האוניברסיטה הפתוחה

המחלקה למתמטיקה ולמדעי המחשב

עבודה מסכמת במדעי המחשב:

שיטות מעקף למנגנוני אנטי-וירוסים

עבודה זו מוגשת כחלק מהדרישות לקבלת תואר

"מוסמך למדעים" .M.Sc במדעי המחשב

באוניברסיטה הפתוחה

המחלקה למדעי המחשב

על-ידי

אייל בביאן

העבודה הוכנה בהנחייתו של פרופ' אהוד גודס

אוקטובר 2024

תוכן עניינים

רשימת איורים

גרסאות המסמך

מס' גרסה	שינויים
1.0	טיוטה ראשונית
2.0	גרסה סופית

1. תקציר העבודה

כיום, עולם ההגנה בסייבר מפותח מאוד ואחד האתגרים הנפוצים כיום בתחום הנוזקות הוא הסתתרות ממוצרי הגנה. זהו משחק של חתול ועכבר כאשר הנוזקה שואפת להיות יותר מתוחכמת מהאנטי-וירוס ואילו האנטי-וירוס שואף להיות יותר מתוחכם מהנוזקה.

לצורך כך, תוכנות אנטי-וירוס מממשות מספר מנגנונים שונים הפועלים ומתמקדים באזורים שונים של פעולת המערכת על מנת לתפוס התנהגות בלתי רצויה העלולה להיחשב כזדונית. נוזקות גם הן לומדות את דרכי הפעולה של אותם המנגנונים ומנסות להתחמק מהם גם בשעת ריצה ואפילו כאשר הנוזקה לא רצה.

קיימות מגוון רב של דרכים לזיהוי נוזקות הנמצאות בשימוש ע"י אנטי-וירוסים. תחום אחד בו נתמקד הוא זיהוי שיטות ההסתתרות של הנוזקות. לשם כך אסביר ביצד פועלות תוכנות אנטי-וירוסים ומה הם המנגנונים אשר הם משתמשות בהם על מנת לאתר נוזקה, תוך כדי הבנת מושגי יסוד בתחום כגון: ניתוח סטטי, ניתוח דינמי, חתימות, סורקים ועוד.

בעבודה זו נסקור את דרכי פעולת האנטי-וירוסים ודרכי מעקף אפשריות עבורם, הן בשעת הריצה והן בעת שהיה על הדיסק. מרבית השיטות שנסקור הן לזיהוי נוזקות והן למעקף האנטי-וירוסים הינם כלליים ורלוונטיים למערכות הפעלה שונות.

בחלק הראשון אתמקד בזיהוי נוזקות שוהות על הדיסק ופרט על ניתוח סטטי וניתוח דינמי. אסביר מהו מנגנון ניתוח סטטי ואציג שיטות נפוצות למעקפו. בנוסף, אתמקד בשיטה מיוחדת המוצגת במספר מאמרים אשר משלבת שימוש בטכנולוגיית ROP chains על מנת לבצע מעקף לניתוח סטטי.

לאחר מכן, נסביר מהו ניתוח דינמי ונציג מספר שיטות שונות אשר נוזקות ישתמשו בהן על מנת לעקוף מנגנון זה.

בחלק השני של העבודה, אתמקד בהצגת שיטות זיהוי לנוזקות בזמן ריצה וכנגדן, אציג שיטות למעקף. אציג שיטה הנפוצה בקרב מרבית תוכנות האנטי-וירוס המשלבת ביצוע hooking לפונקציות WinAPI. אסביר את המושגים והטכנולוגיות המאפשרות את שיטת הזיהוי הזו ואציג שיטות למעקף אשר נמצאו בקרב נוזקות.

השיטה האחרונה בה אתמקד בעבודה היא זיהוי נוזקות בזמן ריצה ע"י שימוש במנגנון AMSI. מנגנון זה מגיע בחלק ממערכת הפעלה windows ופתוח להרחבה עבור תוכנות אנטי-וירוס, דבר אשר הופך אותו לנפוץ מאוד. לאחר שאסביר על המנגנון, אציג שיטות מעקף נאיביות בנוסף לשיטות מעקף מתוחכמות עבור מנגנון AMSI.

לבסוף, אסקור בקצרה את הפרויקט המעשי אשר מלווה את העבודה. במסגרת הפרויקט מימשתי תוכנת אנטי-וירוס אשר מכילה אנליזות לזיהוי בזמן שהיה ובזמן ריצה תוך שילוב ממשק גרפי נוח למשתמש.

2. מוטיבציה

בעשור האחרון, עם התפשטותו הרחבה של האינטרנט והמעבר לעבודה דיגיטלית בכמעט כל תחומי החיים, גברה חשיבותה של אבטחת המידע. תוכנות האנטי-וירוס פעלו בחזית הקרב הזה, בהתמודדות היומיומית נגד תוקפים המבקשים להפר את המידע האישי והארגוני. בעידן בו הנתונים הם זהב, ובו אנשים, ארגונים ומדינות תוקפים ונתקפים בצורה סייברנטית, הצורך בהגנה מתקדמת ואפקטיבית הוא גבוה מאי פעם.

עולם נוזקות גדל לא רק בשל ההתפשטות הרחבה של הטכנולוגיה, אלא גם בשל עלייתם של איכות ההתקפות. ארגוני תקיפה, קטנים וגדולים כאחד, משתמשים בטכנולוגיות מתקדמות ובטקטיקות שכל פעם ממציאות את עצמן מחדש. המורכבות הזו מצריכה התאמה מתמדת של פתרונות ההגנה, ומצביעה על הצורך הגובר להשקיע במערכות אבטחה מתקדמות, כגון תוכנות אנטי-וירוס.

אבטחת המידע היא לא רק בעיה טכנולוגית, אלא גם בעיה אסטרטגית ותרבותית. כאשר כל פריט מידע, החל מקודי הגישה לבנק ועד לתכנון פרויקטים מוסדיים, יכול להיות נגיש לתוקפים, הצורך בהגנה הוא גדול יותר מאי פעם.

אנטי-וירוסים הפכו לכלי מרכזי באמצעותו המשתמשים מגנים על מערכותיהם מפני התקפות דיגיטליות. נגד הרקע השונה של תוכנות זדוניות ואיומי סייבר, האנטי-וירוסים פועל באמצעות תהליכים שונים כדי לזהות, לחסום ולהסיר תוכנות רעות מהמערכת הפועלת. מהימים הראשונים של התפשטות הווירוסים המחשביים, האנטי-וירוסים התפתחו מאוד והפכו לכלים מתקדמים ומורכבים, הכוללים אלגוריתמים מתקדמים, סקירה פעילה של הנתונים, והשוואה לתבניות של נוזקות.

3. מבוא לאנטי-וירוסים

3.1 מהי נוזקה?

נוזקה (Malware) היא בעצם קוד אשר נועד לבצע פעולות זדוניות. נוזקה יכולה להיות מקופלת בקובץ הרצה בינארי, סקריפט, דרייבר או כל סוג אחר של תוכנה. מטרותיה של נוזקה יכולות להיות שונות ומגוונות בהתאם לרצון התוקף, דוגמאות למטרות אפשריות הן:

- גניבת מידע רגיש
- ריגול
- השבתת מערכת
- הצפנת המערכת למטרות כופר

נהוג לחלק נוזקות לתתי-קבוצות שונות על פי הפונקציונליות של הנוזקה:

- **Worm:** נוזקה אשר משתכפלת בין מחשבים ומפיצה את עצמה באופן אוטומטי.
- **Remote Access Trojan (RAT):** נוזקה אשר מאפשרת לתוקף גישה מרוחקת למחשב וביצוע פעולות זדוניות מרחוק בשליטתו.
- **Botnets:** אוסף של מחשבים אשר מודבקים עם אותה הנוזקה. שליטה על הנוזקה מאפשרת ביצוע פעולה בודדת בהיקף מאוד גדול בו-זמנית. כך ניתן בקלות לבצע השבתת מערכת (DDOS).
- **Ransomware:** נוזקה אשר מצפינה את המערכת ולא מאפשרת גישה עד לתשלום כופר ע"י המשתמש.
- **Dropper:** נוזקה אשר מורידה/מתקינה נוזקה אחרת ממקור חיצוני.
- **Packer:** תוכנה אשר כתובה באופן מוצפן/מכווץ על הדיסק ומחלצת את תוכנה האמיתי בזמן ריצה.

3.2 מהו אנטי-וירוס?

אנטי-וירוס הוא מוצר תוכנה אשר נועד להגן על מערכת מפני נוזקות. לרוב הוא נועד בתור פיתרון מניעתי מפני נוזקות עתידיות אך במקרה של הדבקה ע"י נוזקה הוא יכול לאתר את הנוזקה, להסיר את ההדבקה ולמחוק את הנוזקה (מאמרים [1],[2] ו-[3]).

נהוג לחשוב (באופן מוטעה) שאנטי-וירוסים מספקים הגנה הרמטית על מערכת מפני נוזקות. האמת היא שאנטי-וירוסים מתמחים בעיקר בזיהוי ואיתור נוזקות מוכרות ולפעמים יתקשו לאתר ולזהות נוזקה חדשה שאיננה עדיין פומבית.

3.3 מאפייני אנטי-וירוס

רוב תוכנות האנטי-וירוס חולקות מספר מאפיינים זהים, כאשר נסתכל עליהם נוכל להבין יותר טוב את דרכי הפעולה שלהם:

- **כתובים בשפת native:** רוב האנטי-וירוסים כתובים בשפה שהיא non-managed כגון C או C++ מאחר והן נחשבות ליעילות יותר. תוכנות אנטי-וירוס מבצעות המון פעולות ברקע אשר משפיעות על המערכת ולעיתים גם מתערבות בפעילות המערכת. מסיבה זו האנטי-וירוס ייכתב בשפה שיותר קרובה למערכת על מנת לספק ביצועים מרביים. לשפות אלו קיימים גם חסרונות מאחר והן יותר חשופות לזליגות זיכרון ו-corruptions.
- **חתימות:** חתימות הן אוסף של דפוסים לזיהוי נוזקות. חתימות יכולות להיות מאוד פשוטות כגון MD5 על נוזקה יודעה או יכולות להיות יותר מתוחכמות כגון רשימת כל ה-Import-ים אשר הנוזקה מייבאת. חתימות הן הבסיס לזיהוי ואיתור נוזקות של אנטי-וירוסים ומשתנות בין מוצר למוצר. החתימות צריכות לספק איזון בין חתימה מאוד ספציפית על נוזקה לבין חתימה יותר מדי גנרית אשר עלולה להקפיץ false-positives.
- **Scanners:** סורק (Scanner) הוא מנגנון אנטי-וירוס אשר מבצע סריקה על קבצי מערכת (באופן יזום ע"י משתמש או באופן מחזורי פאסיבי) ומריץ חתימות מתאימות אל מול הקבצים על מנת להציף ולזהות נוזקות במערכת.
- **Emulators:** אמולטור (Emulator) הוא מנגנון אנטי-וירוס אשר יודע להריץ קוד בסביבה מדומה (כך שפעולות הקוד לא ישפיעו ישירות על המערכת) וע"י כך לבחון את פעולות הקוד.
- **Compressors:** לעיתים נוזקות יסתתרו בתוך קבצים מכווצים. Compressor הוא מנגנון אנטי-וירוס אשר יודע לחלץ קבצים מכווצים ולסרוק את התוכן שלהם על מנת להציף נוזקות.
- **Firewall:** firewall (חומת-אש) הוא מנגנון אנטי-וירוס אשר סורק ומנטר תעבורה רשתית הנכנסת ויוצאת מהמחשב על מנת לזהות תקיפות רשתיות.
- **הגנה עצמית:** מאחר ונוזקות יודעות שאנטי-וירוס הוא איום נפוץ עבורן, הן ינסו לעיתים לסגור את תוכנת האנטי-וירוס ברגע שהן מצליחות לחדור אל מחשב כלשהו. אנטי-וירוסים מודעים לניסיון זה ולרוב לא יאפשרו סגירה של האנטי-וירוס בצורה פשוטה ע"י קריאה לפונקציות מערכת.

3.4 סוגי מנגנוני הגנה

כפי שציינו לעיל מטרתה העיקרית של תוכנת אנטי-וירוס היא לזהות ולמנוע ריצה של תוכנות זדוניות במערכת ההפעלה.

על מנת לממש הגנה זו בצורה האיכותית ביותר, קיימים מספר מנגנוני הגנה שונים אשר אחראים על השגת מטרה זו. מנגנונים אלו נבדלים אחד מן השני בדרך העבודה שלהם ובשיטת הסריקה שלהם.

תוכנות האנטי-וירוס משתמשות במספר מנגנונים עיקריים:

1. **ניתוח סטטי** – ניתוח קוד שוהה על מערכת הקבצים מבלי להריץ אותו.
2. **ניתוח דינאמי** – דימוי הרצה של קוד בסביבה מוגנת ואיתור התנהגות זדונית על סמך הפעולות שהוא מבצע.
3. **ניתוח זיכרון** – ניתוח תהליך בזמן ריצה לאיתור התנהגות זדונית.
4. **ניתוח תקשורת** – ניתוח תעבורה מהמחשב/אל המחשב לצורך איתור תקשורת זדונית. כיום, ניתוח תקשורת נהפך לבעיה קשה מאחר ורוב התעבורה כיום עוברת מוצפנת מעל פרוטוקולים כדוגמת TLS/SSL אך קיימות דרכים שונות לחתום גם את התקשורת המוצפנת (לדוגמה JA3/JA4).

יש לציין כי המנגנונים הנ"ל משלימים אחד את השני ומספקים מענה הגנה כולל על המערכת.

במהלך העבודה אחלק את המנגנונים הללו ל2 קבוצות:

- **ניתוח קוד שוהה** – מנגנוני אנטי-וירוס אשר אחראים על ניתוח איומים אשר נמצאים במערכת הקבצים ואינם רצים כעת. מנגנונים אלו יסרקו את הדיסק באופן מחזורי ובנוסף במקרה של אירועים מסוימים כגון יצירה של קובץ חדש במערכת. מנגנונים מסוג זה אשר נתמקד בהם במהלך העבודה יהיו הניתוח הסטטי והניתוח הדינאמי.
- **ניתוח קוד בזמן ריצה** – מנגנוני אנטי וירוס אשר אחראים על ניתוח איומים בזמן ריצה של תהליך. לרוב מנגנונים אלו יבצעו אחד מ2 דברים: סריקה של זיכרון התהליכים במערכת לצורך בדיקת אנומליות שונות/חשש לפעילויות זדוניות או זיהוי של פעילות זדונית ע"י תהליך כלשהו במערכת. המנגנון העיקרי מסוג זה אשר נתמקד בו במהלך העבודה יהיה ניתוח הזיכרון.

4. זיהוי בזמן שהיה

4.1 ניתוח סטטי

4.1.1 מהו ניתוח סטטי?

ניתוח סטטי הוא אחד ממנגנוני ההגנה וזיהוי האיומים של תוכנות אנטי וירוס. מטרתו היא לנתח את הקוד של התוכנית מבלי להריץ אותו. בשיטה זו, ניתן לזהות תוכנה זדונית מבלי להריץ אותה ובכך לא לסכן את המערכת. תוכנת האנטי-וירוס מנתחת את הקוד הבינארי של התוכנית בכדי לקבוע את רמת הסיכון שלה. קיימות מספר שיטות שונות על מנת לקבוע את רמת הסיכון של קוד אך לפני שנדון בהן נסביר על מזהים וחתימות של אנטי-וירוסים.

4.1.2 חתימות ומזהים

4.1.2.1 מזהים

מזהה (IoC - Indicator of compromise), הינו פריט מידע ספציפי וייחודי במידת האפשר עבור אפיון התנהגות זדונית כלשהי. מזהים יכולים להיות לדוגמה: כתובת IP אשר מעורבת בתקשורת החשודה, כתובת URL כלשהי, Hash של קובץ זדוני, רצף של בתים ייחודיים אשר מעידים על פעילות חשודה של הקוד ועוד. על מנת שמזהה כלשהו יהיה איכותי עליו לאפיין בצורה המיטבית ביותר את ההתנהגות הספציפית אותה הוא רוצה לאפיין ולצמצם מקרי False Positives.

קיימים 2 קשיים מרכזיים באפיון התנהגות זדונית על סמך מזהה בודד:

- יכולים להיווצר הרבה False Positives – לא בהכרח יהיה קיים מזהה בודד אשר מספק אפיון ספציפי עבור ההתנהגות הזדונית.
- הרבה מזהים יכולים להיות ברי החלפה – לדוגמה, בהינתן נוזקה אשר מתקשרת עם כתובת ה-IP x.x.x.x נוכל להריץ בדיוק את אותה הנוזקה המתקשרת עם הכתובת y.y.y.y ובכך לעקוף את החתימות. לכן, נדרש למצוא מזהים שאינם ברי החלפה בקלות (רצף בתים קריטיים בליבת הפרויקט, פרוטוקול התקשורת של הנוזקה ועוד).

4.1.2.2 חתימות

חתימות כפי שצוין כבר לעיל, הן אחד ממנגנוניו הבסיסיים של אנטי-וירוסים על מנת להציף התנהגות זדונית. חתימות הן בעצם אוסף של מזהים אשר מאפיינים בצורה יותר חזקה התנהגות זדונית. מאחר וקשה מאוד למצוא מזהה יחיד בודד אשר יאפיין את ההתנהגות הזדונית בצורה איכותית, החתימות מנסות להצליב בין אוסף של מזהים שונים על מנת לתת אמירה יותר איכותית.

לכל אנטי וירוס קיים מילון של חתימות. מילונים כאלה הם מסדי נתונים גדולים המורכבים מרצפי בתים קצרים, או חתימות מתוכנות זדוניות. בעת ניתוח תוכנית, האנטי וירוס מצליב את מסד הנתונים עם הקוד הבינארי של התוכנית ומחפש מקרים של חתימות ידועות. במידה ונמצאה התאמה כלשהי הקוד מסווג כאיום.

4.1.3 ניתוח סטטי - סוגי חתימות שונות

Checksum 4.1.3.1

Checksum היא שיטת החתימות הנפוצה ביותר באנטי וירוסים. אלגוריתם ה-CRC (Cyclic Redundancy Check) מקבל מאגר כקלט ומפיק Checksum (4 בתים עבור CRC32). checksum-ים של נוזקות מושוות ל-checksum-ים של כל הקובץ או checksum-ים של חלקים ספציפיים בהתאם לפורמט של הקובץ (כגון PE או ELF). אלגוריתם ה-CRC מהיר, אך הוא מייצר False Positives רבים בגלל שהוא רגיש להתנגשויות, כמוצג במאמר [6].

אנטי-וירוסים רבים יוצרים חתימות דמויות CRC מותאמות אישית. חלקם, למשל, מבצעים פעולת XOR עם ה-CRC של חלק מקבצי הרצה ומשתמשים בפלט כחתימה של קובץ ה-PE. אחרים מייצרים checksum-ים של CRC של חלקים מהקובץ (למשל כותרת עליונה וכותרת תחתונה) ומשתמשים ב-hash המיוצר כחתימה. עם זאת, checksum-ים מותאמים אישית אלו סובלים מאותן בעיות כמו ה-checksum-ים המקוריים, הם נוטים לתוצאות שגויות מכיוון שקל למצוא התנגשויות.

Byte Streams 4.1.3.2

Byte Stream הוא רצף של בתים בקוד עליהם תהיה קיימת חתימה. זוהי החתימה הפשוטה ביותר אך עם זאת יכולה להיות אפקטיבית מאוד (מוצגת במאמר [5]). המטרה היא לחתום על רצף בתים ייחודי וייעודי עבור נוזקה אשר לא יהיה בר החלפה בקלות רבה.

Strings 4.1.3.3

זהו מקרה פרטי של Byte Stream. לפעמים יהיו הרבה מאוד מזהים אשר כתובים בתוך הקוד עצמו (לדוגמה: כתובת IP לתקשורת, הודעת שגיאה ייחודית, נתיבים לקבצים). ההבדל העיקרי הוא שחתימה על בתים לרוב תתבצע על לוגיקה (קריאה ייחודית לפונקציה לדוגמה) ואילו חתימה על מחרוזות תתבצע על משאבים בהם הקוד מבצע שימוש.

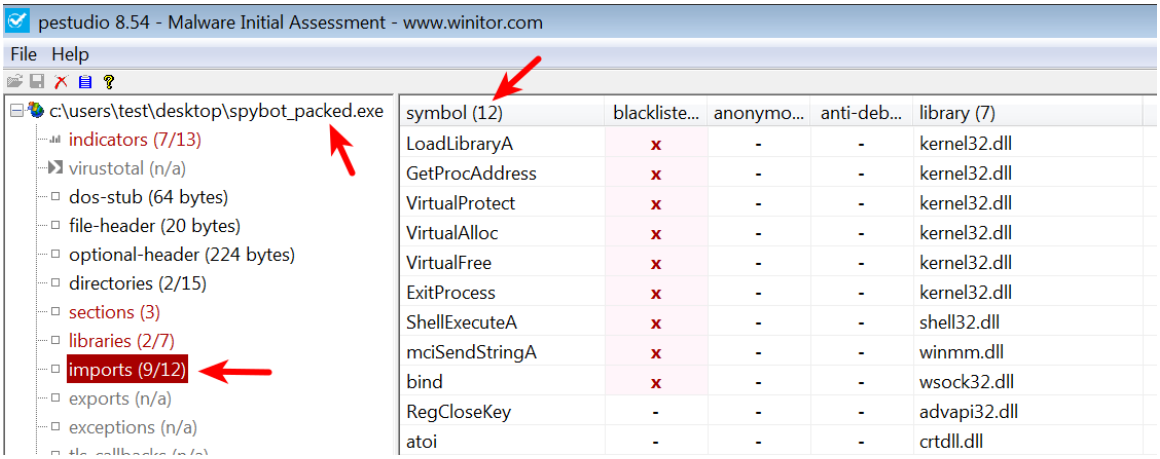
Fuzzy Hash 4.1.3.4

חתימה יותר מתקדמת אשר נעשית בה שימוש בקרב אנטי-וירוסים ומוצגת במאמר [6]. בשונה מ-hash רגיל אשר יודע לחתום בצורה חד-חד ערכית עבור כל קלט, מטרתו של ה-fuzzy hash היא לאתר 2 קלטים דומים אך לא זהים. ע"י כך נוכל לאתר בצורה איכותית וריאנטים שונים של נוזקות למרות "ניסיונות קוסמטיים" שבוצעו על מנת לעקוף חתימות ולהסתיר מזהים.

PE Header information 4.1.3.5

מקור מספר [2] מראה לנו כי קיים הרבה מידע אשר ניתן לחלץ מה-Headers של קובץ ההרצה של התוכנה (PE – Portable Executable), לדוגמה: נוכל ללמוד הרבה על אופי התוכנה רק מהפונקציות אשר היא מייבאת. במידה ונראה את הפונקציה CreateFile אשר מיובאת מתוך kernel32.dll נוכל להסיק שקיימת התנהגות כחלק מהקוד אשר מנסה ליצור קובץ על הדיסק.

נוכל לאפיין התנהגות זדונית ע"י הצלבה של מספר פונקציות בהן הקוד משתמש. נסתכל לדוגמא על איור 1:



איור 1: טבלת ה-import-ים של spybot, נלקח ממקור [2]

באיור 1, מדובר בגרסא Packed של נוזקה בשם spybot. ניתן לראות שבקושי קיימים import-ים לתוכנה, אך מתוך המספר המצומצם הזה ניתן לראות שקיימות הפונקציות הבאות:

- LoadLibraryA – פונקציה האחראית על טעינת DLL.
- GetProcAddress – פונקציה האחראית על חיפוש כתובת של פונקציה מתוך DLL טעון.
- VirtualProtect – פונקציה האחראית על שינוי הרשאות של מרחבי זיכרון בזמן ריצה.
- ShellExecuteA – פונקציה האחראית על הרצת קוד מערכת.

שילוב שלושת הפונקציות הראשונות נמצא בשימוש נפוץ אצל נוזקות המבצעות טעינה רפלקטיבית של קוד (טעינת קוד בזמן ריצה ממקור חיצוני). ואילו הפונקציה האחרונה מריצה קוד מערכת בהינתן קלט כלשהו.

על סמך אבחנה זו, בנוסף לידיעה שקיימים מספר מאוד מצומצם של פונקציות בפרויקט מעלה את רמת החשד שמדובר בנוזקה אשר מבצעת טעינה של קוד בזמן ריצה.

קיים עוד מלא מידע שניתן לחלץ מתוך ה-Header-ים של קובץ הרצה:

- **Export-ים** – במקרה של DLL נוכל לראות את כל הפונקציות אשר הוא מייחצן לשימוש ובכך להבין האם מדובר בספרייה זדונית.
- **סביבת קימפול** – אחד מהנתונים שקיימים בתוך כל קובץ הוא הסביבה והתאריך שהקובץ קומפל בו. הרבה כותבי נוזקות ירצו להסתיר מידע זה או שלעיתים יזבלו אותו כדי לא לקשר אליהם. לדוגמה: במקרה שנראה קוד שקומפל בשנת 2050 נוכל להסיק שמישהו התערב ידנית בקוד מה שיעלה לנו את רמת החשד לגבי קובץ ההרצה.
- **Resource-ים** – לעיתים, נוזקות יכילו בתוכם resource-ים מעניינים על מנת לנסות להסתיר את הפעילות שלהם. לדוגמה במקרה של טעינה רפלקטיבית יכול להיות מימוש בו הקוד עצמו קורא את התוכן של resource פנימי שלו, טוען אותו לזיכרון ומריץ אותו. כאשר

נדע לחלץ את ה-resource-ים האלה מתוך הקובץ, נוכל לסרוק גם אותם על מנת לאתר התנהגות חשודה.

4.1.4 שיטות למעקף ניתוח סטטי

נסקור מספר שיטות שונות לביצוע מעקף ניתוח סטטי אשר נמצאות בשימוש ע"י נוזקות. יש לציין, שלא כל שיטה תעבוד בהכרח תמיד. הצלחת המעקף תלויה בחוזק החתימה ולא תמיד מפתח הנוזקה יוכל לדעת בדיוק מהו מוצר ההגנה אותו הוא נדרש לעקוף ואלו חתימות מוטמעות בו.

Binary Patching 4.1.4.1

השיטה הכי בסיסית ונאיבית. מאחר וחתימות מתבססות על מחרוזות ייחודיות או על hash של הנוזקה, הדרך הפשוטה ביותר לעקוף חתימה היא ע"י שינוי ערכים בנוזקה עצמה. נוכל לערוך את קוד הנוזקה ולקמפל מחדש במידת הצורך או אפילו לערוך את הבינארי עצמו ולתקן אותו (Patching).

שיטה זו מתבססת על כך שאנחנו בסיכוי טוב יודעים שמדובר בחתימה פשוטה אשר מחפשת רצף תווים ייחודי ואותו נחליף.

Obfuscation 4.1.4.2

אובפוסקציה היא טכניקה לשינוי קוד על מנת להפוך אותו לפחות קריא. קיימות 2 דרכים עיקריות לביצוע אובפוסקציה:

- Rename obfuscation
- Control-flow obfuscation

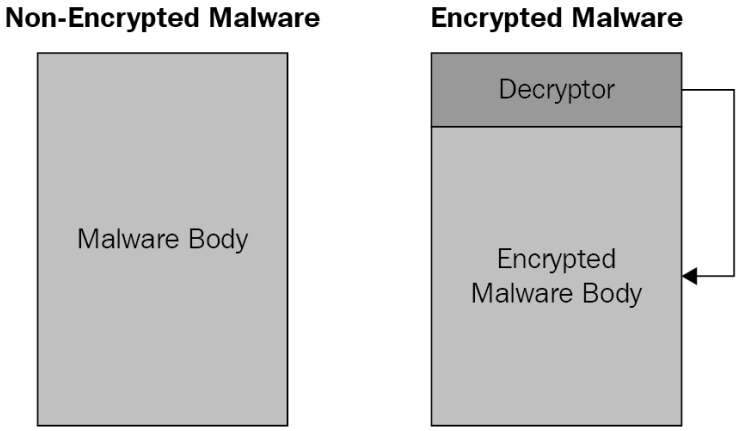
Rename obfuscation – שינוי כל המשתנים בקוד המקור של הנוזקה לשמות לא משמעותיים. שיטה זו נפוצה בעיקר בשפות script או בשפות managed (שפות אשר הקומפיילר ממיר לקוד ביניים, בזמן ריצה קיים מנוע אשר ממיר את קוד הביניים לקוד מכונה). בשפות אלו, נוכל לשחזר את קוד המקור של הנוזקה יחסית בקלות ובאמצעות ניתוח של קוד המקור נוכל להבין בדיוק מה הנוזקה עושה. תהליך ה-Rename obfuscation מקשה על כך ומוסיף עוד שלב בדרך להבנת הלוגיקה של הנוזקה.

Control-flow obfuscation – המרת מבנה הקוד למבנה אחר כך שיישמר עדיין את הלוגיקה של הנוזקה. שיטה זו מתייחסת לכך שקיימות חתימות (נרחיב על כך בפרק על ניתוח דינאמי) אשר מסתמכות על מבנה הקוד ועל סדר הפעולות המתבצעות בקוד. ע"י שינוי מבנה הקוד נוכל לקבל מבנה אחר לגמרי שעוקף את החתימה. בהתייחסות לניתוח סטטי, שינוי של מבנה הקוד יישנה גם את הבתים של הנוזקה (מאחר וה-Opcode-ים של הפקודות השתנו). במקרה זה, קיים סיכוי שנעקוף חתימות אשר מתבססות על תנאים ולוגיקה hardcoded.

Encryption 4.1.4.3

הצפנה היא עוד שיטה נפוצה אשר נוזקות משתמשות בה על מנת לעקוף מנגנוני ניתוח סטטי. כפי שהוסבר לעיל, ניתוח סטטי הוא הרצת חתימות סטטיות על הקובץ מבלי להריץ אותו. בכך, אנחנו יכולים לבצע יחסית מהירה מבלי לפגוע/לסכן את המערכת.

אולם, ע"י ניתוח סטטי בלבד נוכל לפספס נוזקות אשר לא מייחצנות את ההתנהגות האמיתית שלהם. לדוגמה במקרה של הצפנה, הנוזקה יכולה להצפין את עצמה ולהשאיר חלק קוד בתחילת ההרצה אשר מפענח את כל התוכן המוצפן ומריץ אותו. כך נוכל להתגבר על מרבית סוגי החתימות שהזכרנו לעיל (strings, hashes, bytes streams...), איור 2 ממחיש את הקונספט של נוזקה מוצפנת.



איור 2: תרשים packed malware, נלקח ממקור [1]

Packers 4.1.4.4

Packers הם תוכנות אשר משמשות לכיווץ קבצי הרצה בינאריים. בדומה לקונספט שהצגנו לעיל בחלק על הצפנה, נוכל להשתמש בתוכנה מסוג packer על מנת לכווץ ולהצפין את הנוזקה ולעטוף אותה ע"י חלק קוד בתחילתה אשר מחלץ אותה וממשיך את ההרצה.

כיצד packer עובד?

קונספט ההצפנה שהצגנו לעיל לרוב יהיה ממומש כחלק מקוד המקור של התוכנה אשר מצפין ומפענח את עצמו. מטרתו של ה-packer היא לבצע מניפולציה כזאת על קובץ ההרצה (לאחר קימפול הקוד). קיימים מספר pakcer-ים ידועים (כדוגמה upx) אשר לוקחים קבצי הרצה כקלט ומחזירים כפלט קובץ הרצה חדש ארוז. כיום ממומשים מנגנונים מובנים בתוך תוכנות אנטי-וירוס אשר יודעים להתמודד עם packer-ים מוכרים. על מנת לבצע מעקף גם למגנונים אלו נוכל לכתוב מנגנון packing עצמאי שאיננו מוכר ע"י תוכנות אנטי-וירוס.

ה-Packer מסתכל על מבנה הקובץ הבינארי, מפרסר אותו ומכווץ את ה-text section. במבנה קובץ הרצה (PE, ELF...) ה-text section הוא החלק אשר מכיל את קוד המערכת שאמור לרוץ. בנוסף הוא מוסיף לתחילת ה-text section חלק קוד נוסף אשר אחראי לפענח את ה-text section. לאחר מכן, ה-Packer משנה את נקודת ההתחלה ממנה ירוץ הקובץ (entry point, מופיע כחלק מה-header-ים של קובץ ההרצה).

בנוסף ל-text section, ה-packer יכול לבצע זאת על עוד section-ים אחרים בקובץ (לדוגמה על resource-ים, data sections ועוד).

4.1.4.5 שימוש ב-ROP gadgets

מאמרים [4] ו-[7] מציגים שיטה מעניינת נוספת להתחמקות ממנגנוני ניתוח סטטי והיא שימוש ב-ROP gadgets. ראשית לפני שנצלול לפתרון הטכני אשר המאמר מציג, ננסה להבין בקצרה מהו מנגנון ROP (Return Oriented Programming).

Return Oriented Programming או בקיצור ROP היא טכניקה להרצת קוד זדוני ע"י שימוש בקוד תמים ומוסברת בהרחבה במקור [8]. השיטה נפוצה מאחר והיא משמשת בעיקר למעקף מנגנוני הגנה שונים כגון: חתימת קוד והגנות על מרחב ההרצה בזיכרון (DEP, NX bit). השיטה מתבססת על הרעיון שנוכל לבנות קוד מערכת זדוני אשר מורכב מחלקי קוד תמימים אשר מפוזרים בזיכרון במקומות שונים.

נסתכל לדוגמה על קוד המערכת הבא:

```
xor eax,eax
inc eax
pop ebx
int 0x80
```

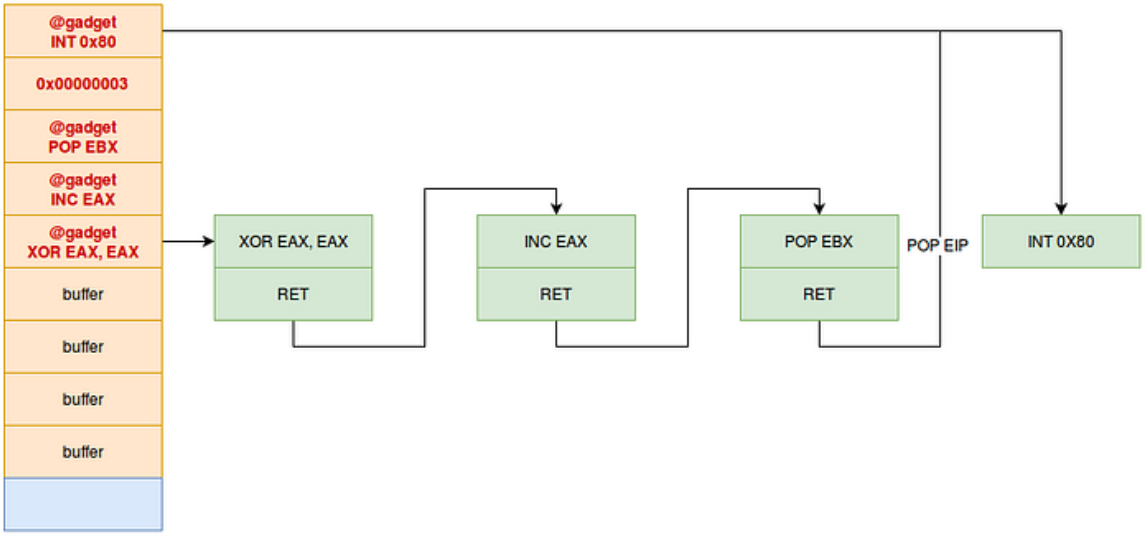
הקוד לעיל מבצע את הפעולה הפשוטה של לצאת מהתהליך הנוכחי. הפעולה int 0x80 מבצעת קריאה ל-syscall, ה-syscall שיש לבצע נקבע ע"י ערכו של הרגיסטר eax אשר במקרה שלנו מושם להיות 1 (איפוס ואז העלאה ב-1). לפי טבלת ה-syscall-ים, הערך 1 שייך ל-sys_exit אשר יוצא מהתוכנית ומחזיר את ערך ההחזרה אשר נמצא ברגיסטר ebx (בקוד שלנו ebx נקרא מתוך המחסנית ע"י פקודת pop).

הפתרון ש-ROP בא להציע הוא – ביצד נוכל להרכיב את הקוד הנ"ל מתוך חלקי קוד קיימים בזיכרון, הדרך לבצע זאת היא ע"י מציאת gadget-ים מתאימים. ROP gadget היא פיסת קוד אשר נמצאת בזיכרון במקום לגיטימי ומכילה קטע קוד אשר רלוונטי ל-shellcode שלנו בצירוף פקודת return אשר נקראת בסיום כל פונקציה. כאשר נשרשר הרבה מאוד gadget-ים במחסנית אחד לשני נוכל לייצר את ההתנהגות הבאה:

```
Gadget 1: xor eax,eax
          ret

Gadget 2: inc eax
          ret

Gadget 3: pop ebx
          ret

Gadget 4: int 0x80
```

כעת, בין כל פקודה לפקודה מתבצע return. הפקודה return קוראת את כתובת החזרה שלה מתוך המחסנית וכך נוצר מצב שהפקודות שלנו ישורשרו אחת אחרי השנייה מאחר והכתובת הבאה שיש לקפוץ אליה לאחר ביצוע ה-return היא הכתובת של ה-gadget הבא שיש להריץ.

נוכל להסתכל על איור 3 על מנת להבין יותר טוב את מצב המחסנית והקריאות:



איור 3: המחשת תהליך ROP-Chain, נלקח ממקור [8]

כפי שניתן לראות באיור 3, נשרשר למחסנית את כל הכתובות של ה-gadget-ים אחד אחרי השני ובכך נייצר התנהגות שרשרת בה כל פקודת return תקפיץ אותנו לקטע קוד אחר בזיכרון. למרות שכל פעם אנחנו "קופצים" לאזור אחר בזיכרון המצב הלוגי במערכת שקול להרצה של הפקודות באופן רציף בדומה ל-shellcode הראשון. שרשור של מספר ROP gadgets אחד אחרי השני נקרא ROP chain.

כפי שציינתי לעיל, טכניקה זו משמשת לרוב בתור שיטה לעקיפת מנגנוני הגנה של מערכת ההפעלה להרצת בינאריים בזיכרון. החוקרים במאמר [7] לקחו את הרעיון צעד אחד קדימה ומשתמשים בטכניקה זו על מנת להמיר נוזקה ל-ROP chain בכדי לבצע מעקף למנגנוני ניתוח סטטי של תוכנות אנטי-וירוס.

במאמר, החוקרים פיתחו כלי בשם ROPInjector אשר מקבל כקלט shellcode וקובץ בינארי. מטרת הפרויקט היא לייצר ROP chain מלא המדמה את ה-shellcode מתוך התוכן של הקובץ בינארי. הפרויקט עצמו מורכב מ-5 שלבים:

1. ניתוח והמרת ה-shellcode לחלקים קטנים
2. מציאת ROP gadgets עבור חלקי ה-shellcode
3. בחירת gadget-ים מתאימים מתוך ה-gadget-ים שנמצאו/השלמה של gadget-ים חסרים
4. כתיבת רצף פקודות אשר בונה את ה-ROP chain במחסנית
5. ביצוע patching לקובץ הבינארי

בשלב הראשון, הקוד מנתח את ה-shellcode שהתקבל ומנסה לפצל אותו לחלקים קטנים של קוד. מאחר ו-shellcode אינו יותר מרצף בינארי של בתים, על הקוד לבצע לו reverse engineering בזמן ריצה ולנסות לפרסר אותו לרצף של פקודות בשפת מכונה.

השלב השני, הוא מעבר על כל הקובץ הבינארי אשר התקבל בניסיון לחפש קטעי קוד אשר יכולים לשמש כ-gadget-ים מתאימים עבור קטעי הקוד שנמצאו בשלב הראשון. כפי שכבר הסברנו לעיל על ROP chain, נדרש למצוא קטעי קוד אשר בסופם מתבצע אחת מן הפעולות הבאות:

- *ret*
- *retn*
- *pop regX*
- *jmp regX*

הפקודות הנ"ל מחזירות את השליטה לכתובת לבחירתנו (ע"י ערך הקבוע במחסנית או ערך הנמצא ברגיסטר שנשלוט בו).

בשלב השלישי לאחר מציאת מבחר גדול של gadget-ים פוטנציאליים עבור כל אחד מן קטעי הקוד, נדרש למצוא את ה-gadget הטוב ביותר כך שאינו משנה מצב של register-ים אחרים ויודע להשתלב בצורה איכותית עם שאר ה-gadget-ים השונים.

יכול להיווצר מקרה בו לא קיים בבינארי gadget עבור קטע הקוד שאנחנו מחפשים. מטרתו הנוספת של השלב השלישי היא להזריק את ה-gadget-ים החסרים למקומות פנויים בקוד. בכל בינארי קיימים מקטעים אשר הינם בהרשאות הרצה אך הם ריקים, מקטעים אלו נקראים code caves. Code caves יכולים להיווצר במקרה של padding בין פונקציות שונות בקוד או כהקצאה לשימוש עתידי במקטע זה בזמן ריצה, במקרה שלנו נוכל להשתמש בהם כמקום להכניס את ה-gadget-ים החסרים ובכך "לייצר" gadget-ים לשימושנו כחלק מה-ROP chain.

לאחר סיום השלב השלישי קיים ברשותנו ROP chain מלא אשר כל מה שנשאר הוא להכניס אותו למחסנית ולקרוא לו בזמן ריצה, מטרת השלב הרביעי היא לדאוג לביצוע פעולה זו. מאחר והמחסנית נבנית ומוקצית בזמן ריצה קיימות לנו 2 אופציות לכתיבת ה-ROP chain למחסנית:

1. כתיבת ה-ROP chain למחסנית בזמן ריצה, נחבר רצף פקודות push של כתובות ה-gadget-ים שיש להריץ ונדאג שקוד זה ירוץ כחלק מהבינארי.
2. העתקת ה-ROP chain למחסנית בזמן ריצה מתוך ה-data section.

החוקרים טוענים שקיימת עדיפות להשתמש בשיטה הראשונה מאחר ויכולות להיות בעיות תאימות של encoding/decoding בתהליך ההעתקה של ה-data למחסנית. קטע הקוד האחראי לבנות את ה-ROP chain יראה מהצורה הבאה:

[1] *call build_chain*

[2] *jmp past_the_chain*

[3] *build_chain:*

[4] *push <VA of gadget N>*

[5] *....*

[6] *push <VA of gadget 1>*

[7] *ret*

[8] *past_the_chain:*

[9] *other instructions/chains*

נוכל לראות כי כאשר נקרא בזמן ריצה לפקודה [1] יתבצע תהליך של דחיפת כל ה-gadget-ים למחסנית בסדר הפוך ורק לאחר מכן קריאה לפקודת ה-return (מאחר והקריאה מתבצעת על פי ערך החזרה של הפונקציות).

לאחר הרכבת ה-ROP chain והרכבת קטע הקוד שיוצר אותו כל מה שנשאר הוא השלב האחרון. בשלב החמישי מכניסים את קטע הקוד שבונה את ה-ROP chain לבינארי ודואגים להרצת הקוד.

החוקרים בחרו לכתוב את הקוד שבונה את ה-ROP chain כחלק מה-text section של הקובץ הבינארי, אך העלו 2 דרכים שונות להרצת הקוד:

1. החלפת ה-entry point – ניתן להחליף את הכתובת המופיעה ב-NT_HEADER של הקובץ תחת השדה AddressOfEntryPoint לכתובת של הקוד. השדה הנ"ל מציין מה תהיה כתובת ה-text section אשר ממנה יש להתחיל לרוץ לאחר טעינת הבינארי לזיכרון.
2. ביצוע hook ל-ExitProcess – על מנגנון ה-hook-ים נדבר בהרחבה בהמשך (להלן בפרק "5.1.1 מהו API hooking?") אך כעת נוכל לחשוב עליו כדריסה של כתובת של פונקציה במערכת. החוקרים בחרו לדרוס את הכתובת של הפונקציה ExitProcess אשר נקראת כאשר מתבצעת סגירה של התהליך ולהחליף אותה בקריאה לקוד שבונה ומריץ את ה- ROP chain.

החוקרים בדקו את הכלי שלהם אל מול מספר קבצי הרצה פופולאריים במתארים שונים. כפי שניתן לראות באיור 4.

Table 1 List of PE files used as carriers in the experiments and various statistics (ROP entry method and reverse tcp shell)

PE	Size, kB	File version	Candidate gadgets found in PE	Gadgets injected	Gadgets used from PE
Acrobat.exe	650	19.10.20069.49826	338	78	12
AcroRd32.exe	1423	11.0.8.4	5599	67	26
cmd.exe	305	6.3.9600.16384	608	76	16
firefox.exe	439	63.0.3.6892	1634	65	28
java.exe	187	8.0.192.12	1148	76	18
nam.exe	1828	1.0a11a	3515	67	25
notepad++.exe	2783	7.6.0.0	7778	55	41
Rainmeter.exe	39	2.4.0.1678	11	83	0
wmplayer.exe	163	12.0.9600.19145	60	82	3

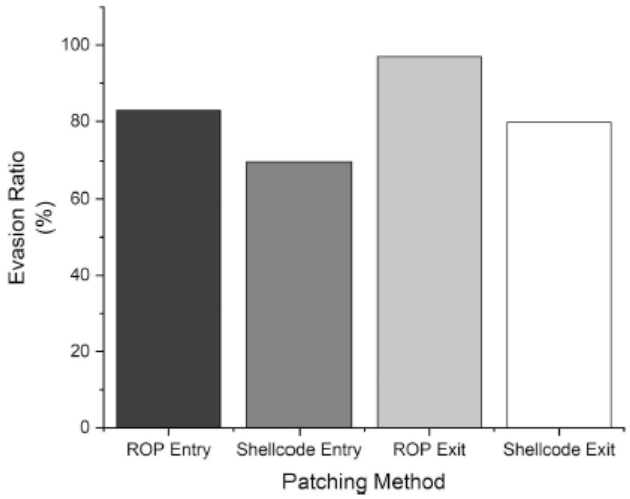
איור 4: תוכנות פופלריות שנבדקו אל מול ROPInjector, נלקח ממאמר [7]

החוקרים בדקו 4 מתארי הרצה שונים:

- הרצת הקוד שבונה את ה-ROP chain בתחילת התוכנית ע"י החלפה של ה-Entry point
- הרצת הקוד שבונה את ה-ROP chain בסיום התוכנית ע"י ביצוע hook ל-ExitProcess
- הרצת ה-shellcode ישירות ע"י החלפה של ה-Entry point
- הרצת ה-shellcode ישירות ע"י ביצוע hook ל-ExitProcess

על מנת לבדוק את תוצאות הפרויקט, החוקרים הריצו את הפרויקט על shellcode פשוט אשר יוצא חיבור reverse tcp shell אל מול כל אחד מן הבינאריים שהוצגו לעיל.

איור 5 מציג את התוצאות כאשר בדקו כל אחד מקבצי ההרצה המטופלים ע"י ROPInjector בכל אחד ממתארי ההרצה השונים אל מול מנועי ניתוח סטטי של 57 סוגי אנטי-וירוס שונים.



איור 5: תוצאות הבדיקה של המתארים השונים, נלקח ממאמר [7]

ניתן לראות שבמתאר השני שהוצג (הרצת הקוד שבונה את ה-ROP chain בסיום התוכנית ע"י ביצוע hook ל-ExitProcess), הושג בממוצע סטטוס התחמקות מאנטי-וירוסים של 98.31%.

לסיכום, המאמר מציג שיטה מתוחכמת ומאוד אפקטיבית להתחמקות ממנועי אנטי-וירוס ע"י פצ'פוץ' קובץ הרצה מוכר כך שירכיב shellcode זדוני ויריץ אותו בצורה מותממת וחשאית אשר עוקפת מנועי ניתוח סטטי של אנטי-וירוסים.

4.2 ניתוח דינאמי

4.2.1 מהו ניתוח דינאמי?

ניתוח דינאמי הוא אחד ממנגנוני ההגנה וזיהוי האיומים של תוכנות אנטי וירוס. בשונה מניתוח סטטי, מטרתו היא לחלץ מידע על התנהגות הקובץ ע"י הרצתו. בשיטה זו, ניתן לזהות תוכנה זדונית ע"י הרצה שלה בסביבה אמיתית או וירטואלית ובכך לראות איזה פעולות התוכנה מבצעת.

כפי שכלי הניתוח הסטטי שהצגנו לעיל נקראים סורקים (scanners), כלי הניתוח הדינאמי העיקריים הם emulator ו-sandbox.

- Emulator – מנגנון אשר מדמה הרצה של קובץ על מנת להשיג ולחלץ מידע על ההתנהגות שלו.
- Sandbox – סביבה מנותקת (לרוב ווירטואלית) אשר משמשת להרצת קבצים מבלי שיעשו פעילות שיכולה להשפיע על המערכת.

בשונה מהסורקים של הניתוח הסטטי אשר יתרחשו אחת לכמה זמן, הניתוח הדינאמי יתבצע מאחורי הקלעים לפני הרצת תוכנה כלשהי.

4.2.2 שיטות לניתוח דינאמי

4.2.2.1 VM Analysis

השיטה הפופולרית ביותר לניתוח דינאמי אשר מוצגת במקורות [2] ו-[5] היא הרצת הקובץ emulator בסביבת sandbox וירטואלית. בשיטה זו נעשות הפעולות הבאות:

1. יוצרים סביבה ווירטואלית
2. שומרים snapshot (תמונת מצב נוכחית של המערכת)
3. מריצים את התוכנה בסביבה הווירטואלית
4. שומרים snapshot נוסף של המערכת
5. משווים בין ה-snapshot-ים השונים ומנתחים את התוצאות

באמצעות ביצוע צעדים אלו נוכל לנתח את ההתנהגויות השונות אשר התוכנה ביצעה ובכך לדעת האם לסווג את התוכנה כזדונית או לא.

במהלך ההשוואה של ה-snapshot-ים נבדקים מספר מרכיבים עיקריים:

תהליכים – ניטור תהליכים שונים במערכת ובדיקת המאפיינים השונים שלהם לפני ואחרי הרצת התוכנה. לדוגמה כך נוכל לזהות תוכנה זדונית אשר מחולצת לזיכרון בזמן ריצה בשונה מניתוח סטטי.

מערכת הקבצים – מערכת הקבצים היא מקום אשר להרבה נוזקות תהיה התממשקות איתה בצורה כזאת או אחרת. רוב המידע של המחשב מאוחסן במערכת הקבצים ולכן נוזקות ינסו לגשת לקבצים אשר עלולים להיות רגישים/אסטרטגיים על מנת לקבל אחיזה יותר מלאה במערכת.

Registry – ה-registry הוא מאגר מרכזי של הגדרות אשר מערכות windows משתמשות בו. נרשמות שם הגדרות והעדפות כלליות על המערכת וגם תוכנות יכולות לשמור שם ערכים הרלוונטיים רק אליהן. בדומה למערכת הקבצים גם כאן נוזקות ירצו לשנות ערכים רגישים אשר

יספקו להן אחיזה יותר מלאה במערכת. ערכים כאלו לדוגמה, קבצים אשר ירוצו בעת עליית המחשב, קונפיגורציות רגישות, תוכנות ברירת מחדל ועוד.

תקשורת – לפעמים נוכל לאתר נוזקה על סמך תקשורת זדונית אשר היא מנסה לבצע. לדוגמה, תקשורת אל מול שרת צד שלישי זדוני, הרצת פניות לא לגיטימיות אשר ישויות ברשת ועוד.

4.2.3 שיטות למעקף ניתוח דינאמי

כפי שצוין לעיל, הניתוח הדינאמי קורה בסביבה וירטואלית לפני הרצה של תוכנה במערכת הוא צופן 2 אתגרים עיקריים בתוכו אשר נוזקות ינסו לנצל אותן על מנת לעקוף מנגנון זה:

1. חוויית שימוש חלקה ומהירה למשתמש – על הניתוח הדינאמי לקרות בזמן יחסית קצר על מנת לא לשבש את חוויית השימוש של המשתמש במערכת ולהקשות עליו עם זמני המתנה ארוכים. יש לזכור שמלבד זמן ההרצה של התוכנית, לוקח זמן גם להרים את ה-sandbox ועוד זמן לנתח את התוצאות.
2. ריצה בסביבה וירטואלית – מאחר והניתוח הדינאמי קורה בסביבה וירטואלית, נוזקות ינסו לשאוף לגלות את הסביבה בה הן רצות ובכך לדעת האם להפעיל את הלוגיקה הזדונית או לא.

נציג כעת מספר טכניקות שונות למעקף ניתוח דינאמי אשר מנסות לכוון אל אחת משני אתגרי הניתוח אשר הצגנו כעת.

4.2.3.1 עיכוב זמנים

נוזקות ינסו לעכב את זמן הרצת הלוגיקה הזדונית בכדי שהזמן המוקצב לניתוח הדינאמי יסתיים ובכך התוכנה תסווג כלא מזיקה ותמשיך לרוץ.

שיטות לביצוע עיכוב זמנים יכולות להיות ע"י ביצוע sleep ארוך בתחילת הקוד למשך מספר דקות או ע"י מנייה של מספר מאוד גדול.

4.2.3.2 הקצאת משאבים

נוזקות ידעו שבחלק מהניתוח הדינאמי הן יורצו ב-sandbox. Sandbox כפי שכבר הוסבר היא סביבה וירטואלית אשר מדמה את המערכת אך מנותקת ממנה. על מנת להרים sandbox, המערכת מקצה כמות משאבים כלשהי (RAM, CPU) לטובת ה-sandbox.

נוזקות ינסו לאתגר את כמות המשאבים שה-sandbox מקבל ובכך לגרום לסגירת ה-sandbox ויציאה מהניתוח הדינאמי. לדוגמה, בהינתן sandbox אשר מוקצה לו 2GB RAM, נוזקה תנסה להקצות זיכרון בגודל 4GB RAM וישר לשחרר אותו רק על מנת לבדוק האם היא רצה כחלק מ-sandbox. יש לציין שהתנהגות זו אינה מוגדרת היטב ותלויה במימוש מנגנון ה-sandbox אשר תוכנת האנטי-וירוס הגדירה.

4.2.3.3. זיהוי מאפייני VM

ברוב במקרים ה-sandbox שהאנטי-וירוס ירים יהיה תשתית וירטואלית נוכל לנסות לזהות זאת ע"י מאפיינים ספציפיים של מכונות וירטואליות. לדוגמה עבור מכונות וירטואליות של VMware קיימים מספר סימנים מזהים:

- כרטיס הרשת של המערכת יוחלף להיות עם MAC שונה כאשר ה-vendor שלו יהיה מסוג VMware inc.
- ייווצרו ערכי registry חדשים וייחודיים ל-VMware. לדוגמה נוכל לראות תחת הנתיב הבא: HKLM\System\CurrentControlSet\Services\Disk\Enum, המכיל את כל ה-disk controllers של המערכת רכיבים חדשים המכילים את המילים Virtual או VMW או Vbox.

4.2.3.4 זיהוי התנהגות שאינה מוגדרת היטב

זוהי שיטה הרבה יותר כללית אשר אומרת מתרכזת ברעיון הבא – התנהגות ה-sandbox ממומשת ע"י האנטי-וירוס וקיימים מקרה קצה אשר האנטי-וירוס מנסה לדמות בצורה כלשהי ואותם נוכל לתפוס ולזהות שמדובר ב-VM.

דוגמה: זיהוי mutex-ים ב-Comodo

דוגמה מעניינת אשר מוצגת במאמר [6] היא במקרה של comodo. Comdo הוא אנטי-וירוס אשר גם לו קיים מנגנון ניתוח דינאמי. בדומה לאנטי-וירוסים אחרים הוא מספק sandbox אשר נועד לדמות את מערכת ההפעלה, מקרה ספציפי אשר נתמקד בו הוא mutex. Mutex הוא מנגנון של מערכת ההפעלה לביצוע סנכרון בין תהליכים שונים במערכת. מאחר ומדובר בסביבה מנותקת על ה-sandbox לדמות את מערכת ההפעלה וזה אומר לנסות "לסמלץ" משאבים אשר התוכנה מבקשת במידת הצורך.

ב-comodo ממומשת התנהגות מוזרה בה כאשר נבקש מהמערכת ליצור mutex חדש ע"י הפונקציה OpenMutex נקבל אובייקט שמאותחל להיות עם הערך 0xBBBB.

במקרה הזה, נוכל לבצע קריאות ליצירת שני mutex-ים שונים ולבדוק האם שניהם שווים ל-0xBBBB. במידה וכן, נוכל להסיק שאנחנו רצים בתוך sandbox של comodo ובכך להחליט לא להריץ את הלוגיקה הזדונית.

יש לציין שזו רק דוגמה ספציפית אחת מיני רבות והרעיון הכללי הוא למצוא התנהגויות של מערכת ההפעלה אשר ה-sandbox ירצה לנסות "לסמלץ" בעצמו אך לא יעשה את זה בצורה טובה.

התנהגויות נוספות יכולות להיות: קריאות לפונקציות מערכת שלא מתועדות מספיק טוב, ניסיון יצירה של אובייקטים שמורים במערכת, ניסיון טעינה של עוד קבצים חדשים ל-sandbox ועוד.

5. זיהוי בזמן ריצה

חלקה הראשון של העבודה התרכז בעיקר בזיהוי אנומליות על קבצים שוהים (באופן מחזורי ע"י ניתוח סטטי ולפני טעינתם לזיכרון ע"י ניתוח דינאמי). לתוכנות אנטי-וירוס קיים מנגנון נוסף של זיהוי אנומליות והוא ניטור תמידי של תהליכים בזמן ריצה.

קיימות אינספור דרכים שונות לאיתור אנומליות בזמן ריצה בתלות ברמת המורכבות והמקצועיות של האנטי-וירוס בפרט ושל המערכת בכלל. לדוגמה, אנטי-וירוס המכיל מנגנון ריצה ב-kernel יוכל לתת מענה יותר רחב ולתפוס יותר אנומליות במערכת מאשר אנטי-וירוס שרץ רק במצב user.

בחלק זה, נתמקד בשתי טכניקות מעניינות אשר תוכנות אנטי-וירוס לרוב מממשות על מנת לאתר אנומליות בזמן ריצה.

5.1 API hooking

5.1.1 מהו API hooking?

כפי שציינתי לעיל, עולם הבעיה של איתור נוזקה בזמן ריצה הוא רחב מאוד ולצורך כך תוכנות אנטי-וירוס ניסו לאפיין אנליזות ושיטות זיהוי גנריות ככל האפשר. API hooking היא שיטה המתבססת על רעיון מאוד פשוט והוא שבסופו של דבר קוד יהיה חייב לרוץ במערכת. כלומר, תמיד נוכל לבצע סריקות וחיפוש אקטיבי של חלקים בזיכרון שנראים חשודים אך קיים צוואר בקבוק מאוד ברור והוא זמן הריצה של הקוד במעבד.

נוכל לבצע ניטור על קוד האמור לרוץ כעת במערכת ובכך נוכל לאפיין האם מדובר בקוד זדוני או לא.

מאמר [9] מציג בהרחבה כיצד עובד מנגנון API hooking וכיצד ניתן לזהות אותו. אך על מנת להבין את הטכניקה לעומק נסקור בקצרה שני מושגים ונראה כיצד הם מהווים חלק מהותי מתוך השיטה לזיהוי אנומליות בזמן ריצה.

WinAPI

WinAPI, או Windows API הוא ממשק תוכנה המיוחצן ע"י מערכת ההפעלה המאפשר גישה לביצוע פעולות במערכת ההפעלה. רוב הפונקציות המוכרות לנו כיום מממשות מעין מעטפת המשתמשת בפונקציות אלו אשר נחשבות "נמוכות" יותר וקרובות למערכת ההפעלה. מאחר ורוב הפונקציות המומשות בספריות כאלו ואחרות יצטרכו לקרוא ל-WinAPI שם יתבצע המיקוד של תוכנות אנטי-וירוס.

Hooking

Hooking הוא מנגנון המאפשר הרצה של קוד לבחירתנו לפני הרצה של קוד אחר במערכת. קיימות מספר שיטות לביצוע hooking כאשר הנפוצות הן: inline hooking ו-IAT hooking.

IAT Hooking

IAT Hooking, היא שיטה בה נחליף כתובת של פונקציה בזיכרון בכתובת אחרת. IAT או בשמו המלא Import Address Table הוא מבנה נתונים הנקרא מהקובץ הבינארי ונשמר בזיכרון

התהליך אשר מחזיק מיפוי של שמות כל הפונקציות בהן התהליך נדרש לבצע שימוש אל מול הכתובת שלהן בזיכרון.

ביצוע hooking בשיטה זו משמעו פשוט להחליף את כתובת הפונקציה בזמן ריצה לכתובת של פונקציה אחרת אשר מבצעת לוגיקה כלשהי ולאחר מכן קוראת לפונקציה המקורית.

Inline Hooking

Inline hooking, היא שיטה אשר בה נדרוס את הבתים הראשונים בתחילת פונקציה בה נרצה להתערב ונחליף אותם בקריאה ללוגיקה שלנו הטעונה במקום אחר בזיכרון. לאחר סיום הלוגיקה, נחזיר את השליטה חזרה לפונקציה המקורית.

ניתן לראות את איור 6 המדגים את תהליך העבודה של inline hooking.

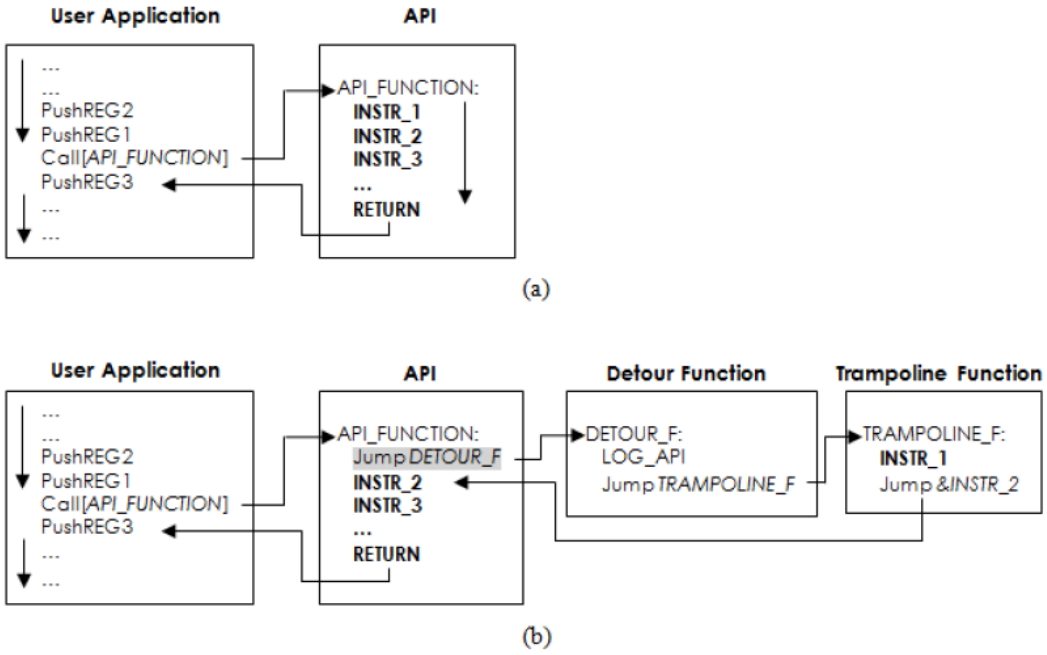


Figure 4. (a) Normal API call code execution flow, and
(b) Inline hook in action

איור 6: מבנה Inline Hooking בזיכרון, נלקח ממאמר [9]

נוכל לראות שבתחילת הקוד אנחנו קופצים למקום אחר בזיכרון, מריצים שם קוד לבחירתנו ואז חוזרים בחזרה לקוד המקורי.

לאחר שהבנו מהו WinAPI וביצד עובד מנגנון hooking נוכל להבין את הרעיון מאחורי API hooking.

בתחילת הרצה של כל תהליך, תוכנות אנטי-וירוס יטענו ספרייה משלהם לזיכרון התהליך אשר מכילה פונקציות בדיקה שונות. הרעיון הוא שבעת כל קריאה לפונקציה של WinAPI תתבצע קריאה לפונקציית הבדיקה של האנטי-וירוס והפונקציה תוודא שהפעולה אותה הפונקציה התבקשה לבצע הינה לגיטימית, במידה וכן פונקציית הבדיקה תעביר את השליטה לפונקציה המקורית. במידה ונמצא

שהלוגיקה העתידה להתבצע אינה לגיטימית על פי שיקול הדעת של פונקציית הבדיקה, הקריאה תחסם.

על מנת להחליף כל קריאה לפונקציית WinAPI עם קריאה לפונקציית בדיקה של האנטי-וירוס, האנטי-וירוס יחליף כל קריאה לפונקציית WinAPI לקריאה לפונקציית הבדיקה בספרייה שהוא טען. רוב תוכנות האנטי-וירוס ישתמשו לשם כך במנגנון Inline hooking אך קיימים באלה שיבצעו גם IAT hooking. נוכל להסתכל בדוגמה הבאה של אנטי-וירוס מסוג bitdefender על מנת להבין כיצד נראית טכניקה זו.

5.1.2 Case study: Bitdefender

נסתכל על אנטי-וירוס Bitdefender אשר מממש את הטכניקה שהצגנו לעיל.

הפונקציה CreateRemoteThreadEx היא פונקציית WinAPI אשר נמצאת בשימוש רחב של תוקפים מאחר והיא מאפשר להריץ קוד בתהליך מרוחק. מקור מספר [10] מתאר כיצד אנטי-וירוס Bitdefender מממש הגנה מפני קריאות זדוניות לפונקציה זו.

נוכל לשים לב שהאנטי-וירוס טוען את עצמו לזיכרון ביצירה של תהליך חדש, כמתואר באיור 7:

Base	Module	Party	Path
0000000000400000	injector2.exe	User	C:\Users\Administrator\Desktop\injector2.exe
00007FF8DCBA0000	atcuf64.dll	User	C:\Program Files\Bitdefender\Bitdefender Security\atcuf\264738995640000000\atcuf64.dll
00007FF8FD240000	apphelp.dll	System	C:\Windows\System32\apphelp.dll
00007FF8FFCA0000	kernelbase.dll	System	C:\Windows\System32\KernelBase.dll
00007FF9002E0000	msvcrt.dll	System	C:\Windows\System32\msvcrt.dll

איור 7: איתור ספריית Bitdefender בזיכרון תהליך, נלקח ממקור [10]

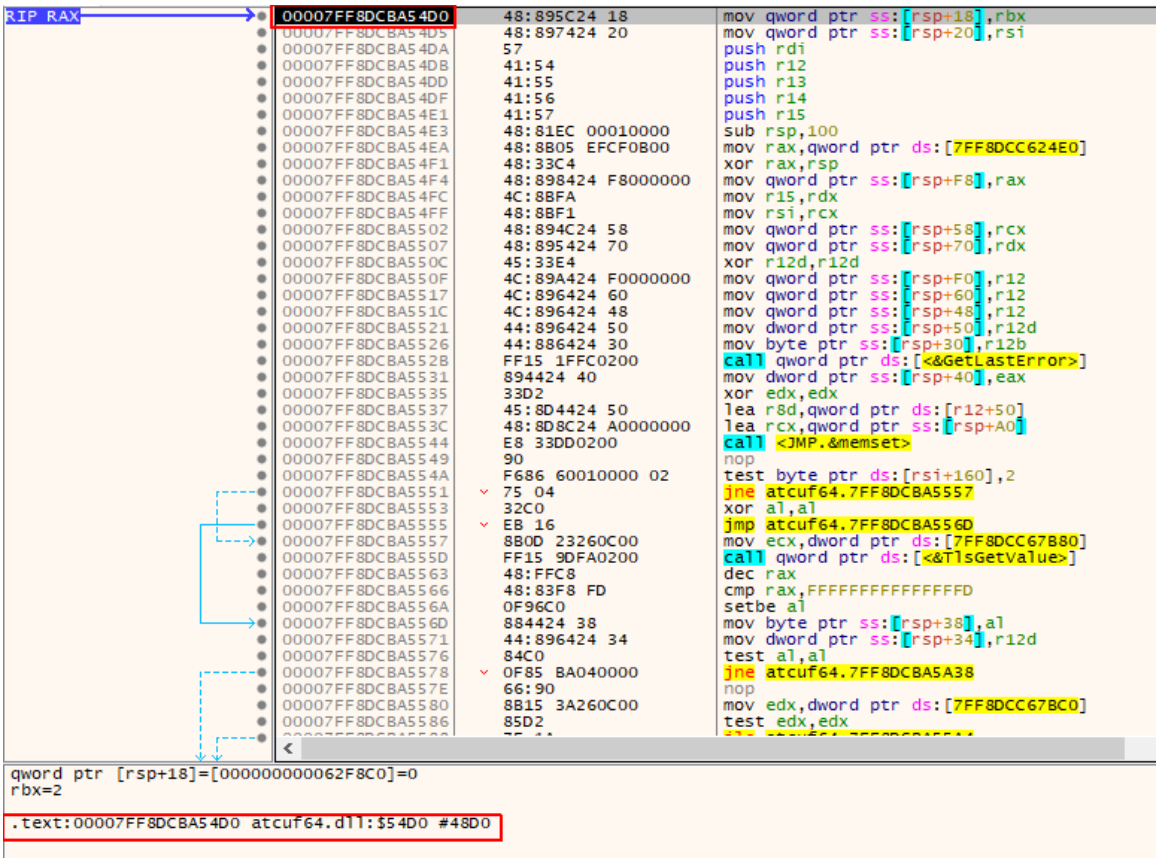
באיור 7 ניתן לראות שהספרייה atcuf64.dll שייכת לאנטי-וירוס ונטענת לזיכרון לצד שאר הספריות הפונקציונליות. נסתכל כעת על הפונקציה CreateRemoteThreadEx בזיכרון אשר טעונה בספריה Kernel32.dll

```
00007FF8FFCD1C70   ^ E9 0BE95382      jmp 7FF882210580                    CreateRemoteThreadEx
00007FF8FFCD1C75     41:55            push r13
00007FF8FFCD1C77     41:56            push r14
00007FF8FFCD1C79     41:57            push r15
00007FF8FFCD1C7B     48:81EC 50050000 sub rsp,550
00007FF8FFCD1C82     48:8B05 E7812500 mov rax,qword ptr ds:[7FF8FFF29E70]  rax:EntryPoint
00007FF8FFCD1C89     48:33C4          xor rax,rsp                         rax:EntryPoint
```

איור 8: מימוש קפיצה בתחילת פונקציה בזיכרון, נלקח ממקור [10]

באיור 8 נוכל לראות שהפעולה הראשונה בתחילת הפונקציה מבצעת קפיצה לקטע קוד אחר בזיכרון.

לאחר ביצוע הקפיצה והמשך ה-flow בסופו של דבר נראה כי הקפיצה מובילה לספרייה atcuf64.dll שכפי שראינו מקודם שייכת לאנטי-וירוס של Bitdefender, כמתואר באיור 9.



איור 9 : קפיצה לקוד Bitdefender ומימוש לוגיקה פנימית, נלקח ממקור [10]

5.1.3 מעקף API hooking

הדרך הפשוטה ביותר להתחמק ממנגנון זה היא ביצוע אותה הדרך בה המנגנון הותקן מלבתחילה. נחלק את הפתרון ל2 מקרים: תיקון עבור IAT hooking ותיקון עבור inline hooking.

IAT unhooking

עבור IAT hooking נדרש למצוא מחדש את הכתובת האמיתית של הפונקציה ואז להחליף את הערך המתאים ב-Import Address Table בזיכרון התהליך שלנו. כשם ש-IAT הוא מבנה המחזיק את המיפוי של הפונקציות אשר נדרש לייבא אותם, קיים מבנה בשם EAT (Export Address Table) המחזיק את כל הפונקציות אשר בינארי כלשהו (לרוב DLL) מייחצן לשימוש.

על מנת למצוא את הכתובת המקורית של פונקציה כלשהי נוכל ללכת לכתובת של ה-DLL אשר מחזיק את הפונקציה בזיכרון, לפרסר אותה מתוך ה-EAT שלו ולחשב מה הכתובת האמיתית של הפונקציה. לאחר שחישבנו את הכתובת של הפונקציה עלינו לערוך את ה-IAT ולהחליף חזרה את הכתובת של הפונקציה לכתובת המקורית.

נסכם בקצרה את סדר פעולות שהסברנו עד כה:

1. חיפוש ה-DLL המכיל את הפונקציה הרצויה בזיכרון.
2. מעבר על ה-EAT של ה-DLL וחיפוש אחר הפונקציה הרצויה.
3. חישוב כתובת הפונקציה המורכב מהערך ב-EAT + ה-entry point של ה-DLL.
4. שינוי הרשאות ה-IAT להרשאות כתיבה.

5. החלפת כתובת הפונקציה בזיכרון לכתובת המקורית שחישבנו.
6. שינוי הרשאות ה-IAT חזרה והסרת הרשאות הכתיבה.

Inline unhooking

עבור inline hooking נדרש למצוא את הכתובת של הפונקציה בזיכרון ולהסיר את הבתים הראשונים של הפונקציה שמכילים את הקפיצה לספריית הבדיקה של האנטי-וירוס.

הבתים הראשונים בתחילת כל פונקציית WinAPI יהיו קבועים ויכילו פקודות של שמירת המצב הקיים, הערך הבינארי של ה-OPCODE-ים יכול להשתנות בין מעבד אחד לאחר אך הרעיון בבסיסו הוא זהה.

דוגמה לקוד prolog של פונקציה:

```
mov   [RSP + 8], RCX
push  R15
push  R14
push  R13
sub   RSP, fixed-allocation-size
lea   R13, 128[RSP]
```

כל מה שעלינו לעשות הוא להחליף את הבתים הראשונים אשר מבצעים את הקפיצה לקוד הבדיקה עם הבתים הראשונים אשר אמורים להיות קיימים בזיכרון.

נסכם בקצרה את סדר פעולות שהסברנו עד כה:

1. חיפוש הכתובת של הפונקציה הרצויה לעריכה.
2. שינוי הרשאות הספרייה הרלוונטית להרשאות עריכה.
3. החלפת התחילית של הפונקציה הרצויה עם ערך hardcoded.
4. שינוי הרשאות הספרייה חזרה להרשאות המקוריות.

במידה ולא נרצה לבצע זאת בצורה hardcoded, נוכל להשתמש בשיטה יותר מתוחכמת והיא לחפש את הקובץ על הדיסק ממנו נטענה הפונקציה (קבצי ה-DLL של WinAPI ימצאו לרוב בנתיב C:\Windows\System32), לפרסר את ה-EAT ולחשב את כתובת הפונקציה שלנו (בדומה לתהליך שעשינו ב-unhooking IAT). לאחר שמצאנו את כתובת הפונקציה שלנו, כל מה שנשאר הוא לקרוא את הבתים הראשונים של הפונקציה מהדיסק ולהחליף אותם עם הבתים בזיכרון.

במקרה כזה סדר הפעולות יראה אחרת:

1. חיפוש הספרייה אליה הפונקציה שייכת בדיסק.
2. מעבר על ה-EAT של ה-DLL וחיפוש אחר הפונקציה הרצויה.
3. חישוב כתובת הפונקציה המורכב מהערך ב-EAT + ה-entry point של ה-DLL.
4. קריאה של הבתים הראשונים של הפונקציה מתוך הדיסק.
5. חיפוש הכתובת של הפונקציה הרצויה לעריכה.
6. שינוי הרשאות הספרייה הרלוונטית להרשאות עריכה.
7. החלפת התחילית של הפונקציה הרצויה עם התחילית שחישבנו מתוך הדיסק.
8. שינוי הרשאות הספרייה חזרה להרשאות המקוריות.

AMSI 5.2

5.2.1 מהו AMSI?

Antimalware Scan Interface או בקיצור AMSI הוא מנגנון אשר הוצג לראשונה בשנת 2015 ע"י מערכת ההפעלה Windows ומספק יכולות זיהוי נוזקות. תוכנות אנטי-וירוס רבות משתמשות במנגנון זה על מנת לסרוק סקריפטים במטרה לאיתור סקריפטים זדוניים. AMSI יודע לנתח קבצי סקריפט על הדיסק בנוסף לסקריפט אשר רץ ישירות בזיכרון.

כיום, AMSI תומך בסריקת הרכיבים הבאים:

- PowerShell
- בקשות User Account Control – UAC
- JavaScript
- VBScript
- .NET Framework
- WMI
- Windows Script Host (wscript.exe, cscript.exe)
- Office VBA macros

5.2.2 כיצד פועל AMSI?

מאמר מספר [1] מציג בהרחבה כיצד פועל מנגנון AMSI. המנגנון ממומש כ-AMSI.dll אשר הוא קובץ DLL המספק ממשק בין תהליכים אל מול תוכנות אנטי-וירוס. בעת יצירת תהליך ה-DLL נטען לזיכרון התהליך ואחראי על סריקת קוד שהתהליך מנסה להפעיל. כאשר התהליך מנסה להריץ קוד בלשהו מתוך הזיכרון שלו, ה-DLL מיירט את הקוד ושולח אותו לאנטי-וירוס אשר באחריותו לנתח את הקוד ולהסיק האם מדובר בקוד זדוני או לא.

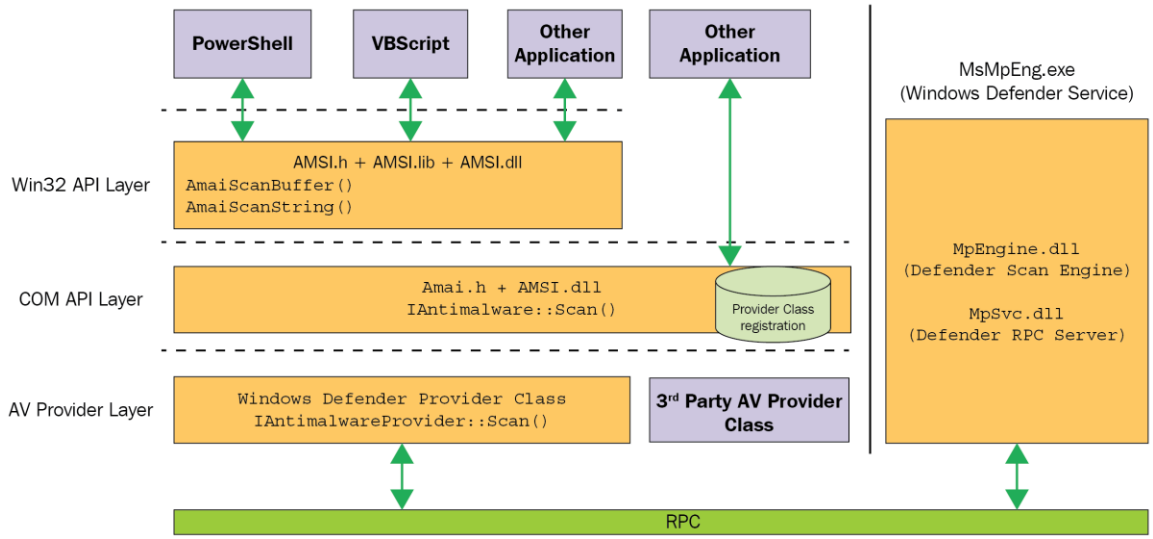
מערכת ההפעלה Windows מספקת את הפלטפורמה לסריקת מרחב זיכרון לצורך איתור סקריפט זדוני בתוכו. תוכנות אנטי-וירוס לרוב יתממשקו עם מנגנון זה כאשר תוכנת האנטי-וירוס תשלב בתוכה קריאות ל-AMSI.dll לצורך איתור סקריפטים זדוניים.

AMSI מספק שתי פונקציות API מרכזיות בכדי לסרוק מרחבי זיכרון לצורך איתור של אנומליות:

- AmsiScanBuffer
- AmsiScanString

כאשר תוכנת אנטי-וירוס מבקשת סריקה של מקטע זיכרון כלשהו ע"י פונקציות ה-API הנ"ל, הבקשה תשלח ל-DLL של AMSI אשר ינסה לסרוק את מרחב הזיכרון בהתאם לרכיבים הנתמכים שצוינו לעיל. כל אחד מן הסורקים הייעודיים אשר AMSI משתמש בו נקרא provider ויכול להיות מסופק ע"י AMSI עצמו או ע"י תוכנת צד-שלישי. ע"י כך ש-AMSI תומך בהוספה של provider-ים והרחבה של הרכיבים אשר הוא יודע, לסרוק כל תוכנת אנטי-וירוס יכולה להוסיף מנגנוני סריקה ל-AMSI כרצונה.

איור 10 ממחיש כיצד עובד תהליך בקשת סריקה ל-AMSI:



איור 10: AMSI Internals, נלקח ממקור [1]

נוכל לראות שבמידה ונבקש סריקה עבור רכיבים מוכרים ונתמכים (בגון PowerShell ו-VBScript) הקריאה תועבר דרך AMSI.dll אל ה-Provider המסופק לה ע"י Windows Defender. לאחר קבלת הבקשה ל-provider מתבצע עיבוד של המידע ומתבצעת סריקה של הקוד ע"י מנגנון הסריקה של Windows Defender (MpEngine.dll) מעל בקשת RPC.

לאחר הסריקה ע"י Windows Defender, הבקשה חוזרת אל פונקציות הקריאה של AMSI ובהתאם לתוצאת הסריקה, הפונקציה מחזירה אחת מבין התוצאות האפשריות:

- AMSI_RESULT_CLEAN
- AMSI_RESULT_NOT_DETECTED
- AMSI_RESULT_BLOCKED_BY_ADMIN_START
- AMSI_RESULT_BLOCKED_BY_ADMIN_END
- AMSI_RESULT_DETECTED

בהתאם לערך החזרה מפונקציות ה-API, מנוע האנטי-וירוס יודע להסיק האם מדובר בקוד זדוני או לא.

במידה ומדובר ברכיב נוסף שאיננו נכלל ברשימת הרכיבים הנתמכים, נוכל לרשום אותו ישירות ל-provider חיצוני המסופק ע"י צד שלישי (במקרה שלנו תוכנת אנטי-וירוס) אשר יהיה אחראי לבצע את הסריקה בעצמו ולהחזיר את ערך החזרה המתאים חזרה לפונקציית ה-API.

כפי שניתן לראות, AMSI הוא כלי חזק אשר מאפשר לתוכנות אנטי-וירוס לנצל סריקות של סקריפטים בזמן ריצה מעל Windows Defender בצורה קלה ויעילה. כמו כן, המנגנון נועד להרחבה ובנוי לכך שתוכנות אנטי-וירוס יוסיפו provider-ים משלהם בכדי להרחיב את אפשרויות הסריקה.

5.2.3 שיטות למעקף AMSI

מאז ההצגה של AMSI לראשונה ב-2015, פורסמו מספר רב של שיטות לעקוף את המנגנון. עם הזמן המנגנון השתכלל וחלק מהשיטות נעשו כבר לא רלוונטיות. נציג בקצרה מספר שיטות המוצגות במקור [11] אשר משמשות נוזקות למעקף של מנגנון AMSI.

5.2.3.1 Code Manipulation

השיטה הנאיבית ביותר אשר לא דורשת התערבות במנגנון היא ניסיון מעקף של ה-provider אשר מבצע את הסריקה. המטרה היא לנסות להתאים את הקוד עצמו כך שה-provider יטעה בסריקה. נוכל לבצע זאת ע"י השיטות הבאות:

- פיצול הקוד למספר מחרוזות ולצרף אותם בזמן ריצה
- הצפנה של הקוד
- ביצוע obfuscation
- קידוד (base64 לדוגמה)

שיטה זו איננה הרמטית ותלויה במימוש של ה-provider הקיים, דבר שלא תמיד נוכל לצפות מראש.

5.2.3.2 Unchain Providers

שיטות מתקדמות יותר דורשות התערבות במנגנון AMSI עצמו על מנת לספק פיתרון יותר הרמטי מאשר ביצוע מניפולציות לקוד.

אחת מהשיטות האלו מתמקדת ב-provider-ים של המנגנון, בניסיון למחוק את כל ה-provider-ים הקיימים מזיכרון התהליך כך שלא יהיו שום סורקים קיימים.

בעת יצירת תהליך וטעינה של AMSI.dll לזיכרון, נוצר אובייקט מרכזי בשם IAntimalware. אובייקט זה מהווה ממשק בין הפונקציה AmsiScanBuffer המהווה צוואר בקבוק אל מול כל ה-provider-ים הקיימים וטעונים לזיכרון. באתחול האובייקט IAntimalware, נבנית רשימה של כל ה-provider-ים בזיכרון ונשמרת כאיבר של IAntimalware. בכל קריאה לפונקציה AmsiScanBuffer, האובייקט IAntimalware עובר על כל ה-provider-ים ברשימה שהוא מחזיק וקורא לפונקציית הסריקה של כל אחד מהם.

ה-provider-ים עצמם שמורים בערך ייעודי ב-registry ונטענים בזמן ריצה לזיכרון. כאשר כל provider מוגדר כאובייקט מסוג class ומוחזק בתוך DLL אשר הנתיב שלו שמור ב-registry. ביצירה של אובייקט מסוג IAntimalware נטענים ה-DLL-ים מהנתיבים השמורים ב-registry וע"י קריאה לפונקציה DllGetClassObject, נטען ה-class של ה-provider מתוך כל ה-DLL.

הרעיון מאחורי המעקף הוא פשוט: לאתר את הכתובות של כל ה-provider-ים הטעונים בזיכרון, לערוך להם את הפונקציה DllGetClassObject ולאתחל את רשימת ה-provider-ים מחדש. כעת, כאשר ינסו לקרוא את ה-provider מתוך ה-DLL המתאים הפונקציה תיכשל וה-provider לא יירשם.

סדר הפעולות ייראה כך:

1. קריאה של הנתיב registry הבא: HKLM:\SOFTWARE\Microsoft\AMSI\Providers. נתיב זה מכיל את כל ה-class-ים המתאימים של ה-provider-ים שיש לטעון.
2. עבור כל אחד מן ה-class-ים שנמצאו:

a. נחפש את הכתובת של ה-class ב-registry על פי הנתיב הבא: HKLM:\Software\Classes\CLSID\<class id>\InprocServer32

b. נבצע טעינה של ה-DLL, מאחר והוא כבר טעון בזיכרון נקבל את ה-handle אליו.
c. נחפש את הכתובת של הפונקציה DllGetClassObject.
d. נערוך את הפונקציה בזיכרון כך שתחזיר שגיאה.

3. לאחר שערכנו את כל ה-provider-ים בזיכרון, נקרא לפונקציה AmsiUninitialize שסוגרת את אובייקט ה-IAntimalware הקיים.
4. כעת, בניסיון יצירת אובייקט IAntimalware חדש, הוא לא יצליח לאתחל אף אחד מן ה-provider-ים והרשימה תהיה ריקה.

Memory Patching 5.2.3.3

שיטה זו דומה מאוד ברעיונה לשיטה הקודמת וממקדת את הפעילות שלה על ה-provider-ים. אך בשונה מהשיטה הקודמת במקום לנסות לאתחל מחדש את רשימת ה-provider-ים כרשימה ריקה, היא מנסה לערוך את הרשימה הנוכחית.

כפי שכבר צוין לעיל, לכל provider קיימת פונקציית סריקה אשר האובייקט IAntimalware קורא לה ובוחן את ערך החזרה שלה. במידה וחזר הערך AMSI_RESULT_CLEAN, הסריקה זיכתה את הקוד שהועבר אליה ומגדירה אותו כבטוח להרצה.

המעקף עובד בצורה הבאה –

1. נאתר את האובייקט IAntimalware בזיכרון.
2. נעבור על רשימת ה-provider-ים אשר IAntimalware מכיל.
3. עבור כל provider:
   a. נאתר את כתובת הפונקציה Scan, אשר אחראית על הסריקה.
   b. נערוך את תוכן הפונקציה להחזיר תמיד AMSI_RESULT_CLEAN.

על מנת לאתר את האובייקט IAntimalware, נצטרך ליצור AMSI Context חדש. ביצירה של context חדש נקבל מצביע לאובייקט IAntimalware אשר כבר קיים בזיכרון. כעת כשיש לנו את הכתובת של IAntimalware, כל מה שנצטרך זה לעבור בלולאה על רשימת ה-provider-ים שלו ולהחליף השורה הראשונה של פונקציית הסריקה שלו בפקודות:

mov eax, 0x0

ret

כך הפונקציה תסיים ישר את ריצתה מבלי לבצע את לוגיקת הסריקה המקורית ותחזיר ברגיסטר eax את הערך 0 (הערך שאליו שווה AMSI_RESULT_CLEAN).

6. פרויקט מעשי: BabiAntiVirus

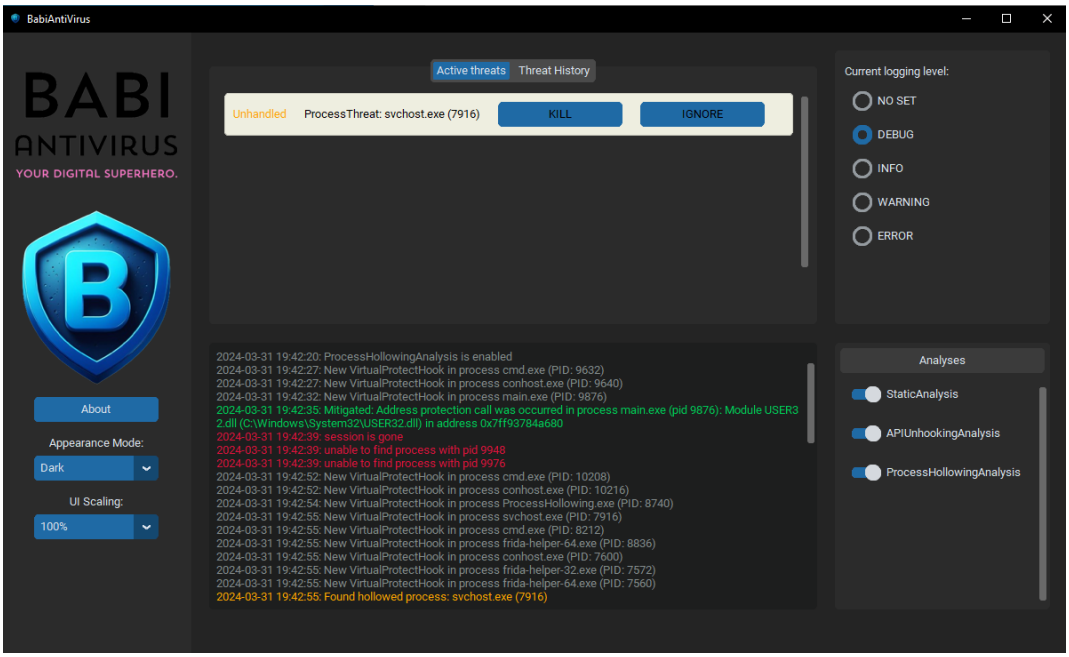
6.1 רקע

בפרויקט מימשתי מערכת אנטי-וירוס אשר מגיבה לאירועים במערכת ומכילה אנליזות לניטור וסיכול איומים אשר מומשו באופן עצמאי. הפרויקט נכתב תוך כדי השקעה בעיצוב וארכיטקטורה חזקים אשר מאפשרים הרחבה של הפרויקט באופן קל והוספת אנליזות.

הפרויקט מורכב מ-3 חלקים מרכזיים:

1. **Monitors** – אחראים לנטר ולהאזין לאירועים שונים שקורים במחשב ולהגיב אליהם בהתאם. כל monitor אחראי על חלק מסוים במחשב וכל אנליזה חייבת להיות מופעלת דרך אחד מן המוניטורים הקיימים.
   דוגמאות ל-monitors: מערכת קבצים, תהליכים.
2. **Analyses** – בדיקות אשר אחראיות למצוא אנומליות ואירועים חריגים במחשב. כל אנליזה מכילה פונקציית בדיקה ו-monitor. ה-monitor ירוץ וימתין לאירוע חדש, כאשר אירוע חדש יתרחש ה-monitor יפעיל את פונקציית הבדיקה המתאימה. על פונקציית הבדיקה לסווג את ההתנהגות כזדונית או לגיטימית, במידה ובמדובר בהתנהגות זדונית עליה ליצור אובייקט חדש מסוג Threat.
   דוגמאות ל-analyses: API Unhooking, Static Analysis, Process Hollowing
3. **Threats** – אובייקטים האחראים לייצג איומים במערכת. כאשר אנליזה מזהה התנהגות זדונית, עליה ליצור אובייקט Threat חדש. ביצירת Threat חדש, תוקפץ למשתמש התראה על איום חדש בצירוף דיאלוג להמשך הטיפול באיום. על כל אובייקט Threat לממש 2 התנהגויות לטיפול באיומים – התעלמות מהאיום וטיפול באיום.
   דוגמאות ל-threats: איום מסוג קובץ, איום מסוג תהליך.

איור 11 מציג תצלום מסך של המערכת כאשר היא מזהה איום מסוג תהליך. על המשתמש להחליט כעת האם להתעלם מהאיום או לטפל בו.



איור 11 : BabiAntiVirus הצגת איום למשתמש להמשך טיפול

6.2 מימוש הפרויקט

לפרויקט 3 מטרות עיקריות:

- מימוש תשתית אנליזות נוחה אשר ניתנת להרחבה בקלות.
- מימוש אנליזות איכותיות אשר מאתרות נוזקות.
- מימוש ממשקי גרפי נוח וברור למשתמש.

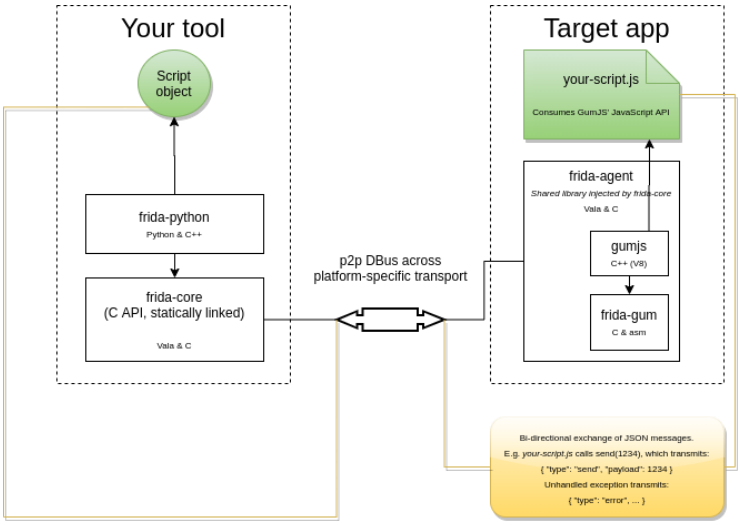
מסיבות אלו, בחרתי לממש את האנטי-וירוס באופן user-mode-י לחלוטין מבלי שום מעורבות של רכיב קרנלי כלשהו.

הפרויקט נכתב בסביבת פיתוח Python תוך כדי שילוב מספר טכנולוגיות.

6.2.1 Frida

Frida היא toolkit אשר מאפשר להזריק קוד javascript לתהליכים שונים במערכת תוך כדי התממשקות עם API פייתוני. Frida היא ספריה אשר פופולרית מאוד בקרב reverse engineers מאחר והיא מאפשרת שליטה מוחלטת בתהליך במצב attach.

בנוסף, ל-Frida קיימות מספר רב של יכולות המגיעות כחלק מה-toolkit (Frida-ps, Frida-cli, Frida-trace ועוד). ראה איור 12 המדגים את מנגנון הפעולה של Frida.



איור 12 : מבנה פעולה Frida

בעת ביצוע attach לתהליך אחר, אנחנו טוענים DLL בשם Frida-agent המשמש כמנוע להרצת הקוד JavaScript אשר אנחנו רוצים להריץ באותו התהליך.

לאחר מכן מוזרק קוד ה-JavaScript בהתאם ללוגיקה אשר בחרנו לממש. קיים ממשק שליחת הודעות בין הקוד המוזרק לבין התוכנה המזריקה (התהליך שלנו). ממשק זה ממומש ע"י ערוץ תקשורת בין 2 התהליכים המאפשר שליחה של הודעות JSON ביניהם.

כפי שמתואר באיור 12, התהליך שלנו המשתמש בAPI של Frida-python קורא מאחורי הקלעים ל-Frida-core אשר ממומשת ב-C ומאפשרת הזרקה של קוד לתהליכים שונים. האיור מציג בפירוט כיצד התהליך הראשי של Frida יודע לתקשר עם תהליך אחר אשר הוא מוזרק אליו מעל pipe-ים של מערכת ההפעלה וע"י כך נוכל לתקשר עם הקוד המוזרק אשר החדרנו בתהליך החיצוני.

עיקר השימוש ב-Frida היה כאשר נדרש לכתוב monitor שיודע להריץ קוד מוזרק בכל תהליך חדש. לעיתים נדרש לבצע בחינה על מקטעי זיכרון של תהליך ואף לשנות חלק מהם בהתבסס על לוגיקה כלשהי אשר ממומשת באנליזה. לצורך כך, שימוש ב-Frida הוא האופציה היעילה ביותר.

6.2.2 PythonForWindows

PythonForWindows היא ספרייה פייתונית אשר משמשת לביצוע אינטראקציה עם מערכת ההפעלה Windows. הספרייה מאפשרת לגשת למבנים פנימיים של Windows, להציג מידע מורחב על אובייקטים של מערכת ההפעלה, ביצוע פעולות WinAPI במימוש מובנה של Python, ביצוע hooking ועוד יכולות רבות.

6.2.3 CustomTkinter

CustomTkinter היא ספרייה פייתונית המאפשרת שילוב יכולות GUI מתקדמות בקוד Python. החלק הוויזואלי של הפרויקט נכתב תוך כדי שימוש בספרייה זו.

6.2.4 Git

מערכות ניהול גרסאות המסייעת למפתחים בניהול קוד. הפרויקט נכתב בצורה מסודרת כאשר לכל חלק branch נפרד, ועל מנת להכניס קוד למוצר הסופי נדרש לבצע מיזוג בין ה-branch-ים השונים.

6.3 מסקנות הפרויקט

לאחר מימוש הפרויקט, הפרויקט נבדק אל מול מגוון נוזקות (חלקם מוכרות וחלקם לא מוכרות) בניסיון להבין את איכות האנליזות שנכתבו.

להלן שלבי בדיקת של הפרויקט אל מול כל אחת מן הנוזקות שנאספו:

1. נריץ את הנוזקה בסביבת הבדיקה ללא שום הגבלות (אנטי-וירוס כבוי).
2. נדליק את האנטי-וירוס ונפעיל אחת מן האנליזות הקיימות.
3. נריץ שוב את הנוזקה כאשר האנליזה דלוקה.
4. נבצע זאת עבור כל אחת מן האנליזות הקיימות בפרויקט.
5. נשווה את התוצאות של הרצה ללא שום אנליזה אל מול הרצה כאשר כל אחת מן האנליזות דלוקות.

האנליזות שמומשו ונבדקו בפרויקט הן: static analysis, process hollowing, API unhooking.

האנליזות אשר הוביחו את עצמן הן: static analysis ו-process hollowing, מאחר והן זיהו את הנוזקות אשר היו מיועדות אליהן.

לעומת זאת, האנליזה של API Unhooking, לעיתים עבדה ולעיתים לא הצליחה לזהות את האיום בזמן. פספוסים אלו התבצעו מאחר והמנגנון של Frida לא תמיד הצליח להזריק אליהם את הקוד שלנו לפני הקריאות הזדוניות.

נקודה זו חשובה מאוד מאחר והיא מחזקת את ההבדל אשר עמדנו עליו בפרק של זיהוי בזמן ריצה בין עבודה ב-User mode לבין עבודה ב-Kernel mode. כאשר השיטה שנבחרה במימוש הפרויקט תצטרך להתמודד אל מול race conditions כאלו באופן קבוע.

7. סיכום

במהלך העבודה על נושא שיטות המעקף למנגנוני אנטי-וירוסים, נחשפתי לעולם מורכב ודינאמי שמאתגר את יכולות האבטחה הקיימות. התהליך כלל מחקר מעמיק על טכניקות שונות שמפתחי נוזקות משתמשים בהן בכדי לעקוף רכיבי הגנה שונים, ובמיוחד את ההתפתחויות האחרונות בתחום.

העבודה כללה:

- סקירה של מנגנוני אנטי-וירוסים והעקרונות שעליהם הם מבוססים.
- סקירה של סוגים שונים של מנגנונים לאיתור נוזקות.
- ניתוח שיטות המעקף הנפוצות של אותם מנגנונים.
- הצגת הפרויקט המעשי שנעשה בנושא, המממש מערכת אנטי-וירוס על שלל חלקיה.
- הצגת מסקנות הפרויקט המעשי.

כפי שכבר ציינתי בתחילת העבודה, עולם אבטחת המידע תמיד מתקדם ויוצר מעין משחק אינסופי של חתול ועכבר בין התוקפים למגנים. המטרה המרכזית של מוצרי הגנה היא לאתר נוזקות באופן גנרי ככל הניתן מבלי לפגום בפעילות המשתמש. לעומת זאת, מטרת התוקף היא ליצור את הנוזקה החשאית ביותר אל מול כל מוצר הגנה שקיים מאחר ותוקף לא תמיד יידע איזה מוצר הגנה הוא יפגוש.

בפן האישי ,העבודה העמיקה את הבנתי בתחומי אבטחת המידע והסייבר, והחלה עלי השפעה רבה בהבנת חשיבות האבטחה הדיגיטלית בחיינו. אני מתכוון להמשיך ולפתח את הידע שלי בתחום, לשמור על עדכניות עם מגמות חדשות ולהביא לידי ביטוי את הידע שרכשתי במהלך העבודה גם בפרויקטים עתידיים.

8. רשימת מקורות

[1] Yehoshua, Nir, and Uriel Kosayev. *Antivirus Bypass Techniques: Learn practical techniques and tactics to combat, bypass, and evade antivirus software*. Packt Publishing Ltd, 2021.

[2] Monnappa, K. A. *Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware*. Packt Publishing Ltd, 2018.

[3] *Koret, Joxean, and Elias Bachaalany. The antivirus hacker's handbook. John Wiley & Sons, 2015.*

[4] Panagopoulos, Ioannis. *Antivirus evasion methods*. Diss. University of Piraeus (Greece), 2020.

[5] Donadio, Jérémy, Guillaume Guerard, and Soufian Ben Amor. "Collection of the main anti-virus detection and bypass techniques." *Network and System Security: 15th International Conference, NSS 2021, Tianjin, China, October 23, 2021, Proceedings 15*. Springer International Publishing, 2021.

[6] Giorgos, Poulios. *Advanced Antivirus Evasion Techniques*. Diss. University of Piraeus (Greece), 2015.

[7] Ntantogian, Christoforos, et al. "Transforming malicious code to ROP gadgets for antivirus evasion." *IET Information Security* 13.6 (2019): 570-578.

[8] https://medium.com/@li_allouche/return-oriented-programming-rop-chain-358878d8bb02

[9] Shaid, Syed Zainudeen Mohd, and Mohd Aizaini Maarof. "In memory detection of Windows API call hooking technique." *2015 International conference on computer, communications, and control technology (I4CT)*. IEEE, 2015.

[10] *https://shells.systems/defeat-bitdefender-total-security-using-windows-api-unhooking-to-perform-process-injection/*

[11] *https://www.deepinstinct.com/pdf/antimalware-scan-interface-amsi-unchained*

Table of Contents

Abstract

One of the most common challenges today in the world of cyber security is evasion from defense products. It's a game of cat and mouse where the malware strives to be more sophisticated than the antivirus and the antivirus strives to be more sophisticated than the malware.

For this purpose, antivirus software implements several different mechanisms that work and focus on different areas of the system's operation to catch unwanted behavior that could be considered malicious. The attackers themselves also learn the methods of operation of those mechanisms and try to avoid them while running and when idle.

There is a wide variety of ways to identify malware which is used by antiviruses. One area we will focus on is the detection of malware evasion methods. To this end, I will explain how antivirus programs work and what are the mechanisms they use to detect malware, while understanding basic concepts in the field such as: static analysis, dynamic analysis, signatures, scanners and more.

In this work, we will review the way antiviruses work and possible evasion methods, both while running and while idle on disk. Most of the methods we will review both for detecting malware and for antiviruses evasion are generic and relevant to different operating systems.

In the first section of this work, I will focus on the identification methods for malware which resides on the disk and detail static analysis and dynamic analysis. I will explain what a static analysis mechanism is and present common methods to bypass it. In addition, I will focus on a special method presented in several articles which combines the use of ROP chains technology in order to bypass static analysis.

After that, we will explain what dynamic analysis is and present several different methods which malwares use in order to bypass this mechanism.

In the second section of the work, I will focus on detection methods for malware at runtime. On the other hand, I will present methods for bypassing those mechanisms. I will present a very common method among antiviruses that combines hooking WinAPI functions. I will explain the concepts and technologies that enable this detection method and present a few bypass methods that have been found to be harmful.

The last method I will focus on in my work is the detection of vulnerabilities at runtime using the AMSI mechanism. This mechanism comes as part of the Windows operating system and is open to expansion for antivirus programs, which makes it very common. After I explain the mechanism, I will present naive bypass methods in addition to more sophisticated methods for bypassing the AMSI mechanism.

Finally, I will briefly review the project that accompanies the work. In the project, I've implemented antivirus software that contains analyses to identify malwares both at runtime and idle while integrating a user-friendly graphic interface.

The Open University of Israel

Department of Mathematics and Computer Science

Review of evasion methods for antivirus mechanisms

Final Paper submitted as partial fulfillment of the requirements

towards an M.Sc. degree in Computer Science

The Open University of Israel

Department of Mathematics and Computer Science

By

Eyal Babian

Prepared under the supervision of Prof. Ehud Gudes

October 2024