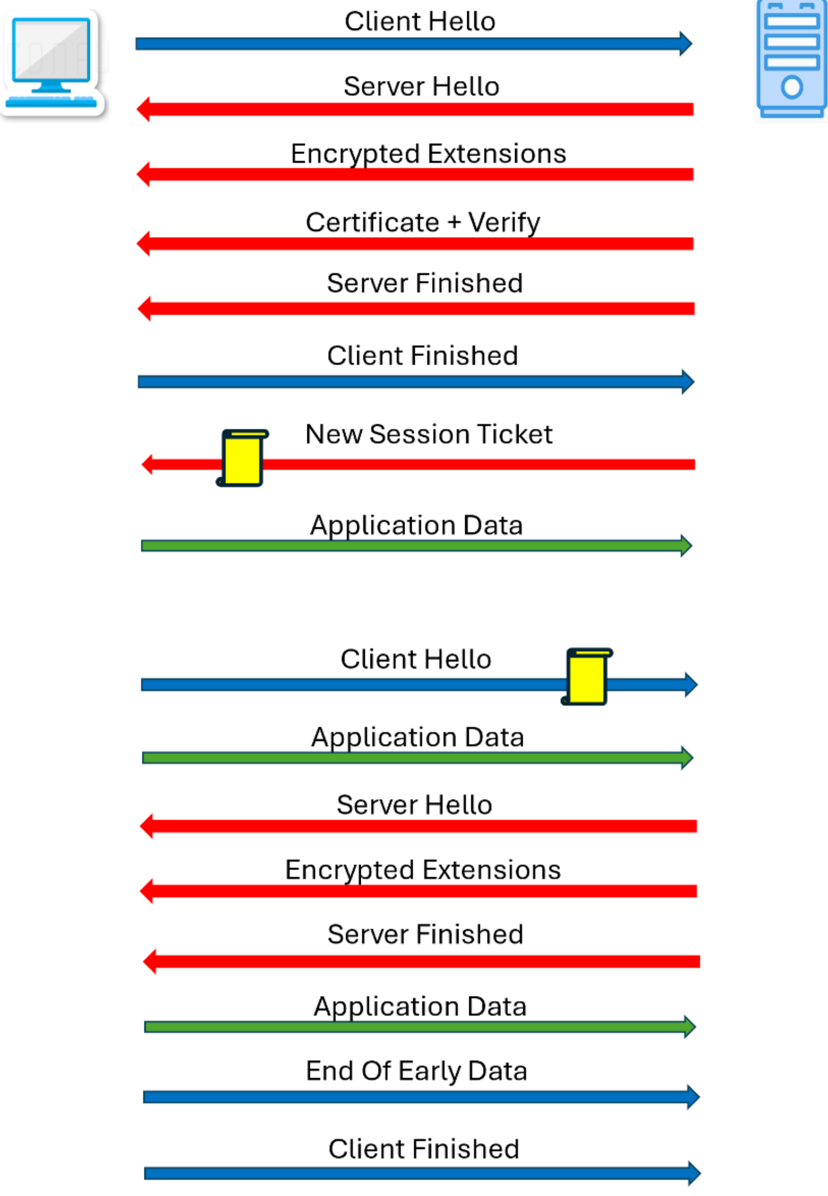
כאן נכנס לתמונה גורם ה-Ossification. הרעיון היפה של ה-TFO לא עבד בשטח. בערך 20% מהניסיונות להקים קישורי TFO לא עבדו. הסיבה לכך היא שאותם Middleboxes סברו ש"היי, מה פתאום יש מידע על גבי פקטת TCP Syn או TCP Syn Ack? זה ממש לא איך שפרוטוקול TCP אמור לעבוד. סכנה! זו עלולה להיות מתקפה, מוטב שאעיף את הפקטה הזו". וכך יצא שרעיון יפה, שהיה יכול להאיץ את האינטרנט, בוטל.

לסיכום, אין קשר ישיר בין TFO ל-QUIC. ה-TFO הוא דוגמה לכך שמי שרוצה לפתור בעיות כגון TCP Head of Line Blocking עלול להתקשות מאוד לעשות זאת באמצעות שינויים והגדרות של TCP.

הורדת כמות ה-RTT

הורדת כמות ה-RTT היתה ונשארה שאיפה שמלווה אותנו בכל תהליך השיפור של הפרוטוקולים. המטרה היא תמיד להקטין את כמות ההלוך-ושוב שהלקוח עושה עם השרת עד שהלקוח יכול לשלוח לשרת את ה-HTTP GET הנכסף, או כל מידע אחר של שכבת האפליקציה.

כפי שראינו, גרסת TLS 1.3 עשתה עבודה יפה בהורדה של RTT מגרסה 1.2. כאשר מבצעים Handshake רגיל מחכים 1-RTT, ואילו עבור Session Resumption מחכים 0-RTT, כפי שמראה האיור הבא מתוך הפרק על TLS 1.3:



TLS 1.3 Handshake ולאחריו Session Resumption

אלא שבאיור יש פרט קטנטן שחסר בו... כל עוד TLS 1.3 עובד מעל TCP, נוסף גם ה-1-RTT של ה-TCP Handshake. היינו רוצים לבטל אותו, ואם אי אפשר לגמרי אז לפחות עבור Session Resumption. אין ספק ש-TFO היה יכול להשתלב פה בצורה נהדרת, והיה אפשר להשיג 0-RTT אמיתי, כאשר פקטת ה-TCP Syn ששולח הלקוח כבר מכילה גם את ה-Client Hello וגם את ה-HTTP GET.

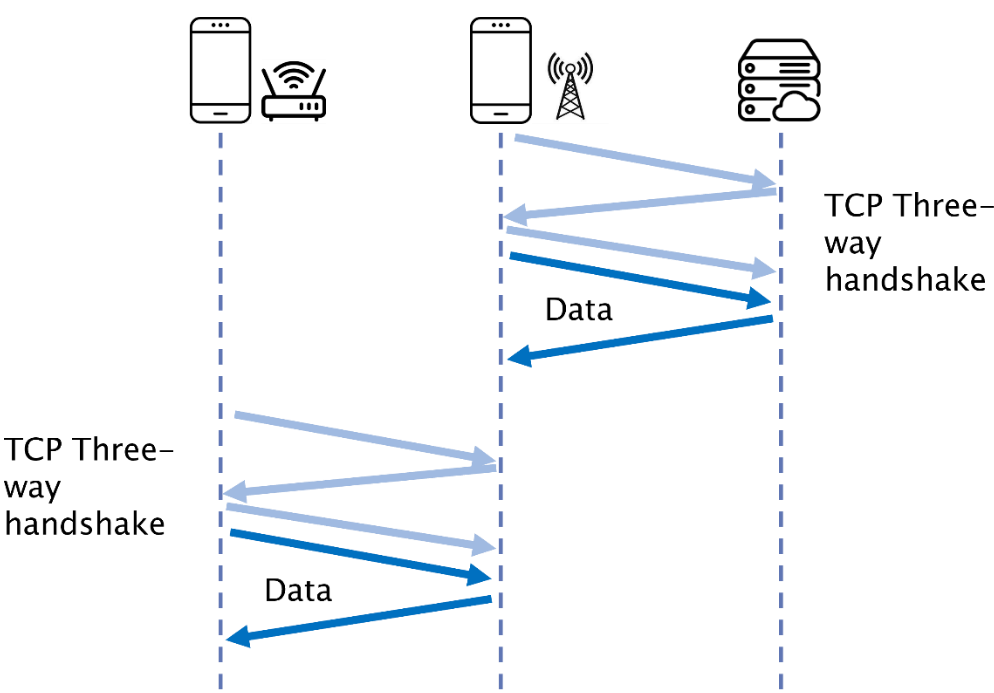
אבל TFO אינו עובד. כל עוד אנחנו מחוברים ל-TCP, אנחנו צריכים לשלם את המחיר של ה-Three Way Handshake.

Connection Migration

אתם גולשים בסמארטפון תוך כדי נסיעה (לא נהיגה, כמובן). הגעתם הביתה. הסמארטפון שלכם מזהה את הרשת הביתית ומתחבר אליה. מה קרה לכתובת ה-IP שלכם?

השתנתה כמובן. כל עוד הייתם ברשת הסלולרית, כתובת ה-IP שלכם הוקצתה על-ידי הרשת הסלולרית. כעת, היא מוקצית על-ידי שרת ה-DHCP הביתי, שהוא גם הראוטר שלכם.

סוקט מורכב, כזכור, מארבעה מזהים, שאחד מהם הוא כתובת ה-IP של הלקוח. שיניתם כתובת IP – טאק, כל קישורי ה-TCP שלכם התנתקו וצריך להקים אותם מחדש. נצטרך תוך כדי הגלישה לבצע Three Way Handshake חדש, וצפוי גם שהדבר יגרום לכך שחלק מהמשאבים שנשלחו אלינו לא יתקבלו ויתדרשו הורדה מחדש.



כלומר, TCP אינו יודע לזהות Connection Migration ולאפשר מעבר חלק. אין ללקוח אפשרות לומר לשרת "היי, זה עדיין אני, בוא נמשיך מאיפה שהיינו".

מבוא ועקרונות QUIC

פרוטוקול QUIC, קיצור של Quick UDP Internet Connection. הגרסה הראשונית שלו פותחה על-ידי גוגל ב-2012. גוגל רצו ליצור פרוטוקול שיאיץ את הגלישה לשירותים שלהם, לדוגמה יוטיוב.

כאשר חברה מסחרית מפתחת פרוטוקול, היא יכולה לשמור אותו לעצמה או לנסות להפוך אותו לתקן. אם החברה בוחרת לשמור אותו לעצמה היא מרוויחה יתרון תחרותי, כי אף אחד לא יכול להשתמש בו חוץ ממנה. במקרה של גוגל, הכסף שהיא מרוויחה מגיע לא מהדפדפן אלא מפרסומות, בין היתר ביוטיוב. לכן לגוגל אין רווח כלכלי מיוחד מכך שרק הדפדפן שלה יתמוך בפרוטוקול החדש שהם המציאו. לעומת זאת, אם יצרני דפדפנים נוספים ישתמשו בפרוטוקול שלהם, אז הגלישה ליוטיוב תהיה יותר מהירה – מה שיעלה את כמות הצופים ביוטיוב וגם יפחית עלויות של שרתים. בשנת 2016, גוגל מחליטים לשתף את ה-IETF, צוות המשימה של האינטרנט, ברעיונות שלהם לגבי הפרוטוקול החדש, כדי שיהפכו אותו לתקן. תקן פירושו שיש מסמך בשם RFC (קיצור של Request For Comments) שמתוארים בו כל הפרטים הקשורים לפרוטוקול.

בשנת 2021, ה-IETF מסיים את גיבוש התקן, ומתוך הכרה בחשיבות ובחדשנות שלו מעניק ל-RFC של QUIC את המספר העגול 9000. התקן מגדיר את פורט 443 בתור הפורט של QUIC.

נסקור בקצרה את השינויים המרכזיים שמציג QUIC:

התקשורת עוברת מעל UDP. המשמעות היא ש-QUIC כבר אינו יכול להתבסס על האמינות שמספק TCP, אלא צריך לדאוג בעצמו לכך שהתקשורת תהיה אמינה. את הרעיון הבסיסי של יצירת אמינות QUIC לקח מ-TCP ולכן נזהה אותו מיד. זרם המידע שנשלח מחולק לחלקים ממוספרים. הצד המקבל מרכיב אותם יחד לפי הסדר, והצד השולח מצפה לאישורי קבלה.

TLS 1.3 משולב בתוך QUIC. מה המשמעות של "משולב בתוך"? תאורטית היה אפשרי ש-QUIC יבנה שכבה של אמינות מעל UDP, ומעל השכבה הזו יעבור TLS 1.3 או כל פרוטוקול אבטחה אחר. אבל לא כך. מה שמיד נראה בהסנפה, הוא ש-QUIC מעביר ממש רשומות של TLS 1.3. נזהה את הרשומות המוכרות לנו מתהליך ה-Handshake. כלומר, פרוטוקול שנמצא מעל QUIC מקבל ממנו סוקט שהוא גם אמין וגם מאובטח.

הצפנה של ה-Header. ב-TLS 1.3, למרות ההצפנה, היו דברים מסויימים שאפשר היה לקרוא בגלוי. לדוגמה, את ה-TCP Header. מה אפשר לעשות עם מספרי SEQ ו-ACK? הרי מידע אינו עובר שם. ובכן – יש מה לעשות. אפשר להשתמש בהם כדי לסדר לפי הסדר את זרם המידע המוצפן. הסידור אמנם אינו מפענח את ההצפנה, אבל כן עוזר בתור שלב ראשון והכרחי לשבירת המפתח. פעולה נוספת שאפשר לבצע היא מעקב אחרי מעבר של תקשורת מערוץ פיזי אחד לערוץ פיזי אחר. למרות המעבר, צפוי שה-SEQ וה-ACK ימשיכו מהמקום שבו הם היו. מחקרים שונים מצאו גם שאפשר לזהות תעבורה של שרתים מסויימים, לדוגמה נטפליקס, לפי מאפיינים של TCP.

גרסה חדשה של HTTP. גרסת HTTP/2 היתה צריכה לטפל בדברים שלא היו קיימים בשכבות שמתחתיה, אבל QUIC כן עושה. הנה שלוש דוגמאות:

- ב-HTTP/2 יש כפי שראינו Stream ID. אך QUIC כבר כולל Stream ID.
- למדנו על HPACK, דחיסה של ה-Header שקיימת ב-HTTP/2. ב-QUIC יש דחיסה משלו, QPACK.
- HTTP/2 ביצע לפעמים PINGים ברמת שכבת האפליקציה, בתור Keep Alive, במקום המנגנון הלא תמיד עובד של TCP. גם זה משהו ש-QUIC כבר מבצע.

מסיבות אלו וסיבות נוספות, מעל QUIC עוברת גרסה חדשה, HTTP/3.

מעבר מ-TCP ל-QUIC

פרוטוקול QUIC עובד מעל פורט 443 של UDP. הבעיה הראשונה של לקוחות היא לדעת – אילו שרתים תומכים ב-QUIC?

ללקוח יש שתי אפשרויות. האחת, לזכור ששרת מסויים תומך ב-QUIC. אם כבר ביצענו התקשרות QUIC מול שרת, בפעם הבאה ננסה אותו שוב ב-UDP 443.

האפשרות השנייה היא להקים התקשרות TCP רגילה עם TLS מגרסה 1.2 או 1.3, ולבחון את שדה ה-Alt Svc, קיצור של Alternative Service. שדה זה הוא אחת מהאופציות של פרוטוקול HTTP/2. שרת שתומך ב-QUIC יודיע שם על תמיכה ב-"h3", קיצור של HTTP/3, כמו בדוגמה הבאה:

```
⌄ Header: alt-svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000
    Name Length: 7
    Name: alt-svc
    Value Length: 46
    Value: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000
```

QUIC Header

הורידו את קובץ ההסנפה ואת קובץ המפתחות שבלינק הבא:

https://data.cyber.org.il/networks/QUICfiles.zip

בשלב ראשון העלו את קובץ ההסנפה **ללא** המפתחות.

פקטה 3180 היא נקודת ההתחלה שלנו. הלקוח פונה לשרת שתומך ב-QUIC ומתחיל בהקמת קשר.
נקליק קליק ימני על הפקטה ונבחר UDP Stream → Follow.

נפתח את הפקטה. אנחנו מבחינים בגישה לפורט 443 מעל UDP, ומעל זה QUIC IETF. כזכור, IETF הוא צוות המשימה שהפך את פרוטוקול QUIC לתקן. כלומר, Wireshark מזהה שזוהי הגרסה שתוקננה ולא גרסה שונה, כגון ה-QUIC של גוגל.

```
> Frame 3180: Packet, 1292 bytes on wire (10336 bits), 1292 bytes captu
> Ethernet II, Src: HonHaiPrecis_25:11:f9 (9c:30:5b:25:11:f9), Dst: Tec
> Internet Protocol Version 4, Src: 192.168.1.103, Dst: 142.250.75.110
> User Datagram Protocol, Src Port: 62487, Dst Port: 443
> QUIC IETF
```

נפתח את QUIC:

```
v QUIC IETF
  > QUIC Connection information
    [Packet Length: 1250]
    1... .... = Header Form: Long Header (1)
    .1.. .... = Fixed Bit: True
    ..00 .... = Packet Type: Initial (0)
    [.... 00.. = Reserved: 0]
    [.... ..00 = Packet Number Length: 1 bytes (0)]
    Version: 1 (0x00000001)
    Destination Connection ID Length: 8
    Destination Connection ID: a288782ad708fa67
    Source Connection ID Length: 0
    Token Length: 0
    Length: 1232
    [Packet Number: 1]
    Payload […]: dc700a244edace6c1aff272ec2c1c013b70f43b96092
```

זהו ה-Header של QUIC. נעבור על השדות המעניינים:

- סוג ה-Header: Long Header, שמשמש בהקמת קשר. לאחר מכן יש שימוש ב-Short Header חסכוני יותר בבתים.
- סוג הפקטה: לפנינו סוג Initial, שמשמש לחלק הלא מוצפן בתהליך ה-Handshake. כזכור ב- TLS 1.3 יש מעבר למצב מוצפן מיד לאחר קביעת הסוד המשותף. ב-QUIC, הפקטות הלא מוצפנות הן מסוג Initial ואילו המוצפנות נקראות Handshake. מיד נראה אותן.
- Destination Connection ID (בקיצור DCID): השדה המעניין ביותר ב-Header. התקשורת מתבצעת מעל UDP ולכן צריך מזהה כלשהו של הקישור, שיאפשר לקשר בין פקטות של אותו קישור. המזהה

נקבע אקראית על-ידי הלקוח, אך כאשר השרת יחזיר תשובה הוא יבחר במזהה אחר והלקוח ימשיך עם המזהה שהשרת בחר. גם את זה נראה מיד.

- שדה אורך של ה-DCID. כפי שרואים, שדה האורך הוא 8 ואכן ה-DCID כולל שמונה בתים (שש עשרה ספרות הקסדצימליות).

QUIC Frames

נעבור למידע עצמו, שם נמצאים הדברים המעניינים. הדבר הראשון ששמים לב אליו הוא שלמרות שמדובר עדיין במידע לא מוצפן, הוא עבר תהליך ששינה אותו. זו אינה הצפנה, אלא ערבול של הבתים באופן שמוגדר בתקן. לכן Wireshark יכול לשחזר את המידע גם בלי מפתחות הצפנה. אם נבחר בתצוגה באפשרות Decrypted QUIC נראה את הבתים של המידע הלא מעורבל.

```
0000  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00
Packet (1292 bytes)   Decrypted QUIC (1215 bytes)
```

מה יש במידע עצמו?

המידע מחולק לחלקים שנקראים פריימים – Frames. יש פריימים מסוגים שונים, כאשר בפקטה הנוכחית יש שלושה סוגים:

- PING: זוהי המקבילה ל-HTTP PING. בית בודד שערכו 0x01 ומטרתו היא פשוט לשמור על Keep Alive עם השרת.
- PADDING: ריפוד באפסים, כדי שהפקטה תגיע לאורך מסוים.
- CRYPTO: זהו הפריים המעניין אותנו. פריימים של קריפטו נושאים את ה-Handshake של TLS.

נתבונן בפריים הקריפטו הראשון:

```
⌄ CRYPTO
    Frame Type: CRYPTO (0x0000000000000006)
    Offset: 976
    Length: 827
    Crypto Data
    [Reassembled PDU in frame: 3181]
```

לאחר השדה של סוג הפריים, מופיעים ההיסט והאורך. מה זה אומר?

כדי להבין זאת, נבצע תרגיל מחקר קטן. נעבור (עשו זאת!) על פקטה 3180 ועל פקטה 3181 ונחפש את כל הפריימים מסוג קריפטו. ניצור רשימה של כל ההיסטים וכל האורכים שיש בהם, ונחשב גם את הסכום של ההיסט + אורך.

נמיין את השורות על פי הערך בטור ההיסט, מהקטן לגדול:

Offset	Length	Total
0	70	70
70	1	71
71	1	72
72	11	83
83	125	208
208	34	242
242	7	249
249	383	632
632	221	853
853	33	886
886	17	903
903	42	945
945	5	950
950	26	976
976	827	1803

אפשר לראות שכל פריים מתחיל בהיסט שהפריים שלפניו מסתיים בו. המסקנה היא ש-QUIC מחלק את המידע בין פריימים שונים, אך כולל בתוכו את המידע הדרוש כדי שהצד הקולט יוכל להרכיב מחדש את הפאזל. מדוע מתבצעת החלוקה הזו? כדי להקשות על מי שמאזין לנו. כרגע, הפריימים אינם מוצפנים ולכן אפשר לעקוב אחרי שדה ה-Offset ולהרכיב את המידע בסדר הנכון, אך כאשר הפריימים יעברו למוצפן יהיה בלתי אפשרי אפילו לסדר את המידע לפי הסדר.

אנחנו גם רואים פה שימוש ראשון ל-DCID. הצד המקבל קיבל שתי פקטות UDP, שרק החיבור שלהן מחלץ את המידע. הצד המקבל ידע לקשור את הפקטות זו לזו באמצעות ה-DCID שלהן.

כדי לצפות בתוכן ה-TLS Handshake, נעבור לפקטה 3181.

```
⌄ CRYPTO
    Frame Type: CRYPTO (0x0000000000000006)
    Offset: 83
    Length: 125
    Crypto Data
  › [15 Reassembled QUIC CRYPTO Data Fragments (1803 bytes):
  ⌄ TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    ⌄ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
```

בסוגריים המרובעים, Wireshark מציין בפנינו שפריים זה הוא הרכבה של פריימים קודמים, בדיוק כפי שראינו. לאחר מכן מופיע המידע שהורכב מכל הפריימים, ואז מתברר שזו הרשומה המוכרת לנו של Client Hello.

ה-SNI – Server Name Indication – הוא www.youtube.com, וכך מתברר מהו האתר שאיתו אנחנו מקימים קשר. שימו לב שאנחנו עדיין לא במוצפן, כך שהמידע לאן אנחנו גולשים ב-QUIC נגיש לכל מי שיש לו גישה לפקטות שלנו.

רשומה זו כוללת, נוסף על השדות המוכרים לנו, שני שדות מעניינים.

שדה ALPN – Application Layer Protocol Negotiation. ניכנס ונגלה כי הפרוטוקול המוצע על-ידי הלקוח הוא h3 (כפי שציינו לפני כן, זהו שם קוד של HTTP/3).

שדה quic_transport_parameters מגדיר הגדרות שונות כגון כמות המידע המקסימלית שלקוח יכול לשלוח לשרת וכמות זרמי המידע שאפשר להקים מול השרת בו זמנית. לקוח היה רוצה כמובן לשלוח לשרת כמה שיותר מידע ולעבוד מול השרת בכמה שיותר ערוצים במקביל, אך השרת משרת לקוחות רבים ואינו רוצה שלקוח יחיד "יתעלק" עליו ולא ישאיר לו משאבים ללקוחות אחרים.

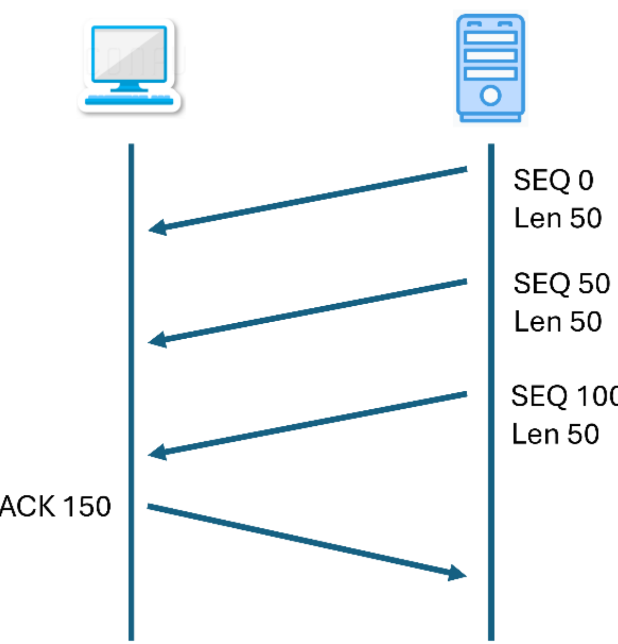
אנחנו עוברים לפקטה הבאה בזרם, פקטה 3216. השרת עונה ללקוח.

הפריים הוא מסוג חדש – ACK. לפנינו מנגנון חלופי ל-ACK של TCP. לכן, לפני שנסביר על השדות השונים, נעצור לרגע כדי להסביר על ה-ACK של TCP והבעיות שיש בו ושאותן QUIC רוצה לפתור.

TCP ACK

מנגנון האמינות של TCP מבוסס על כך שלכל בית (Byte) של שכבת האפליקציה יש מספר סידורי עוקב, Sequential number, או בקיצור SEQ. הצד המקבל את הבתים מאשר – Acknowledge או בקיצור ACK – את המספר הסידורי האחרון שהוא קיבל באופן רציף. לדוגמה, אם נשלח ACK 150 לדוגמה, זה אומר שהתקבלו כל הבתים עד 149 וכעת הצד המקבל מצפה לבית מספר 150. עד כאן – מה שלמדנו בפרק על TCP. אך זה לא הסוף.

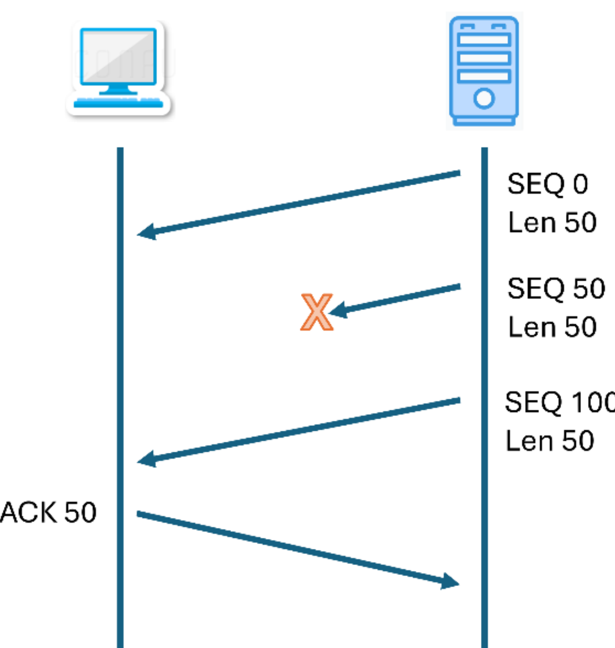
ה-ACK הוא Cumulative, או "שיתופי". נניח שהיו שלוש פקטות, כל אחת באורך של 50. הראשונה התחילה ב-SEQ 0 (והסתיימה ב-SEQ 49), השנייה התחילה ב-SEQ 50 והשלישית ב-SEQ 100. במקרה זה, ACK 150 מאשר את כל שלוש הפקטות. גם אם לא נשלחו עליהן ACKים, או שה-ACKים נפלו בדרך עקב תקלה, מי ששלח את שלושת הפקטות יכול להיות בטוח ששלושתן התקבלו.



החיסרון של ה-Cumulative ACK הוא במקרה שבו פקטה הולכת לאיבוד.

אם לצד המקבל יש "חור" בפקטות שהוא קיבל, אין לו דרך להתקדם עם ה-ACKים. נחזור לדוגמה של שלוש הפקטות שראינו קודם. נניח שהפקטה השנייה לא התקבלה. הצד המקבל יכול לשלוח ACK 50 על הפקטה הראשונה. מה בנוגע לשלישית? אם הצד המקבל ישלח ACK 150, הצד השולח יסיק מזה שגם הפקטה השנייה

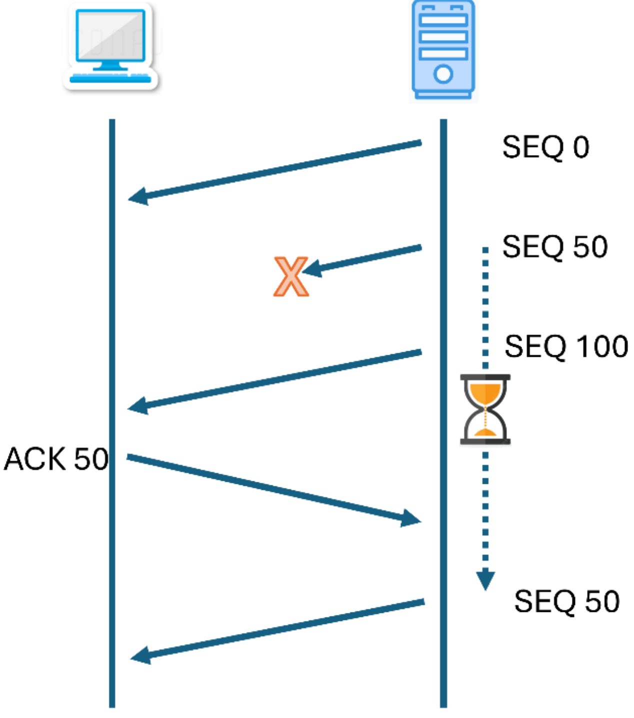
התקבלה והוא לא ישלח אותה שוב. לכן הצד המקבל "תקוע" על 50. התקיעה הזו, שנגרמת בעקבות פקטה שנפלה, היא שגורמת לתופעת ה-Head of Line Blocking שסקרנו.



במימושים ישנים של TCP, הצד הקולט היה זורק את כל הפקטות שהגיעו אחרי הפקטה החסרה, לא שולח עליהן ACK גם אם הפקטה החסרה הושלמה, ובכך מאלץ את השולח לשלוח אותן שוב גם אם הן הגיעו תקינות. כיום לא סביר שהפקטות החדשות ייזרקו, ובמקום זאת הן יישמרו בזיכרון וימתינו לכך שהפקטה החסרה תגיע. אז, יבוצע Cumulative ACK על כל הפקטות שהגיעו עד כה. ועדיין, גם במימושים אלו בעיית ה-Head of Line Blocking קיימת. בעקבות ההמתנה לפקטה החסרה, התעכבו כל הפקטות הבאות, כולל פקטות של Streamים שאינם קשורים לפקטה החסרה. TCP אינו מעביר את המידע לשכבת האפליקציה כל עוד הפקטה החסרה לא הושלמה.

נסקור מה עושה הצד השולח בתרחיש כזה. כיצד הוא יודע שפקטה נפלה בדרך?

עבור כל פקטה, הצד השולח "פותח שעון" ומחכה פרק זמן מוגדר שנקרא Timeout. אם עד ה-Timeout לא מגיע עליה ACK, הצד השולח מבצע Retrasmit, שליחה חוזרת. האיור הבא ממחיש Retransmit של פקטה שלא התקבל עליה ACK לאחר Timeout:



מה שמעלה את השאלה – כמה זמן ממתינים ל-ACK? כיצד קובעים Timeout "טוב"?

נחשוב על המקרים השונים האפשריים. אם קבענו זמן המתנה ארוך מדי, אז נבזבז זמן בהמתנה ל-ACK. יש פרק זמן שאם עד שחלף ה-ACK לא הגיע, הוא כבר לא יגיע. מצד שני, אם קבענו זמן המתנה קצר מדי, מה שנקרא Premature Timeout, אז סביר שנכריז על Retransmit גם על חבילות שהגיעו ליעד באופן תקין. ה-ACK של חבילות אלו בדרך ופשוט טרם התקבל. גם זו בעיה, כי השולח יבזבז משאבים על שליחות חוזרות, במקום להתקדם עם שליחה של פקטות חדשות.

הפתרון הוא שהשולח מנסה להעריך את ה-RTT בינו לבין המקבל, ולקבוע זמן המתנה שהוא ארוך במקצת מה-RTT. לדוגמה, אם ה-RTT בין הצדדים הוא שנייה, אז לא הגיוני לחכות ל-ACK עשר שניות וגם לא הגיוני לחכות חצי שנייה. זמן המתנה של שנייה ועוד תוספת קטנה, כגון עשירית שנייה, הוא זמן מוצלח. התוספת נדרשת גם כיוון שלעיתים פקטות מתעכבות קצת בדרך (נתקלנו בכך כאשר בתחילת לימודי הרשתות ביצענו פינג לשרת כלשהו, וראינו שהזמן שחולף עד לתשובה יכול להיות שונה בין פקטה לפקטה) וגם כיוון שמסובך להעריך בצורה מדוייקת את ה-RTT בין הצדדים.

איך הצד השולח יכול להעריך את ה-RTT בינו לבין הצד המקבל בצורה טובה ככל האפשר?

השיטה מבוססת על מדידת הזמן שלקח להודעות קודמות שלו לקבל ACK.

בתחילת התקשורת, השולח יקבע זמן Timeout ארוך יחסית, או שייקח אומדן ראשוני מפרק הזמן שלקח לעשות Three Way Handshake. ככל שמתקבלים יותר ACKים כך השולח יכול לדייק את ההערכה שלו על ה-RTT וכך לשפר את קצב התקשורת בין הצדדים.

אך ההערכה של ה-RTT אינה פשוטה. ב-TCP יש מספר דברים שמקשים על החישוב. נסקור את הבעיות, וחלק זה חשוב במיוחד להבנה מכיוון שכאשר נסקור את מנגנון ה-ACK של QUIC נראה כיצד הוא נבנה כדי להתמודד עם בעיות אלו.

בעיה ראשונה: **ה-Cumulative ACK "מותח" את זמן ה-ACK של חלק מהפקטות**. כאשר סקרנו את ה-Cumulative ACK, נתנו דוגמה שבה ACK יחיד מאשר שלוש פקטות. הפקטה בעלת SEQ 0 קיבלה את ה-ACK רק לאחר ששתי פקטות נוספות הגיעו לצד המקבל בהצלחה. הזמן שלקח האישור שלה מוטה כלפי מעלה ולא משקף את ה-RTT.

בעיה שניה: **לצד הקולט לוקח זמן מרגע שהוא מקבל את הפקטה ועד שהוא שולח את ה-ACK**. הזמן שלוקח לצד השולח לקבל את ה-ACK על פקטה מורכב מחיבור של ה-RTT עם זמן נעלם, שהוא משך הזמן שלוקח לצד הקולט לשלוח את ה-ACK. לדוגמה, שלחנו פקטה וקיבלנו ACK אחרי 5 מילישניות. יכול להיות שה-RTT הוא 5 מילישניות, אבל אם לצד השני לקח 2 מילישניות לענות לנו, אז ה-RTT הוא בעצם רק 3 מילישניות. למה יש שיהוי בצד המקבל? קודם כל, כל חישוב לוקח זמן כלשהו ויכולים להיות עיכובים בגלל עומס. אבל יכול גם להיות שהצד המקבל משתהה בכוונה בשליחת ה-ACK. הצד המקבל יכול לקוות שאם הוא יחכה עוד קצת, תתקבל עוד פקטה מהשולח ואז פקטת ה-ACK שלו כבר תאשר מספר פקטות יחד, מה שיחסוך לו שליחה של פקטה. אפשרות נוספת שעומדת בפני הצד המקבל היא להעלות את שדה ה-ACK בפקטת המידע הבאה שהוא ישלח, וכך לחסוך שליחה של פקטת ACK נפרדת. הצד המקבל יכול לחשוב "יש לי פקטה שאני עומד לשלוח תיכף, אז במקום לשלוח עכשיו פקטת ACK, שווה להמתין רגע".

בעיה שלישית: **ב-ACK על Retrasmit, אי אפשר לדעת בביטחון אם ה-ACK הוא על הפקטה המקורית או על הפקטה החוזרת**. השדות השונים של TCP יהיו זהים ולצד המקבל אין דרך לדעת על איזו פקטה התקבל ה-ACK. נכון שרוב הסיכויים הם שה-ACK הוא על הפקטה החוזרת, אבל תמיד יכולות להיות הפתעות. כמובן שלהחלטה על איזו פקטה התקבל ה-ACK יש השפעה גדולה על הסקת ה-RTT.

נסכם את הלקחים עבור מי שמתכננים פרוטוקול חדש:

- מועיל לשמר את היכולת לאשר מספר פקטות בבת אחת. זה מקטין את העומס שיוצרים ה-ACKים וחוסך שידורים חוזרים של פקטות שה-ACK שלהן נפל.

- מצד שני, צריך לתקן את המצב ש"חור" עוצר את העיבוד של הפקטות הבאות, גם אם הן שייכות למשאב אחר, שכל הפקטות שלו הגיעו תקין.
- כדאי לעזור לצדדים לגלות איפה יש "חורים" בקבלת הפקטות ולהשלים רק אותם.
- כדי לשפר את האומדן של ה-RTT, כדאי להעביר מידע על כמה זמן ה-ACK השתהה. כלומר, כמה זמן עבר מרגע שהפקטה התקבלה ועד שיצא עליה ACK.
- במקרה של Retransmit, צריך למצוא דרך להבדיל בין ACK על הפקטה המקורית לבין ACK על השידור החוזר שלה.

כעת כשהבנו את המטרות הללו, אפשר להתקדם ולבחון איך נראית פקטת ACK של QUIC.

QUIC ACK

חזרנו מסקירת ה-TCP אל עולם ה-QUIC.

הדבר הראשון שנבחין בו ב-QUIC הוא שאין מספרי SEQ. במקום זאת, לכל פקטה יש מספר סידורי. המספר אינו נשלח בצורה גלויה, אך Wireshark יודע לחלץ אותו ולהציג אותו. בשונה מ-TCP, שבו ה-ACKים תואמים לכמות הבתים שהתקבלו, ב-QUIC ה-ACKים תואמים למספרי הפקטות.

הבדל נוסף מ-TCP הוא שמספרי פקטות אינם חוזרים על עצמם כאשר משדרים מחדש פקטה שנפלה בדרך. זהו תיקון של המנגנון הבעייתי שסקרנו ב-TCP, שבו פקטה שנשלחה מחדש נראית זהה לפקטה המקורית, מה שמקשה על הצד שקיבל עליה ACK לדעת אם הוא התקבל על השליחה המחודשת או על המקור. אנחנו כבר רואים שהשינוי הזה מתקן את אחד הלקחים מ-TCP.

פריים מסוג ACK מעביר את הדברים הבאים:

- כמה זמן הצד המקבל השתהה בין קבלת הפקטה ועד ששלח את ה-ACK. המידע הזה מתקן לקח נוסף מ-TCP, ומאפשר חישוב מדויק יותר של ה-RTT.
- לאילו פקטות יש אישור קבלה
- אילו "חורים" יש, כלומר אילו מספרי פקטות חסרים בצד המקבל

לדוגמה, נניח שלצד המקבל הגיעו פקטות 1,2,3,4,7,8. פקטות 5,6 חסרות. ה-ACK של QUIC ידווח הן על הפקטות שהגיעו והן על אלו שלא הגיעו. הדיווח יתבצע באמצעות שדות שיענו על השאלות הבאות:

- מהי הפקטה בעלת המספר הגבוה ביותר שהתקבלה?

בדוגמה שלנו, התשובה לשאלה זו היא 8.

- כמה פקטות התקבלו לפניה, בלי "חורים"?

התשובה לשאלה זו – 1. זאת מכיוון שישנה פקטה אחת, פקטה מספר 7, שהתקבלה לפני פקטה 8 בצורה צמודה, בלי "חור" באמצע.

- כמה אזורים של חורים ישנם?

כעת יש שתי אפשרויות. או שיש "חור", או שאין. אם אין חור, הכמות תהיה 0. אם יש אזורים של חורים, המספר שלהם ידווח פה. שימו לב שחורים צמודים מדווחים בתור אזור אחד. בדוגמה שלנו, חסרות פקטות 5,6. הן צמודות ולכן הערך בדיווח יהיה "1".

- עבור כל אזור של חורים:
  - כמה פקטות נמצאות בחור?
  - כמה פקטות התקבלו תקינות לפני החור?

הדרך שבה המספרים הללו מועברים היא קצת טריקית להבנה, אבל הדוגמה שלנו תבהיר אותה.

המידע שקיבלנו עד כה מה-ACK מוסר לנו שפקטות 7,8 התקבלו תקין ושיש חור אחד. המסקנה היא שפקטה 6 לא התקבלה. כדי למסור שגם פקטה 5 לא התקבלה, הדיווח יהיה "1". כלומר, חסרה פקטה אחת נוסף על הפקטה שבסוף החור.

המסקנה הבאה חייבת להיות שפקטה 4 כן התקבלה תקין. כדי למסור שגם שלושת הפקטות לפניה התקבלו תקין, ידווח "3".

כלומר בדוגמה שלנו הדיווח יהיה כך:

פקטה אחרונה שהתקבלה – 8

כמות פקטות צמודות לפניה – 1

כמות רווחים – 1

כמות פקטות צמודות לסוף הרווח – 1

כמות פקטות צמודות לסוף האזור הבא שאינו חור – 3

נבחן את פקטה 3229.

```
ACK
    Frame Type: ACK (0x0000000000000002)
    Largest Acknowledged: 4
    ACK Delay: 62
    ACK Range Count: 0
    First ACK Range: 3
```

שדה ה-Largest Acknowledged מציין "4", כלומר, זו הפקטה עם המספר הגבוה ביותר שהתקבל.

ה-First ACK Range הוא 3, כלומר התקבלו 3 פקטות צמודות לפקטה שמספרה הוא 4. כלומר, מאושרות הפקטות 1,2,3,4.

ה-ACK Range Count הוא 0, מכיוון שאין חורים עד כה.

ה-ACK Delay הוא 62. מספר זה מאפשר לחשב את כמות המיקרו שניות (מיליוניות השנייה) שהמקבל התעכב בשליחה.

בזרם המידע שלפנינו יש גם חורים. אך כדי לראות דיווח ACK עם חורים נצטרך להוסיף ל-Wireshark את קובץ המפתחות שלנו. נבצע זאת ונבחן את פקטה 3401:

```
ACK
    Frame Type: ACK (0x0000000000000002)
    Largest Acknowledged: 12
    ACK Delay: 1
    ACK Range Count: 1
    First ACK Range: 2
    Gap: 0
    ACK Range: 3
```

התקבלה פקטה מקסימלית 12.

ה-First ACK Range הוא 2, כלומר התקבלו שתי פקטות לפני פקטה 12. מכאן שפקטות 10,11,12 התקבלו.

ה-ACK Range Count מדווח על חור אחד. כלומר, פקטה 9 היא חור.

ה-Gap הוא 0, כלומר יש אפס פקטות לפני פקטה 9 שהן חור. כלומר פקטה 8 התקבלה.

ה-ACK Range הוא 3, כלומר יש 3 פקטות שהתקבלו לפני פקטה 8. מכאן שפקטות 5,6,7,8 התקבלו.

ומה בנוגע לפקטות שקודמות לפקטה 5? הן כבר קיבלו ACK, אין צורך לחזור על כך, ויותר מזה – QUIC אינו מאפשר לדווח בתור "חור" על פקטה שכבר דווח עליה ACK.

נסכם את מה שראינו ונשווה לדברים שרצינו לשפר ב-TCP:

1. **לקח מ-TCP**: מועיל לשמר את היכולת לאשר בבת אחת מספר פקטות. זה מקטין את העומס שיוצרים ה-ACKים וחוסך שידורים חוזרים של פקטות שה-ACK שלהן נפל. **פתרון של QUIC**: שימרנו את היכולת לאשר כמה פקטות בו זמנית.

2. **לקח מ-TCP**: מצד שני, צריך לתקן את הבעיה ש"חור" עוצר את העיבוד של הפקטות הבאות, גם אם הן שייכות למשאב אחר, שכל הפקטות שלו הגיעו תקין. **פתרון של QUIC**: טרם ראינו זאת בהסנפה, אך לכל משאב שנשלח יש Stream ID, בדומה למה שראינו ב-HTTP/2. באמצעות שימוש ב-Stream ID, QUIC יכול להעלות לשכבת האפליקציה פקטות ששייכות לאותו משאב. כך, גם במקרה של "חור" בקבלה, משאב שהפקטות שלו אינן שייכות ל"חור" יכול להמשיך עיבוד בלי שיהוי.

3. **לקח מ-TCP**: כדאי לעזור לצדדים לגלות איפה יש "חורים" **בקבלת** הפקטות ולהשלים רק אותם. **פתרון של QUIC**: ה-ACK כולל גם מידע לגבי חורים.

4. **לקח מ-TCP**: כדי לשפר את האומדן של ה-RTT, כדאי להעביר מידע כמה זמן ה-ACK השתהה. כלומר כמה זמן עבר מרגע שהפקטה התקבלה ועד שיצא עליה ACK. **פתרון של QUIC**: זמן השיהוי בשליחת ה-ACK נשלח כחלק מה-ACK.

5. **לקח מ-TCP**: במקרה של Retransmit, צריך למצוא דרך להבדיל בין ACK על הפקטה המקורית לבין ACK על השידור החוזר שלה. **פתרון של QUIC**: כל פקטה מקבלת מספר סידורי משלה, המספר אינו חוזר על עצמו גם אם יש שידור חוזר. נשאלת אם ככה השאלה, אם מספר הפקטה שונה מהמספר המקורי, איך הצד המקבל יודע שמדובר ב-Retransmit? ובכן, השיוך מתבצע לפי מאפיינים אחרים שהזכרנו. לכל משאב יש Stream ID וכל משאב מחולק לחלקים, שנשלחים עם שדה שאומר מה ההיסט שלהם מתחילת המשאב. לכן מי שמקבל את הצירוף של היסט יחד עם Stream ID, יכול לדעת שזה מילוי של "חור".

תהליך ה-Handshake והמעבר למוצפן

כפי שרואים בהסנפה, לאחר פקטות מסוג Initial מופיעות פקטות מסוג Handshake. ההבדל ביניהן הוא המעבר למוצפן. פקטת ה-Server Hello כוללת כזכור גם את החלק של השרת ביצירת הסוד המשותף. לאחר פקטה זו, הלקוח יכול כבר להצפין את התקשורת ומתחיל תהליך ה-Handshake המוצפן (אם אינכם רואים אותו, וודאו שהזנתם את קובץ המפתחות).

```
QUIC IETF
  QUIC Connection information
  [Packet Length: 1250]
  1... .... = Header Form: Long Header (1)
  .1.. .... = Fixed Bit: True
  ..10 .... = Packet Type: Handshake (2)
  [.... 00.. = Reserved: 0]
  [.... ..00 = Packet Number Length: 1 bytes (0)]
  Version: 1 (0x00000001)
  Destination Connection ID Length: 0
  Source Connection ID Length: 8
  Source Connection ID: e288782ad708fa67
```

ה-Handshake המוצפן מתחיל לאחר ה-Server Hello, פקטה מספר 3219, ומסתיים עם Client Finished, שמתקיים בפקטה 3402.

כעת, אפשר להבין יותר טוב את הסיפור של פקטה מספר 9, אותה פקטה שראינו שהלקוח לא אישר אותה עם ACK. נבדוק מה השרת שלח. פקטה מספר 3374 כוללת שתי פקטות של QUIC בתוכה, פקטת QUIC מספר 8 ופקטת QUIC מספר 9. אנחנו כבר לומדים מזה משהו – פקטת UDP יכולה לכלול יותר מפקטת QUIC אחת.

אם פקטת QUIC מספר 9 הגיעה, אז מדוע הלקוח שלח עליה "חור" במקום לאשר אותה? נשים לב לכך שהשרת שלח שתי פקטות HTTP/3 עוד לפני שה-Handshake הסתיים.

Source	Destination	Protocol	Info
192.168.1.103	142.250.75.110	QUIC	Initial, DCID=e288782ad708fa67, PKN: 3, ACK, PADDING
192.168.1.103	142.250.75.110	QUIC	Handshake, DCID=e288782ad708fa67, PKN: 5, PADDING, PING
192.168.1.103	142.250.75.110	QUIC	Handshake, DCID=e288782ad708fa67, PKN: 7, PADDING, PING
142.250.75.110	192.168.1.103	QUIC	Handshake, SCID=e288782ad708fa67, PKN: 5, CRYPTO
142.250.75.110	192.168.1.103	QUIC	Handshake, SCID=e288782ad708fa67, PKN: 6, CRYPTO
142.250.75.110	192.168.1.103	QUIC	Handshake, SCID=e288782ad708fa67, PKN: 7, CRYPTO
142.250.75.110	192.168.1.103	HTTP3	Protected Payload (KP0), PKN: 9, STREAM(3), SETTINGS
142.250.75.110	192.168.1.103	QUIC	Handshake, SCID=e288782ad708fa67, PKN: 10, ACK
192.168.1.103	142.250.75.110	QUIC	Handshake, DCID=e288782ad708fa67, PKN: 8, ACK
192.168.1.103	142.250.75.110	QUIC	Handshake, DCID=e288782ad708fa67, PKN: 9, ACK
142.250.75.110	192.168.1.103	QUIC	Handshake, SCID=e288782ad708fa67, PKN: 11, ACK, CRYPTO
142.250.75.110	192.168.1.103	HTTP3	Protected Payload (KP0), PKN: 13, STREAM(3), SETTINGS
192.168.1.103	142.250.75.110	QUIC	Handshake, DCID=e288782ad708fa67, PKN: 10, ACK
192.168.1.103	142.250.75.110	QUIC	Handshake, DCID=e288782ad708fa67, PKN: 11, ACK
192.168.1.103	142.250.75.110	QUIC	Handshake, DCID=e288782ad708fa67, PKN: 12, CRYPTO

השרת מקדם פקטות של שכבת האפליקציה, כך שברגע שתהליך ה-Handshake יסתיים כבר תהיה התחלה של מידע בידי הלקוח. הלקוח קיבל אותן אך אינו מעבד אותן עד שהוא מסיים את העיבוד של ה-Handshake, ולכן

בשלב זה אין ביכולתו לדווח עליהן ACK. עם זאת, הלקוח כן רוצה לדווח לשרת על קבלת יתר הפקטות של ה-Handshake. לפי התקן של QUIC הלקוח רשאי לדווח על חור ולעדכן על קבלה בהמשך.

לאחר ה-Handshake, פקטות ה-QUIC נראות אחרת. נסקור את השינויים:

```
⌄ QUIC IETF
  › QUIC Connection information
    [Packet Length: 70]
  ⌄ QUIC Short Header DCID=e288782ad708fa67 PKN=13
      0... .... = Header Form: Short Header (0)
      .1.. .... = Fixed Bit: True
      ..0. .... = Spin Bit: False
      [...0 0... = Reserved: 0]
      [.... .0.. = Key Phase Bit: False]
      [.... ..00 = Packet Number Length: 1 bytes (0)]
      Destination Connection ID: e288782ad708fa67
      [Packet Number: 13]
      Protected Payload: 1572b913a4fd3d3264e0df7de171f1f98f09620d25b346df114d9abd235cac986e2651b38d0b99
  ⌄ STREAM id=2 fin=0 off=0 len=42 dir=Unidirectional origin=Client-initiated
    › Frame Type: STREAM (0x0000000000000008)
    › Stream ID: 2
      Stream Data: 00041b018001000006800400000740643301c0000017c3013a2ca7f92a80c000000ab1ddd02e0320ac04
› Hypertext Transfer Protocol Version 3
```

ראשית, ה-Header משתנה מ-Long ל-Short. המידע היחידי שיש בו הוא ה-Destination Connection ID. חשוב להדגיש, ש**כל** מה שמופיע לאחר מכן הוא מוצפן.

שנית, סוג פריים חדש – STREAM – מתווסף לסוגים שהכרנו קודם לכן, CRYPTO, ACK, PING, PADDING.

שלישית, ל-Stream יש Stream ID. הרעיון של Stream ID מוכר לנו מ-HTTP/2. הוא מאפשר לייצר הפרדה בין משאבים שונים. ב-HTTP/2 ראינו שההפרדה בין המשאבים סובלת מבעיית ה-Head Of Line Blocking של TCP. המעבר ל-UDP מאפשר הפרדה מלאה. אם חסרות פקטות של משאב בעל Stream ID מספר 2, לדוגמה, אין לזה שום השפעה על Stream ID מספר 3. נציין לעצמנו שבעיית ה-Head Of Line Blocking נפתרה.

רביעית, HTTP/3. כבר אין צורך לבצע חלק מהדברים שביצע HTTP/2. לדוגמה, ב-HTTP/2 יש שדה של Stream ID. כאשר QUIC לוקח על עצמו את המשימה, אפשר לייצר גרסה חדשה של HTTP, גרסה 3.

היררכיה של Connection, Stream, Packet, Frame

נעשה סדר בחלקים השונים שמרכיבים את QUIC.

פקטות UDP הן הבסיס. כדי לחבר בין פקטות UDP ששייכות לאותה תקשורת בין שרת ולקוח, לכל קישור יש DCID, מזהה קישור. ה-DCID משותף הן לשרת והן ללקוח.

כל פקטת UDP יכולה להכיל פקטת QUIC אחת או יותר. לכל פקטת QUIC יש מספר, Packet Number, ייחודי.

כל פקטת QUIC מכילה פריים אחד או יותר של QUIC. פריימים יכולים להיות חלקים של מידע גדול יותר, ובמקרה כזה לכל פריים יהיה נתון ההיסט שלו מתחילת המידע ומה הגודל שלו. כך הצד המקבל יכול להרכיב בחזרה את המידע.

מידע של שכבת האפליקציה נשלח על גבי פריימים מסוג STREAM, שיש להם גם מזהה של Stream ID.

הדוגמה הבאה מראה פריים של Stream מסוים. נשים לב לכך שישנם שני מזהים שמאפשרים לצד המקבל להרכיב את המידע בצורה נכונה:

- ה-Stream ID שקובע לאיזה משאב שייך המידע בפריים
- ה-Offset, ההיסט, שקובע מה המיקום היחסי בתוך המשאב של המידע שבפריים

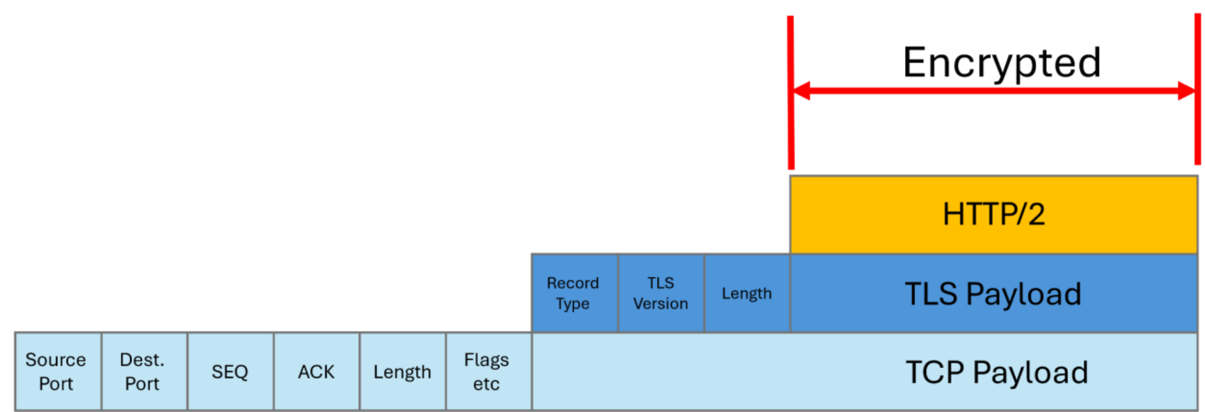
```
⌄ STREAM id=2 fin=0 off=42 len=12 dir=Unidirectional
  › Frame Type: STREAM (0x000000000000000e)
  › Stream ID: 2
    Offset: 42
    Length: 12
    Stream Data: 800f07000700753d302c2069
⌄ STREAM id=0 fin=0 off=0 len=1206 dir=Bidirectional
  › Frame Type: STREAM (0x0000000000000008)
  › Stream ID: 0
    Stream Data […]: 0149b10000d1508cf1e3c2fe8f6a6d8
```

במקרה של הפקטה שלפנינו, מדובר בפריים ששייך למשאב בעל המזהה Stream ID 2, ואשר המיקום היחסי שלו הוא 42 בתים מתחילת המשאב.

הצפנת Headerים

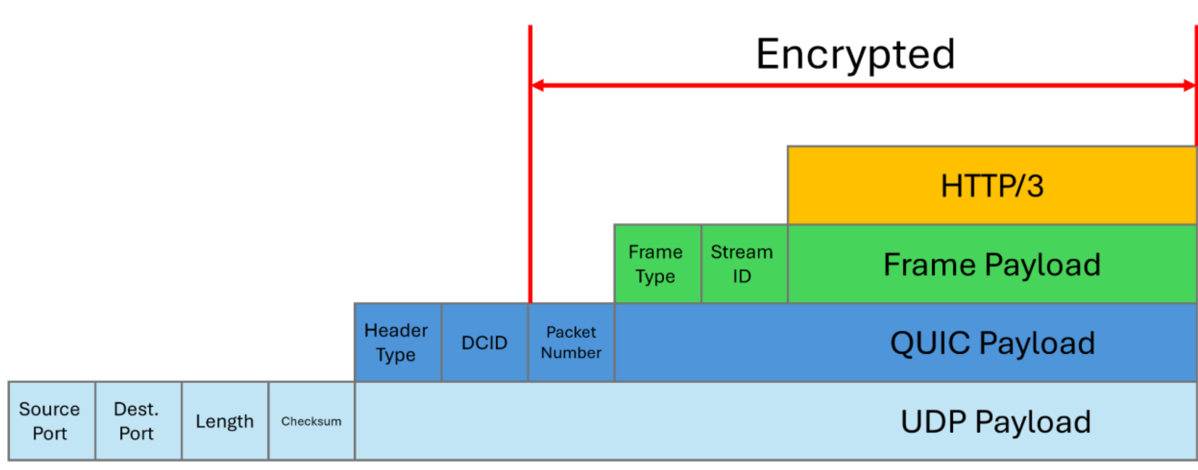
מה יכול לראות מי שמסניף QUIC בלי מפתחות ההצפנה?

האיורים הבאים ממחישים את ההבדלים בין HTTP מעל TLS מעל TCP, לבין HTTP מעל QUIC.



ב-HTTP/2, רק המידע של שכבת האפליקציה מוצפן. כלומר, כל השדות של TCP עוברים בגלוי, בין היתר ה-SEQ. דבר זה מאפשר למי שקולט את התקשורת לסדר את הפקטות לפי הסדר, גם אם אין בידיו את המפתח.

ומה לגבי HTTP/3, מעל QUIC?



אם ב-TLS מעל TCP החלק המוצפן הוא רק המידע, ב-QUIC מוצפנים גם רוב ה-Headerים. מי שתופס תעבורת QUIC בלי מפתח ההצפנה יכול לראות רק את ה-UDP Header, שכולל את מספרי הפורטים של השרת והלקוח, ואת ה-Connection ID.

כל יתר המידע שעשוי לשמש אפילו לא לפענוח אלא רק לסידור המידע המוצפן לפי הסדר הנכון (כלומר מחולק ל-Streamים שבתוכם הפריימים נמצאים לפי הסדר) – כל יתר המידע הזה מוצפן.

מה בנוגע ל-DCID? לכאורה נראה שמה שכן אפשר לעשות זה לאסוף יחד את כל הפקטות בעלות אותו ה-Connection ID. אולי בדרך כזו או אחרת זה יכול להיות שימושי לצורך מעקב. אך למעשה, פרוטוקול QUIC מצפין באופן עקיף גם את ה-DCID.

בשלב מסויים בהתקשרות, השרת שולח ללקוח פריים בשם NEW CONNECTION ID, שבו הוא מציע ללקוח מספר Connection ID נוספים לשימושו. הפריים הוא מוצפן כמובן, ולכן מי שאין לו את מפתחות ההצפנה ושומר את המידע רק לפי ה-Connection ID, יוכל לעקוב אחרי ההתקשרות רק עד הנקודה שבה הלקוח החליט לדלג ל-Connection ID החדש. המשמעות היא שלמרות שמזהה הקישור אינו מוצפן, קשה מאוד לעקוב אחרי מי שמדלג.

```
⌄ NEW_CONNECTION_ID
    Frame Type: NEW_CONNECTION_ID (0x0000000000000018)
    Sequence: 1
    Retire Prior To: 0
    Connection ID Length: 8
    Connection ID: e388782ad708fa67
    Stateless Reset Token: 9854498c25cc14fdba98651bbff77676
```

0-RTT "אמיתי"

כאשר למדנו מהו RTT, הפרדנו בין RTT של TLS לבין RTT של פרוטוקולים אחרים, ובפרט TCP. באותה נקודת זמן זה היה נראה צעד מוזר. בין כה וכה אנחנו עובדים מעל TCP, לכן איזו סיבה יש לספור את ה-RTTים בלי ה-RTT הנוסף שנדרש על-ידי TCP? כעת, כשעברנו ל-UDP, הסיבה לכך מתבררת. ה-RTT מבטא את השיהוי הייחודי לכל פרוטוקול. TLS לא חייב לעבור מעל TCP.

איך נראה 0-RTT "אמיתי"?

TLS 1.3 מאפשר, כזכור, חידוש של קישור קיים. לקוח שרוצה להתחבר מחדש לשרת, יכול לעשות זאת תוך זמן מוגבל באמצעות שימוש ב-Session Ticket.

בהתחברות המחודשת, הלקוח שולח כבר מידע של שכבת האפליקציה. אמנם ניתן להשתמש במפתח ההצפנה הישן להעביר רק כמות מוגבלת של בתים, מה שנקרא Early Data, אך במקביל השרת והלקוח עובדים על יצירת מפתחות חדשים.

כל עוד עבדנו מעל TCP היינו צריכים לשלם RTT נוסף לטובת Three Way Handshake.

פרוטוקול QUIC משתמש ב-TLS 1.3, כך שהיכולת של 0-RTT מוטמעת בתוכו מלכתחילה. קבלנו 0-RTT "אמיתי", שבו לקוח שרוצה לחדש התקשרות עם שרת יכול לשלוח מידע של שכבת האפליקציה ממש מעל הפקטה הראשונה שהוא שולח.

הפקטה תראה כך: פרוטוקול UDP, מעליו פרוטוקול QUIC שיכיל בתוכו Client Hello של TLS 1.3. בתוך ה-Client Hello יהיה ה-Session Ticket. מעל QUIC, שכבת האפליקציה. לדוגמה, בקשת GET של HTTP/3. וכל זאת בפקטה הראשונה שהלקוח שולח לשרת!

TLS over UDP

בסעיף זה נניח בצד לרגע את QUIC ונסקור בקצרה אפשרות אחרת להעברת מידע מוצפן מעל UDP. פרוטוקול DTLS, קיצור של Datagram TLS. פרוטוקול זה מוסיף ל-TLS את המינימום הנדרש כדי להעביר אותו מעל UDP. הסיבה לקיומו של DTLS בעולם שבו קיים כבר QUIC היא פשטות ומהירות. ישנן אפליקציות שבשבילן חשוב יותר שהמידע יעבור מהר מאשר שיושלמו פקטות חסרות. לדוגמה, VoIP, קיצור של Voice over IP. נרצה שהשיחה תעבור מוצפן, אבל אם פקטה אבדה השיחה כבר התקדמה ומיותר לנסות להשלים את ההברה החסרה.
לפרוטוקול זה אין פורט קבוע, במקום זה הפורט שלו תואם -או לעיתים מזכיר -את הפורט המתאים לשכבת האפליקציה שעוברת מעליו. לדוגמה קישורי VPN, שמעבירים בדרך כלל HTTPS, עובדים בפורט 443. עבור אפליקציית DNS הפורט הוא 853 (פורט 53 מעל UDP תפוס כבר מן הסתם…).

לפרוטוקול DTLS יש Retransmission רק על פקטות ששייכות ל-TLS Handshake. הצד השולח משתמש ב-Timeout, ואם לא מתקבל תשובה, הפקטה נשלחת מחדש. לעומת זאת, פקטות שנשלחות אחרי ה-Handshake כבר אינן כוללות מנגנון שידור מחדש. לכל פקטה יש מספר סידורי, גם כדי שהצד המקבל ידע להרכיב מחדש את הפקטות לפי הסדר וגם כיוון שהמספר הסידורי הוא בעל חשיבות בהצפנה. נזכור ש-TLS משתמש ב- Counter Mode כדי למנוע Reply Attacks.

נקודה מעניינת ב-DTLS היא שימוש ב-Extension שלא סקרנו עד עכשיו – Cookie. לאחר שהלקוח פונה לשרת, השרת מחזיר לו Cookie בתוך Extension, ואינו ממשיך בתהליך ה-Handshake עד שהלקוח מחזיר לו את ה-Cookie.

כדי להבין מדוע נדרש ההלוך ושוב עם ה-Cookie, שנראה מיותר במבט ראשון, ניזכר בכלי שהכרנו היטב בחלק הראשון של ספר רשתות – Scapy. כפי שראינו, אפשר לייצר פקטה שכתובת ה**שולח** שלה היא מה שנבחר, כולל כתובות לא קיימות. השרת שמקבל את כתובת ה-IP הזו מאת השולח כביכול, ישלח את התגובה לכתובת זו.

אם אנחנו עובדים עם TCP, התהליך לא יתקדם מעבר לפקטת ה-SYN ACK. השרת יענה לכתובת IP מזוייפת כלשהי, ולא יקבל בחזרה פקטת ACK (או יקבל בחזרה פקטת RST אם יש בכתובת זו מחשב, שלא ביקש לפתוח סוקט מול השרת). כלומר, אם מישהו זייף כתובת IP של שולח, הוא גרם לשרת לשלוח פקטת SYN ACK יחידה.

לעומת זאת מעל UDP, אם מישהו זייף כתובת IP של שולח, עלולה להיגרם לשרת טרחה לא קטנה. השרת צריך לשלוח Server Hello, סרטיפיקט, Server Key Exchange. כל אלו דורשים חישובים וחתימות. כלומר, במקרה של זיוף IP היחס בין כמות הטרחה בצד המזייף לבין כמות הטרחה של בצד השרת הוא גבוה מאד. השיטה הזו קורצת למי שרוצים לבצע מתקפת מניעת שירות על השרת.

ה-Cookie מונע זאת, כיוון שהשרת לא יבצע שום דבר עד שלא וידא שהפקטה מגיעה מכתובת IP שעומד מאחריה לקוח אמיתי.

DNS over QUIC

בפרק הקודם סקרנו איך עובר DNS מעל HTTP/2, הקרוי גם DoH. העקרון של DNS מעל QUIC, הקרוי גם DoQ, הוא דומה. נראה בקשה מצד הלקוח, כאשר סוג הבקשה ושם הדומיין מקודדים Base64. תגובת שרת ה-DNS תופיע בתוך QUIC, מוצפן כמובן.

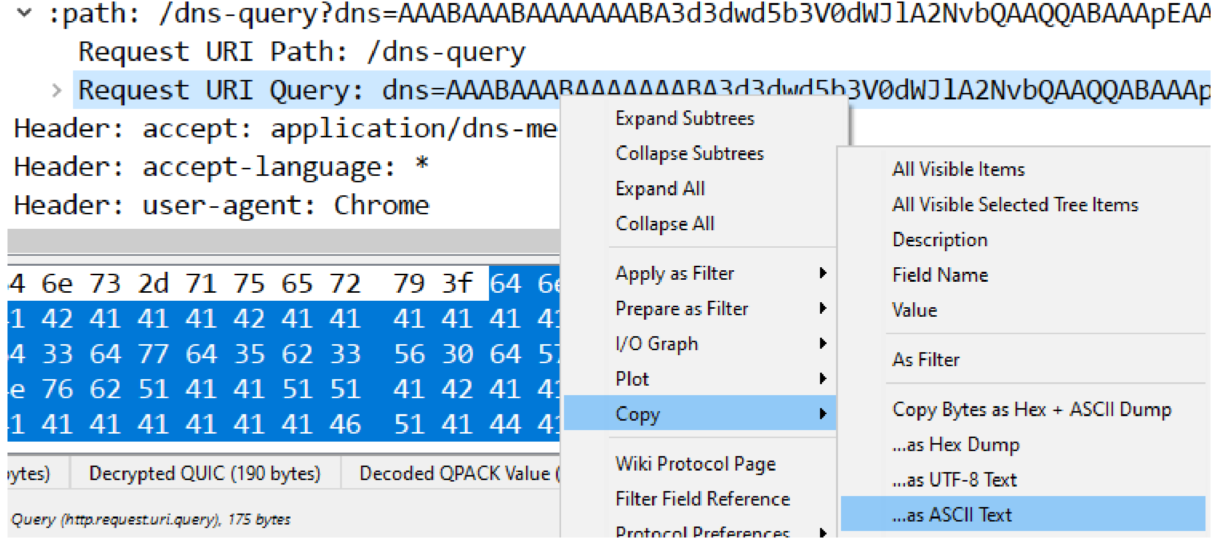
סיפורנו מתחיל בכך שהלקוח, הדפדפן, מקים קישור QUIC מול שרת ה-DNS המאובטח שהוגדר בדפדפן. לאחר הקמת הקישור, כל בקשה של הלקוח היא 0-RTT, אין צורך בביצוע Handshake משום סוג. בהסנפה שאנחנו עובדים איתה, הלקוח מתקשר מול שרת ה-DNS של גוגל, התומך בבקשות DoQ.

נפתח את פקטה 3019.

```
⌄ Hypertext Transfer Protocol Version 3
  ⌄ Request Stream
    ⌄ HEADERS len=11
        [Stream ID: 12]
        Type: HEADERS (0x0000000000000001)
        Length: 11
        Frame Payload: 0c00d18ad781de88878680
        [Decoded Headers Length: 403]
        [Headers Count: 9]
      › Header: :method: GET
      › Header: :authority: dns.google
      › Header: :scheme: https
      › Header: :path: /dns-query?dns=AAABAAABAAAAAAABA3d3dwd5b3V0dWJ1A2NvbQAAQQABAAApEAAAAAAA
      › Header: accept: application/dns-message
```

הלקוח שולח בקשת GET של HTTP/3. המשאב המבוקש מקודד Base64.

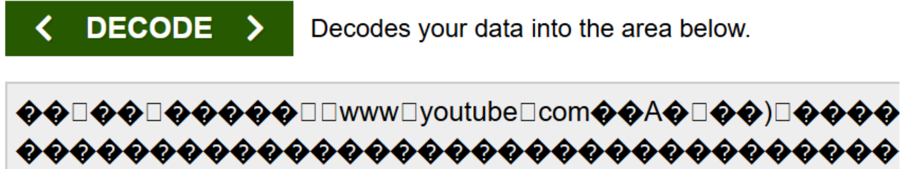
נעתיק אותו באמצעות קליק ימני – Copy – As ASCII Text.



המחרוזת המקודדת היא:

AAABAAABAAAAAAAABA3d3dwd5b3V0dWJlA2NvbQAAQQABAAApEAAAAAAAAFQADABQA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

באמצעות שימוש במקודד Base64 נוכל לקרוא הן את הדומיין המבוקש והן את סוג הבקשה (במקרה זה A, בקשת כתובת IPv4):



התשובה מגיעה בפקטה 3114 (כ-17 אלפיות שנייה בסך הכל משליחת הבקשה). תגובת השרת מתחילה ב-200 OK:

```
Hypertext Transfer Protocol Version 3
  Request Stream
    HEADERS len=15, 200 OK
      [Stream ID: 16]
      Type: HEADERS (0x0000000000000001)
```

לאחר מכן יש מידע מוצפן. ניכנס אל ה-QUIC Decrypted ונראה שם הן חזרה על השאילתא (סוג A, www.youtube.com) והן את התגובה, עם כתובות ה-IP.

```
0030  89 88 80 ec 86 85 84 83  82 00 41 d4 00 00 81 80   ········ ··A·····
0040  00 01 00 07 00 00 00 01  03 77 77 77 07 79 6f 75   ········ ·www·you
0050  74 75 62 65 03 63 6f 6d  00 00 01 00 01 c0 0c 00   tube·com ········
0060  05 00 01 00 00 00 aa 00  16 0a 79 6f 75 74 75 62   ········ ··youtub
0070  65 2d 75 69 01 6c 06 67  6f 6f 67 6c 65 c0 18 c0   e-ui·l·g oogle···
0080  2d 00 01 00 01 00 00 00  aa 00 04 8e fa 4b 6e c0   -······· ·····Kn·
0090  2d 00 01 00 01 00 00 00  aa 00 04 8e fa 4b ae c0   -······· ·····K··
00a0  2d 00 01 00 01 00 00 00  aa 00 04 8e fa 4b 4e c0   -······· ·····KN·
00b0  2d 00 01 00 01 00 00 00  aa 00 04 8e fa 4b 8e c0   -······· ·····K··
00c0  2d 00 01 00 01 00 00 00  aa 00 04 8e fa 4b ce c0   -······· ·····K··
00d0  2d 00 01 00 01 00 00 00  aa 00 04 8e fa 4b 2e 00   -······· ·····K.·
00e0  00 29 02 00 00 00 00 00  01 26 00 0c 01 22 00 00   ·)······ ·&···"··
00f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
Packet (588 bytes) | Decrypted QUIC (528 bytes) | Decoded QPACK Value (20 bytes) | Decompressed Header (516 bytes)
```

כתובת אחת מסומנת בריבוע. באמצעות מחשבון הקס' נמצא כי היא הכתובת 142.250.75.110. מתחת לכתובת זו ניתן להבחין בחמש כתובות דומות ששרת ה-DNS מספק על youtube.

בפקטה 3180 הלקוח מבצע את הפנייה הראשונית לכתובת 142.250.75.110. זו היתה נקודת הפתיחה שלנו לפענוח תעבורת ה-QUIC. סגרנו את המעגל😊

השוואה בין DoT, DoH, DoQ ו-DNS over DTLS

במהלך הלימוד סקרנו מספר שיטות לבצע שאילתות DNS, נוסף כמובן על ה-DNS הרגיל שעובר בפורט 53 מעל UDP.

בשיטת DoH מעבירים DNS מעל HTTP מעל TCP.

בשיטת DoT מעבירים DNS מעל TLS, בלי תיווך של HTTP.

בשיטת DNS over DTLS, שנכנה אותה DoD, מעבירים DNS מעל TLS שעובר מעל UDP.

ואת DoQ סקרנו כעת.

נסכם את השיטות בטבלת השוואה:

	DoQ	DoD	DoH	DoT
שכבת תעבורה	UDP	UDP	TCP	TCP
פורט	443	853	443	853
האם גורם שיושב בין השרת והלקוח יכול לחסום את השירות?	אין אפשרות להפריד מגלישה, הבקשה עוברת מוצפנת	אפשר לזהות לפי הפורט הייחודי	אין אפשרות להפריד מגלישה, הבקשה עוברת מוצפנת	אפשר לזהות לפי הפורט הייחודי
שיהוי ותקורה	אפס שיהוי מול שרת שכבר הקמנו מולו קשר QUIC, תקורה בינונית	אפס שיהוי מול שרת שסיימנו איתו TLS Handshake, תקורה מינימלית	אפס שיהוי מול שרת שכבר הקמנו איתו סוקט מאובטח, פרוטוקול ה-HTTP גורם לתקורה גבוהה יחסית עקב השדות הרבים שב-Header	אפס שיהוי מול שרת שסיימנו איתו TLS Handshake, תקורה מינימלית
שכיחות	נתמך על-ידי שרתי DNS הגדולים	פרוטוקול ניסיוני	נתמך על-ידי שרתי DNS הגדולים	נתמך על-ידי שרתי DNS הגדולים

סיכום

בתחילת הפרק הצגנו ארבע מוטיבציות עיקריות לפיתוח פרוטוקול חדש:

1. בעיית ה-Head Of Line Blocking מעל TCP. אי אפשר באמת ליצור חוסר תלות בין זרמי מידע שונים. זרם מידע אחד שמאבד פקטה, משהה את כל הזרמים שבאים אחריו.
2. בעיית ה-TCP Ossification. הפרוטוקול הגיע למצב שבו שינויים ושיפורים אינם עוברים בהצלחה רכיבי רשת שאמורים להעביר אותם, כך שלמעשה כמעט לא מעשי להוסיף יכולות חדשות.
3. הורדת כמות ה-RTT. ראינו ש-TLS 1.3 לוקח בסך הכל RTT יחיד, אך נוסף עליו ה-RTT של ה- TCP Three Way Handshake, כך שסך הכל נדרשו 2-RTT לתחילת שליחת מידע של שכבת האפליקציה.
4. ביצוע Connection Migration – מעבר חלק בין תווך פיזי אחד לאחר (לדוגמה מרשת סלולרית ל-WiFi) בלי צורך להקים מחדש את הקישור.

במהלך הפרק ראינו איך QUIC עונה על המוטיבציות הללו.

השימוש ב-DUP לבדו פתר את שלוש הבעיות הראשונות. על הדרך, השימוש ב-UDP דרש מ-QUIC לקחת אחריות על יצירת ערוץ אמין ואיפשר ליישם שיפורים בדרך הפעולה של מנגנון ה-ACK. המנגנון החדש מאפשר אישור של מספר פקטות, תוך כדי דיווח על "חורים". הוכנסו שיפורים כדי לאפשר לצדדים להעריך את ה-RTT בצורה מדוייקת יותר לעומת מה שניתן לעשות עם TCP.

השימוש ב-Connection ID בתור אחד מהשדות של QUIC מייתר את הצורך בהקמת סוקט. כל הפקטות ששייכות לקישור מסויים מזוהות על-ידי הצדדים לתקשורת. אפשר לעבור כתובת IP ועדיין הצד השני יידע שפקטה שהגיעה שייכת להמשך קישור קיים.

פרוטוקול QUIC מצפין לא רק את המידע אלא גם את ה-Headerים. מי שעוקב אחרי תקשורת שאינה שלו, לא יצליח אפילו להרכיב יחד את רצף המידע, ששבור בין פריימים שונים.

בזאת סיימנו את הדיון ב-QUIC והשלמנו את ההתקדמות הטכנולוגית עד למועד כתיבת שורות אלה.

נסכם בהשוואה בין HTTP/1.1 לבין HTTP/2 ו-HTTP/3:

	HTTP/3	HTTP/2	HTTP/1.1
שכבת תעבורה	UDP	TCP	TCP
פורט	443	443	443
אבטחה	TLS 1.3	TLS 1.2, TLS 1.3	לא חובה
דחיסת Header	QPACK	HPACK	אין
הורדת משאבים במקביל	אפשר, לכל משאב יש Stream ID משלו	אפשר, לכל משאב יש Stream ID משלו, אך עלול לקרות Head of Line Blocking בגלל השימוש ב-TCP	באמצעות פתיחת מספר סוקטים
RTT מתחילת התקשרות ועד שליחת בקשת GET	1, או 0 במקרה של Session Resumption	4, 3, או 2. מתוכם אחד עבור ה- TCP Three Way Handshake והיתר משתנים לפי גרסת TLS והאם יש Session Resumption	1 אם אין TLS, רק TCP Three Way Handshake. 3 או 4 אם יש גם TLS Handshake 1.2, תלוי אם יש Session Resumption.
ניידות	יש. אפשר להחליף תווך פיזי ולעבור כתובת IP תוך כדי הקישור	אין. החלפת תווך פיזי, שגורמת לשינוי כתובת IP, דורשת הקמת סוקט מחדש	אין. החלפת תווך פיזי, שגורמת לשינוי כתובת IP, דורשת הקמת סוקט מחדש

22.1 תרגיל QUIC

בתרגיל זה נבצע הסנפה של תעבורה וננתח את פרוטוקול QUIC.

כהכנה, בצעו את השלבים הבאים:

1. אפשרו פרוטוקול QUIC בדפדפן.
2. קיבעו בדפדפן שימוש ב-Secure DNS. לשיקולכם אם לבחור את שרת ה-DNS של גוגל או של Cloudflare.
3. ודאו שמשתנה הסביבה SSLKEYLOGFILE מוגדר אצלכם.
4. בדפדפן, מחקו את היסטוריית הגלישה של רבע השעה האחרונה, כדי שתהליכי הקמת הקשר עם השרת לא יהיו מקוצרים, אם גלשתם אליו לאחרונה.
5. בצעו אתחול למחשב והתחילו את ההסנפה לפני ההפעלה הראשונה של הדפדפן. הסיבה לכך היא שאחד הדברים הראשונים שהדפדפן מבצע עם הפעלתו הוא יצירת קשר עם שרת ה-DNS. אם ההסנפה לא תכלול את הפקטות הללו, Wireshark לא יוכל לפענח את התקשורת עם שרת ה-DNS.
6. גילשו לאתר כלשהו שתומך ב-QUIC. אם אינכם מכירים ורוצים לגוון ולא לגלוש ל-YouTube כמו בהסנפה שצורפה לפרק זה, אפשר לבקש מ-AI רשימה של מספר אתרים.

שלב המחקר – ענו על השאלות הבאות:

<u>DNS</u>

1. מיצאו את שאילתת ה-DNS שבה הלקוח מבקש משרת ה-DNS את כתובת ה-IP של האתר שביקשתם. ודאו שהפרוטוקול הוא DoQ.
2. מיצאו את תשובת ה-DNS. פענחו אותה וודאו שהתשובה אכן עונה לשאילתא שמצאתם. טיפ: אפשר לחפש מחרוזת או רצף ערכים הקסדצימליים באמצעות הפילטר

   quic.stream_data contains

   לדוגמה:

   quic.stream_data contains “youtube”

   quic.stream_data contains 8e:fa:4b:fe

מעבר מ-TCP ל-QUIC

1. חפשו תעבורת TCP אל השרת המבוקש. אם קיימת כזו, חפשו תחת HTTP/2 את שדה ה-Alt Svc ומיצאו את התמיכה ב-h3, פורט 443.

QUIC

1. מיצאו את ה-initial של QUIC לאתר המבוקש. מיצאו את ה-Client Hello של TLS 1.3.
2. מיצאו את השדות הבאים:
    a. SNI
    b. QUIC transport parameters
3. בחרו פקטת ACK ופענחו את שרשרת האישורים והחורים שלה (מומלץ לחפש ACK שיש בו Gap). אם קיים חור, עיקבו אחרי רצף ה-Packet Numbers וודאו שרצף המספרים תואם את החור ב-ACK.

כל הכבוד!

מה צופן העתיד?

קשה לצפות איך ייראה חלקו השלישי של ספר רשתות מחשבים, אך הנה מספר תחזיות:

השינויים הטכנולוגיים הולכים ומאיצים ויחד איתם גם **קצב העדכון של הפרוטוקולים נהיה מואץ**. לפני עשור תעבורת HTTP לא מוצפנת היתה נפוצה. כיום גרסה 1.2 של TLS נחשבת מיושנת. אתרים שעדכנו לאחרונה גרסת TLS ל-1.3, כבר נמצאים בפיגור אחרי QUIC. סביר שהעדכונים הבאים יצטרכו להיות תכופים יותר.

כניסה של הצפנות פוסט קוואנטיות. מחשבים קוואנטיים יוכלו לפתור בזמן סביר את הבעיות המתמטיות שעומדות מאחרי RSA ו-DH. המעוניינים בקריאה נוספת מוזמנים לקרוא על אלגוריתם Shor. כבר כיום, יש רעיונות והצעות של תקנים שיכולים לעמוד בפני מחשבים קוואנטיים. ייתכן שהחלק השלישי יצטרך להביא את התאוריה של ההצפנות הפוסט קוואנטיות ולהסביר כיצד תהליך ה-Handshake השתנה בהתאם.

שימוש גובר ב-QUIC יוביל למעבר לגרסאות מתקדמות יותר של הפרוטוקול. גרסת QUIC שאיתה בוצעה ההסנפה בפרק זה היא גרסה 1. גרסה 2 כבר קיימת, RFC 9369 מחודש מאי 2023.

הסתרת הדומיין שהגלישה מתבצעת אליו. כיום, ה-Client Hello מכיל שדה של SNI – Server Name Indication. גורמי מדינה שרוצים לצנזר תקשורת של אזרחים יכולים להשתמש בשדה זה כדי למנוע גישה לאתרים שונים. מסתמן שפרוטוקול ECH, Encrypted Client Hello, ייכנס לשימוש בקרוב. רשומות DNS יכילו שדה של מפתח ציבורי של השרת, כך שהלקוח יוכל להצפין חלק מה-Client Hello, וספציפית את שדה ה-SNI.

ותחזית אחרונה, שהיא אולי גם משאלת לב – **הבנה עמוקה של עולם המחשבים תמשיך להיות משמעותית**. נכון, תפקידים טכנולוגיים מסויימים יהיו מיותרים עקב בינה מלאכותית, אולם תפקידים שדורשים הבנה עמוקה לא רק שלא יוחלפו על-ידי בינה מלאכותית אלא יהפכו משמעותיים יותר. כשם שהחלפת שפת אסמבלי בשפות עיליות לא הפכה את הידע באסמבלי למיותר אלא פתחה עולם חדש של חיפוש חולשות ופרצות אבטחה, כך כניסת הבינה המלאכותית תיצור אתגרים חדשים. מי שיידעו להבין דברים לעומק, לנתח ולחקור, ולהבין מתי הבינה המלאכותית נותנת תוצר שאינו אמין, הם יהיו הכוכבים והכוכבות של עולם המודיעין וההייטק המצפה לנו בטווח הזמן הנראה לעין.

כולי תקווה שספר זה קידם אתכם בדרך להבנה עמוקה.

חומרי העשרה מומלצים

להלן רשימת חומרי העשרה מומלצים שיש אליהם הפניות בפרקים השונים.

פרק 16

AES Explained (Advanced Encryption Standard) - Computerphile

AES GCM (Advanced Encryption Standard in Galois Counter Mode) - Computerphile

תיאור מתקפת ה-BEAST על פרוטוקול SSL

פרק 18

How we created the first SHA-1 collision and what it means for hash security - Defcon 25

פרק 19

Ep 3: DigiNotar, You are the Weakest Link, Good Bye!

The Blockchain & Bitcoin - Computerphile

פרק 20

The POODLE Attack

פרק 21

Elliptic Curves - Computerphile

פרק 22

כשדחיסה היא crime ויוצרת breach באבטחה