**התאמת אפליקציות לעולם ה-High Availability**

מעבדה לתואר מוסמך

מנחים: פרופ' דן דולב, ד"ר טל אנקר, מר ירון ויינסברג

1. **תקציר**

בשנים האחרונות, ככל שגדלה התפוצה וגבר התלות של העולם במערכות מתוקשבות, נוצרה חשיבות מכרעת לזמינות (availability) של מערכות אלו. תחום ה-High Availability (HA) כולל את כל המחקר והפיתוח שמתמקד בהפיכת מערכות ורשתות לזמינות יותר. בתחום כלולים גם מגוון העקרונות וההנחיות שיש לעקוב אחריהם כאשר ניגשים לפיתוח תוכנה שמתוכננת לרוץ בסביבות בזמינות גבוהה (Highly Available). דא עקא, רוב התוכנות הקיימות בימינו לא תוכננו ועוצבו מראש תחת הכללים הנ"ל. האם נחרץ גורלם של תוכנות אלו, ולא נוכל להריץ אותם בסביבות בזמינות גבוהה? מטרת עבודה זו היא לנסות לפרט במה כרוכה הפעולה של הפיכת תוכנה להיות Highly Available למרות שלא פותחה כך במקור, ולנסות לענות על השאלה שהוצגה – האם המאמץ להפוך תוכנות ל-HA הוא אפקטיבי. במהלך העבודה נציג ניסיון להפוך תוכנה פופולרית בשם snort להיותHighly Available ואת המסקנות מהתהליך. תוצרי העבודה נמצאים ב: https://github.com/ElonN/snort-ha

1. **הקדמה**
   1. **High Availability**

תחום ה-High Availability מרכז את המאמצים שנעשים על מנת להוריד ככל האפשר את זמן ההשבתה של מערכות (downtime) – הזמן שמערכות אינן משרתות את הלקוח. טכניקה אחת להגדלת הזמינות היא שימוש נכון ביתירות(redundancy) – כלומר, הקצאת יותר משאבים (כגון מחשבים או ציוד רשת) מהמינימום הנדרש על מנת להגיע למצב שבו גם כאשר חלק מהמערכת כושל, יהיו מספיק רכיבים תקינים כך שהמערכת כולה תמשיך לתפקד. בתוך עולם היתירות, אחד האתגרים המשמעותיים הוא שמירה על סנכרון בין רכיבים כפולים. כאשר ישנם שני רכיבים זהים במערכת למטרת יתירות, נדרש לסנכרן ביניהם כך שגם כאשר רכיב אחד כושל, השני יידע להמשיך לתפקד החל מאותו רגע מבלי לאבד מידע שהיה קיים ברכיב הכושל. לכלל המידע שקיים ברכיב מסוים ומשפיע על המשך התפקוד שלו נקרא המצב (state) של הרכיב, והוא יכול לכלול מרכיבים שונים בתוכנה ובחומרה של אותו רכיב. חשיבות ה-state משתנה בין רכיבים שונים. ישנם רכיבים שהינם stateless ואין להם מצב – אם זה בגלל שבאמת הפונקציונליות שלהם אינה משתנה עם הזמן (למשל – שרת web שמגיש דפים שאינם משתנים), או לחלופין שיש להם state אבל הם יכולים להסתדר גם בלעדיו (למשל – נתב שיש לו routing table דינמי, אך במצב שבו ה-routing table נמחק הנתב יודע לבנות אותו מחדש ולהמשיך לתפקד).

בין רכיבים זהים ויתירים במערכת HA, שאינם stateless, נדרש סנכרון יעיל של ה-state על מנת שכשיגיע רגע האמת ואחד הרכיבים יהיה תקול, הרכיבים האחרים יכילו את כל המידע שצריך על מנת לתפוס את מקומו ולאפשר למערכת להמשיך לפעול בצורה תקינה. משימת סנכרון ה-state אינה קלה והיא תלויה כאמור גם בתוכנה וגם בחומרת הרכיב. לדוגמא – ה-state יכול לכלול מבנה נתונים בזיכרון שמשוכפל לכל מעבד, ואין דרך קלה להעביר את ה-state בין מכשיר עם ארבע ליבות מעבד לבין מכשיר עם שתיים. מסיבה זו, כאשר ניגשים לפתח תוכנה שאמורה לרוץ בסביבה שהיא Highly Available, יש לשים לב באופן מיוחד ל-state ולתכנן אותה כך שתהיה ניתנת לשכפול בקלות. למשל – design pattern נפוץ בסביבות web שהינן HA הוא הפרדה בין החלק שמטפל בבקשות המשתמש לבין מסד הנתונים בו שמור המידע עצמו. באופן כזה מקטינים את התלות בין עיבוד הבקשות (שהוא stateless) לבין ה-state. במצב כזה נאמר ששני חלקים אלו הינם loosely coupled ואפשר להחליף כל אחד מהם בקלות יחסית מבלי שינויים גדולים בחלק השני. עבור תוכנות שלא פותחו תחת עקרונות ה-HA, מלאכת אפיון ה-state יכולה להיות מפרכת. גם אם נתמקד בתוכנה בלבד, הרי שאין דרך גנרית להגדיר מהו ה-state של תוכנה ויש לבצע ניתוח ואיפיון מדויק של ה-state לכל תוכנה בנפרד.

גם לאחר איפיון ה-state – עדיין לא נפתרו כל הבעיות שלנו. עלינו למצוא דרך "לארוז" את ה-state – להגיע למרכיביו השונים בזיכרון או בדיסק או בכל מקום אחר ולהכין אותו לשליחה. לאחר מכן צריך לבצע את השליחה והקבלה במחשב אחר, ולבסוף "לשתול" אותו במופע אחר לגמרי של התוכנה. כל אחד ממרכיבים אלו אינו טריוויאלי, ובהמשך נראה כמה מן הבעיות שיכולות להתעורר בעת המימוש. בנוסף – מובן שאין זה מספיק לבצע תהליך זה באופן חד פעמי, יש צורך לדאוג שיתרחש באופן שוטף.

* 1. **Snort**

על מנת לבחון תהליך של הפיכת תוכנה כללית (שלא פותחה בראיית HA) – להיות מותאמת לסביבת HA, ניקח כדוגמא את התוכנה snort. Snort הינה תוכנת IDS/IPS (Intrusion Detection/Prevention System) פופלרית אשר מטרתה היא זיהוי וחסימת ניסיונות תקיפה ע"י הגדרת סכמות וחוקים וניתוח תעבורת רשת לאור חוקים אלו.

התוכנה הינה open source. פיתוחה החל ב-1998 ועד היום מפותחות גרסאות חדשות, כאשר הגרסא היציבה האחרונה נכון לכתיבת שורות אלה הינה 2.9.9.0. בנוסף – הוחל בפיתוח "הדור הבא" של snort, גרסא 3 או בשמה הנוסף snort++. כרגע snort++ משוחררת בגרסת אלפא 3.0.0a4.

שרת snort צריך להיות ממוקם ברשת כך שתגיע אליו כלל התעבורה שצריכה לעבור אנליזה. ניתן לחבר את השרת בטור – inline – כך שכל התעבורה עוברת דרכו בדרך ליעד, או במקביל, כך שתעבורה משוכפלת ומגיעה אליו – כפי שמתואר בתרשים להלן:



**תרשים – חיבור snort בטור (מימין) ובמקביל**

חיבור בטור מאפשר ל-snort לתפקד כ-IPS ולחסום תעבורה במידת הצורך (ולכן מכונה לעתים – מצב IPS). חיבור במקביל (מצב IDS) אינו מאפשר חסימת תעבורה אך במצב זה ה-snort אינו מהווה צוואר בקבוק מבחינת latency – כלומר, גם אם לוקח זמן ל-snort לעבד פקטה, זה לא פוגע ביעילות הרשת מבחינת זמנים.

ניתן לזהות בבירור כי תוכנה שכזו אינה יכולה להיות stateless – ישנם ניסיונות תקיפה אשר מצריכות שילוב של תנאים מקדימים כדי להעלות התרעה. דוגמא לכך היא זיהוי מחרוזת "זדונית" בתוך רצף של תעבורה – אם המחרוזת מתפצלת לשתי פקטות, ה-snort צריך "לזכור" אילו פקטות כבר עברו על מנת לדעת אם צירוף הפקטה הנוכחית יוצר מחרוזת "זדונית" שצריך להתריע עליה. ה-"זיכרון" הזה הוא בעצם חלק מה-state של snort. בעבודה זו ננסה לאפיין את ה-state של snort.

1. **מהלך העבודה**

העבודה התחלקה לשלושה חלקים

1. מחקר קוד – הבנת החלקים הרלוונטיים ב-snort ואפיון ה-state
2. פיתוח – תכנות מנגנון שיתוף ה-state בין מופעים שונים של snort
3. עבודת תשתית – בניית סביבת High Availability

נתאר בהרחבה כל חלק.

* 1. **הבנת הקוד ואפיון ה-state**

התוכנה snort הינה פרי עבודה של כמעט 20 שנה, והקוד נראה בהתאם – המון שורות קוד (כמעט 300,000) ועיצוב מורכב הכולל שכבות אבסטרקציה רבות. בחרנו לעבוד על גרסה 3 של התוכנה (snort++) – ספציפית 3.0.0a2 - מכיוון שנראה שנפתרו שם בעיות עיצוב שיכלו להקשות עלינו. למשל – ב-snort2 כל פלאגין צריך להרכיב בעצמו את הפקטות במידה והוא צריך לחפש תבנית ברצף מסוים. פעולות בסיסיות כגון הרכבה של TCP Stream – מצריכות התייחסות בכל פלאגין. בגרסא החדשה ניתן לבצע לחפש ברצף ה-TCP עם תמיכה יותר חזקה מתשתית הקוד.

במהלך העבודה נלמדו חלקים מהותיים מהקוד והושגה הבנה מספקת של חלקים אלו – סיכום התובנות שהושגו מצורף כנספח.

לאחר התבוננות ראשונית המסקנה היתה שלנסות לאפיין state מלא ל-snort היא משימה מורכבת בהרבה ממה שנראה מלכתחילה. הסיבה היא ש-snort מכיל תכולות רבות – תכולות תשתיתיות (כגון חלוקת התעבורה לכמה Flows לפי פרוטוקול / כתובות / פורטים) ותכולות פונקציונליות (כגון זיהוי מחרוזת) וכל תכולה כזו שומרת את ה-state שלה בצורה אחרת – ב-hash tree, במשתנים גלובליים ובדרכים אחרות. כלומר – על מנת לאפיין את ה-state של snort יש לאפיין state של כל תכולה בנפרד.

בעבודה זו אפיינו את ה-state של שתי תכולות בסיסיות של snort – אחת פונקציונלית והשניה תשתיתית.

1. תכולת זיהוי סריקת פורטים
2. תכולת מצב ההתקשרויות – סיווג התעבורה להתקשרויות לפי מאפייני פרוטוקול/כתובות/פורטים והחלטה לגבי כל התקשרות האם היא עוברת, נחסמת או עולה לבחינה נוספת

נציג חלקים אלו ואת תוצאות תהליך אפיון ה-state.

* + 1. **זיהוי סריקת פורטים**

אחת התכולות המעניינות ש-snort מציע הוא זיהוי ניסיונות של תוקף לסרוק את הפורטים ברשת. בתכולה זו יש משמעות רבה ל-state שכן אירוע של סריקת פורטים מורכב מפקטות רבות. חשוב לדעת לחבר את כל הפקטות שעוברות לכדי הבנה שצריכה לקפוץ התרעה. אם תוך כדי סריקה משמעותית המערכת נופלת ורץ מופע חדש של snort – יכול לקרות מצב בו אף אחד מהמופעים של snort לא ראה מספיק פקטות על מנת להקפיץ התרעה.

לאחר מחקר מקיף של התהליך (מתואר לעומק בנספח) הושגה ההבנה שכלל המידע הרלוונטי מרוכז ב-hash tree ספציפי בשם portscan\_hash.

בתוך portscan\_hash נשמרים נתוני הסריקה עבור כל שלישיה <protocol, src IP, dst IP >. כאשר מגיעה פקטה, נשלפים נתוני העבר מתוך ה-hash tree והם נבדקים אל מול הסף שמוגדר בקונפיגורציה. במידה ומספר הפקטות הסורקות גדול מהסף המערכת תקפיץ התרעה.

* + 1. **מצב ההתקשרויות**

התעבורה שעוברת ב-snort כוללות מספר רב של התקשרויות – התקשרות היא קבוצת פקטות המשויכת לשיחה ספציפית בין שתי ישויות לוגיות ברשת. בקוד משתמשים במונח Flows – שכן המילה connections בהקשר הזה הינה משויכת ל-TCP בלבד, וכאן הכוונה למגוון רחב יותר של פרוטוקולים. על מנת לחסוך במשאבי עיבוד, snort יודע לסווג התקשרויות לפי החלטות עבר – חלק מההתקשרויות כבר עברו אנליזה ואובחנו כלגיטימיות / זדוניות ולכן כאשר מגיעה פקטה נוספת מאותם התקשרויות אין צורך לבצע אנליזה מלאה עליה. אם הוחלט בעבר לחסום פקטות מהתקשרות מסוימת (אם יש אפשרות לחסום – כלומר במצב IPS) – פקטה חדשה מהתקשרות זו תיחסם ישר, כמעט מבלי לעבד אותה. גם במקרה ההפוך בו התקשרות מסוימת אובחנה כלגיטימית – כאשר פקטה נוספת תגיע, ניתן לתת לה לעבור מבלי להעביר אותה בחינה. כך ניוותר אך ורק עם פקטות מהתקשרויות שעדיין לא סווגו ונחסוך בעיבוד וב-latency.

נשאלת השאלה – היכן נשמר כלל המידע על ההתקשרויות? היינו רוצים להגיע למצב שבו כאשר מופע snort מתקלקל ועוברים למופע snort חדש, נוכל לשמר את הידע לגבי סיווג ההתקשרויות.

כדי להגיע לתשובה, תהליך קליטת הפקטה נבחן בעיון וגם במקרה זה נמצאו כמה hash trees בשמות אינדיקטיביים (tcp\_cache, udp\_cache וכו') אשר הכילו את המידע המבוקש. אותם ה-hash trees מכילים עוד הרבה יותר מידע, אבל אנחנו מצאנו את שחיפשנו - מצב ההתקשרויות (ALLOW/BLOCK). עבור כל התקשרות (חמישיית <protocol, src IP, src port, dst IP, dst port> כאשר בפרוטוקולים מסוימים אין פורטים) נשמר ב-hash tree המצב שלו (FlowState). כאשר התקשרות נוצרת, היא מסווגת כ-FlowState::INSPECT, וכל פקטה בה עוברת אנליזה מלאה, עד אשר המערכת מגיעה להחלטה אם ההתקשרות היא תקנית או זדונית, ואז היא מסווגת כ- FlowState::ALLOW או FlowState::BLOCK בהתאמה.

* 1. **פיתוח מנגנוני שיתוף ה-state**

אפיון ה-state הוביל למסקנה האופטימית כי אם נמצא דרך לשכפל את ה-hash trees המכילים את המידע על ה-state, נוכל לעבור בין מופעים שונים של snort תוך שמירה על המצב הקיים.

* + 1. **שכפול ה-state של מנגנון זיהוי סריקת הפורטים**

כאמור, המידע המבוקש נמצא במבנה נתונים hash tree בשם portscan\_hash. ה-hash tree הספציפי הזה הוא מהמחלקה sfx\_hash הממומשת ב-snort.

מימוש ההעתקה היה יחסית ישיר – הזיכרון מועתק הצידה בצורה סינכרונית ונשלח לשרת ה-redis cache. הבעיה היחידה שהתעוררה נגעה לכמות העדכונים של ה-cache. מובן שלא נרצה לעדכן בכל פעם שפקטה עוברת – לכן התווסף טיימר שגורם לכך שעדכונים יתרחשו בכל מרווח של שנייה אחת.

בצד השני – כאשר מופע חדש של snort עולה לאוויר, הוא מקבל מה-cache את ה-state הרלוונטי ל-portscan ומעדכן את ה-hash tree שלו.

אם נעמיק רגע בפרטים הטכניים – אנחנו מתערבים ב-pipeline של קבלת פקטה בפונקציה ps\_detect (בקובץ ps\_detect.c:1648). שם בודקים האם עבר אינטרוול מסוים ואם כן שומרים. כנ"ל בטעינה, אם ה-snort במצב passive.

* + 1. **שכפול ה-state של מצב ההתקשרויות**

לאחר שכפול מוצלח של ה-hash tree של זיהוי סריקת הפורטים, ננסה לשחזר את ההצלחה בשכפול נוסף של hash trees – הפעם מדובר ב-hash trees האחראים על שמירת מצב ההתקשרויות.

ישנו הבדל מהותי בין ה-hash trees הנוכחיים לבין portscan\_hash מסעיף 3.2.1 – במקרה של portscan\_hash – המידע (מצב סריקת הפורטים) – קיים בתוך ה-hash tree עצמו, ולכן כאשר משכפלים את מבנה הנתונים גם המידע משוכפל. ה-hash trees ששומרים את מצב ההתקשרויות לא מכילים את המידע עצמו אלא מצביעים למידע בזיכרון – מצביעים שכמובן אינם תקפים במעבר בין מחשבים. לכן הפתרון מהסעיף הקודם לא יעבוד על ה-hash tree הנוכחי.

ניתן לפתור זאת בכמה דרכים – למשל ע"י שינוי הקוד של snort כך שהמידע ישוכפל לתוך ה-hash tree או ע"י שימוש ב-offset במקום במצביע. הדרך שנבחרה בעבודה זו על מנת להתמודד עם הבעיה היא אחרת. מסתבר כי המצביעים במבנה הנתונים הנוכחי מצביעים על מידע שמאולקץ ברצף בזיכרון. התוכנה מקצה "גוש" זיכרון רצוף בו מוקצה כל המידע על התקשרויות, שבסופו של דבר נכנס ל-hash tree כמצביע. ניתן לנצל מצב זה לטובתינו – אם כל המידע נמצא שם, לא צריך להשתמש ב-hash tree כלל אלא רק לשכפל את "גוש" הזיכרון. גם אם המצב לא היה כזה – פתרון נוסף הוא לשנות את דרך הקצאת הזיכרון כך שהוא יגיע ממקום ספציפי, ולהעתיק את ה-state ממקום זה.

במקרה זה הקוד נמצא רוב רובו ב-flow\_control.cc שם מטפלים בכל פרוטוקול בנפרד (למשל – ICMP מטפלים בflow control:696). חשוב לציין שאנחנו שומרים כל cache של כל פרוטוקול בנפרד, לכן חשובה לנו ההפרדה. חלק נוסף של הקוד נמצא ב-stream base שם מאתחלים את אובייקט ה-flow control וגם את ה-redis.

* 1. **תשתית High Availability**

על מנת לבחון את תוצרי הפיתוח של עבודה זו נבנתה סביבה המכילה אלמנטים של High Availability. בסביבה זו ישנו שרת, לקוח, ויתירות ברמת רכיבי התקשורת – כלומר ישנם מספר מסלולים אפשריים בין השרת ללקוח וכאשר אחד מהם תקול, השני תופס פיקוד. להלן שרטוט רשת של הסביבה:



**תרשים – שרטוט רשת של סביבת ה- High Availability שבה נבחנו התוצרים**

בתרשים ניתן להבחין בשרת ובלקוח, כאשר יש שני מסלולים ביניהם - דרך R1 (192.168.9.1) ודרך R2 (192.168.9.2). מבחינת הלקוח – כל הקונסטלציה הזאת שקופה לו וישנו מסלול אחד לשרת – דרך הכתובת 192.168.9.254. כתובת זו היא כתובת "צפה" (Virtual IP) והיא יכולה להימצא אצל כל אחד מהנתבים – בתצורת active-passive. כלומר - בנקודת ההתחלה הכתובת אצל R1, וכל התעבורה עוברת דרכו. בזמן תקלה ב-R1, הנתב R2 מזהה את הבעיה ותופס בעלות על הכתובת – מרגע זה התעבורה עוברת דרכו והמערכת ממשיכה להיות זמינה.

השיטה בה נתבים בודקים זמינות אחד של השני היא באמצעות פרוטוקול VRRP (virtual routing redundancy protocol) – פרוטוקול סטנדרטי לתחזוקה ותפעול של Virtual IP בין כמה שרתים. בסביבה זו הפרוטוקול עובר ברשת ניהול נפרדת מהרשת בה עוברים הנתונים. בכל אינטרוול זמן נשלחת הודעה בין הנתבים שמהווה "סימן חיים". כאשר נתב גיבוי מזהה שלא קיבל סימן חיים, אחרי timeout מסוים הוא יתפוס בעלות על הכתובת. קבועי הזמן ניתנים לשינוי כתלות בצורך. כל התהליך הזה מנוהל בידי האפליקציה keepalived שמותקנת על שני הנתבים ומגדירה את ה-Virtual IP ואת קבועי הזמן של VRRP.

ברשת זו מותקנים שני מופעים של snort – שניהם מחוברים inline – הראשון בין R1 ל-R3 והשני בין R2 ל-R3. כאשר snort מחובר inline הוא מתפקד כ-bridge (layer 2) ואין לו IP באף אחד מהצדדים. ה-IP היחיד שיש לו הוא למטרות ניהול (על מנת להתחבר לשרת cache שנציג בהמשך).

אחד האתגרים שנצטרך להתמודד איתם עוסק בנתיב שחוזר מהשרת ללקוח – איך נתב R3 יידע לנתב את הפקטה בכיוון שאינו תקול? הרי הכתובת הצפה מפורסמת מR1/R2 לכיוון הלקוח ולא לכיוון R3. אופציה אחת היא ליצור עוד כתובת צפה, לכיוון R3. כיוון זה אפשרי ועובד, אך מאלץ אותנו להציב את שני המופעים של snort באותה רשת (subnet) כיוון שכתובת צפה יכולה להתחלק רק בין נתבים שנמצאים באותו subnet. החיסרון בלהציב את המופעים של snort באותו subnet הוא שהם יראו את כל הפקטות אחד של השני – ויוציא את העוקץ מהניסיון שלנו לשמר את ה-state (ומנגד – אם תהיה תקלה בגלל פקטה מסוימת, היא תקרה בשני הנתבים במקביל). לכן נרצה לדרוש ששני המופעים של snort יהיו ב-subnet נפרדים. הפתרון בו בחרנו להתשמש הוא שימוש בתשתית keepalived בין R1 ל-R3 למטרות תחזוקת טבלת הניתוב של R3 (ולא למטרות תחזוקת Virtual IP).

אתגר נוסף – ברגע שהמסלול מורכב מיותר מרכיב אחד (במקרה הזה –R1 ו-Snort1) – תקינות המסלול תלויה בכל הגורמים בדרך. כלומר לא מספיק לי ש-R1 ו-R2 בודקים אחד את השני, אני צריך ש-R1 יבדוק גם ש-Snort1 תקין. אם R1 מגלה ש-Snort1 לא תקין – עליו להעביר את הפיקוד ל-R2 כמו במקרה ש-R1 עצמו לא תקין. בדיקה זו הוספה כסקריפט בקונפיגורציה של keepalived.

על מנת שה-snort עצמו יידע אם הוא פעיל או סביל, נצטרך לסנכרן את ה-snort עם המצב של ה-Virtual IP (כלומר – מי הנתב הפעיל). הנתבים יחשפו החוצה קובץ בממשק HTTP, ומופעי ה-snort יגשו לקובץ על מנת לסנכרן את מצבם.

בנוסף, ברשת הניהול קיים שרת cache מסוג redis שמטרתו אחסון ה-state והעברתו בין המופעים השונים של snort. בתחילת הדרך נעשה שימוש בשרת memcache אך הוא נזנח מכיוון שיש בו הגבלה לגודל ערך הניתן לאחסון.

כלל הקונפיגורציות של הרשת והתקנות הרכיבים בה מצורפים לעבודה כנספח.

1. **מסקנות**
   1. **קשה להפוך לHA קוד שלא פותח לשם כך**

בעבודה נעשה ניסיון לקחת אפליקציה שרירות ולהפוך אותה לבעלת מאפייני HA. במהלך העבודה גילינו כי ניסיון שלנו לאפיין state גנרי לתוכנה נכשל – ה-state של התוכנה היה מפורז בין כל חלקי הקוד, בדרכים שונות, מה שהכריח אותנו להגדיר מטרות צנועות יותר כגון אפיון state של תכולה אחת. למעשה – מבין כל האתגרים בעבודה – אתגרי התשתית, הפיתוח והאפיון – אתגר זיהוי ה-state היה הקשה ביותר.

בשורה התחתונה – מבין הרבה מאוד תכולות ש-snort יודע להציע, הצלחנו במסגרת הזמן של העבודה להפוך רק שתיים מהן לניתנות לשיתוף בין מופעים שונים. לא ברור אם כך האם הציפיה באמצעות גישה זו לקחת תוכנה מלאה ולהפכה ל-HA בצורה גנרית ואפקטיבית היא ריאלית.

* 1. **בפיתוח מנגנוני שיתוף יש להתחשב באתגרים לא טריוויאליים**

בבואנו לפתח את מנגנון השיתוף עצמו (לאחר אפיון ה-state) נתקלנו באתגרים לא פשוטים

* + 1. כאשר "אורזים" את ה-state לחבילה הניתנת לשליחה – יש לשים לב שיש מרכיבים שיכולים להיות לא תקפים בין מחשבים שונים. בעבודה זו נתקלנו בבעיה זו בעיקר במצביעים לזיכרון. בעבודה זו עקבנו אחרי המצביעים והעתקנו את המידע מהם הצידה, אך פתרונות אחרים יכולים לכלול שינוי עמוק יותר בקוד עצמו.
    2. תהליך ה"אריזה" צריך מצד אחד להיות מהיר ויעיל, ומצד שני יש להיזהר שלא להשאיר state לא קונסיסטנטי. דרך אחת למימוש היא העתקת כל הזיכרון הצידה בצורה סינכרונית, אך זה כאמור צעד כבד שיכול להשפיע על latency, ומנגד אפשר שלא להעתיק שום דבר ולשלוח הכל בצורה א-סינכרונית (אך זה יכול להיות סכנה לקונסיסטנטיות הנתונים שכן הם משתנים תוך כדי). הפתרון שמומש בעבודה זו הוא העתקה סינכרונית באינטרוולים.
    3. יש לשים לב ש-state יהיה נגיש במהירות ככל האפשר – מנגנון השיתוף ניגש לא מעט פעמים לזיכרון ה-state. ישנה סכנה שהגישה לזיכרון ה-state תיקח זמן (למשל – בגלל שהוא כרגע paged לדיסק) ותוסיף latency. פתרון אפשרי הוא נעילה של ה-state לזיכרון כך שלא יעבור תהליך של paging – אך יש לשים לב שגודל ה-state הוא לא גדול מדי כך שנעילה שלו תקשה על המכונה לפעול. במקרה שלנו – גודל ה-state שהגדרנו היה 20-30MB ונעלנו אותו לזיכרון באמצעות הפקודה mlock.
  1. **בניית סביבת HA מורכבת מתהליכים פשוטים, שנהיים מורכבים משמעותית כשמספר הרכיבים שצריכים להיות מודעים למצב הגיבוי עולה**

בסך הכל בבואנו לבנות סביבת HA, התחלנו משירות פשוט שכל מה שהוא עושה הוא לבדוק באמצעות הודעות תקופתיות האם רכיב מסוים מגיב. ואכן – בסביבה מוקדמת שהסתכמה בשני רכיבים שמגבים אחד את השני (R1 ו-R2 בשרטוט הנ"ל), בניית הסביבה היתה מלאכה קלה.

כאשר התווספו לסיפור עוד כמה רכיבים (R3 ומופעי ה-snort) – המורכבות גדלה בצורה לא ליניארית. כלל הרכיבים שאמורים להיות מודעים למצב הגיבוי צריכים להיות בתקשורת מלאה אחד עם השני, ויש להתחשב בכל האופציות לתקלה. בסביבה שלנו – R3 היה צריך להיות מודע למצב הגיבוי כדי לבחור את חוקי הניתוב הנכונים, ומופעי ה-snort היו צריכים לדעת אם הם במצב אקטיבי או פסיבי. לשם כך הוספו מנגנוני סנכרון רבים בין כלל הרכיבים, שגרמו למורכבות משמעותית.

מנגנוני הסנכרון-

* נוסף עוד instance של keepalive בין r1 לבין r3 כדי שאם r1 ייפול, r3 יידע ויעדכן את חוקי הניתוב שלו לכיוון r2.
* R1, r2 בודקים האם הם נגישים ל-R3 על מנת לכסות את המצב בו snort קורס וצריך לעבור לנתב הבא.
* R1, r2מסתנכרנים עם snort1,snort2 בעזרת service שחושף ממשק WEB על r1/r2. בנוסף, על r1/r2 ישנה הגדרה ל-keepalived שגורמת לכך שבנתיב /tmp/master יש קובץ שמסמן האם הנתב פעיל או לא. את הקובץ הזה נחשוף ב-HTTP החוצה – על snort1/2 יש service בשם pyhttp\_get שמושך קובץ זה בכל שניה על מנת שגם snort יהיה מודע למצבו.

1. **כיווני המשך**

אפשר לקחת עבודה זו ולהמשיך אותה, להלן מספר כיוונים אפשריים

* 1. **המשך "כיבוש" snort**

בעבודה זו הוצגו שתי תכולות שהצלחנו לתת להם מאפייני HA, ישנם עוד המון תכולות כאלה ב-snort, חלקם ליבתיים יותר כגון מנגנון pattern-matching וחלקם פחות.

פרמטר נוסף שיש לבחון אותו לפני שבוחרים תכולה נוספת – התכולות שנחקרו היו כאלה שאינן רגישות לאיבוד פקטות (גם portscan וגם flow control הן רלוונטיות גם ללא 100% מהפקטות). ישנן תכולות שהן מאוד רגישות לאיבוד פקטות ויש לשים לב לכך. למשל – חיפוש substring בתוך stream – חייבים 100% מהפקטות אחרת נפספס.

* 1. **ייעול התהליך**

במצב הנוכחי כל העדכונים והמשיכות מה-cache מתרחשים בצורה סינכרונית, בכל אינטרוול נתון. ניתן לייעל את התהליך בדרכים שונות:

1. שילוב פעולות א-סינכרוניות – הפעולה הכי רגישה מבחינת זמנים היא פעולת השמירה ל-cache. פעולת הטעינה מה-cache מתבצעת רק כאשר ה-snort במצב passive ולכן אנחנו לא חייבים לדאוג ליעילותה. פעולת השמירה ל-cache חייבת להתבצע בזריזות. כאמור, כרגע אנחנו מבצעים פעולה זו בצורה סינכרונית בכל אינטרוול מסוים. ניתן לעבור למצב שהשמירה מתבצעת בצורה א-סינכרונית למשל תוך שימוש בפעולות הא-סינכרוניות המובנות בספריה hiredis (מצריך שילוב עם ספריית events כגון libevent).
2. אופטימיזציה על זמן האינטרוול – כרגע כל האינטרוולים (עדכון ושמירה ל-cache, אינטרוול VRRP, עדכון passive/active ל-snort) נקבעו שרירותית לשניה. אין סיבה לחשוב שזהו האינטרוול האופטימלי – ניתן לבצע בדיקות אופטימיזציה לפרמטר זה.
   1. **שימוש ב-RDMA לשיתוף ה-state**

כלל לא הזכרנו כי בעבודה זו ניסינו לשתף state בעזרת זיכרון משותף, ולא בעזרת הודעות – מכיוון שרצינו שבמידת הצורך נוכל להמשיך לשיתוף ה-state בעזרת טכנולוגיית RDMA שתתרום מאוד ליעילות ולמהירות. מימוש שלם שכולל RDMA יעלה מורכבויות אחרות לפרויקט זה.

* 1. **חיפוש תוכנה יותר קלה לשינוי**

בחירת התוכנה snort היוותה אתגר גדול מאוד – יכול להיות שיותר מדי. העובדה שכל תכולה קטנה לוותה בלמידה מאוד ארוכה וכל שינוי בקוד הצריך הבנה ותיאום בין המון רכיבים שונים, גרמה להתקדמות להיות איטית. ניתן לבחור תוכנה פשוטה יותר אך חשובה (כגון – iptables) ולהשתמש בסביבת ה-HA שהוצגה ואולי כך להגיע לתוצאות משמעותיות יותר.

1. **נספחים**
   1. **נספח טכני – מחקר קוד**

להלן תובנות לגבי מבנה הקוד:

**How do packets get through snort?**

The flow from a packet on the network until portscan detection

- main -> snort\_main -> main\_loop -> pig.start()

- pig.start inits new analyzer and calls it's function call operator() in a new thread like this:

"

analyzer = new Analyzer(source);

athread = new std::thread(std::ref(\*analyzer), idx, ps); // calls (\*analyzer)(idx, ps) in a new thread

"

- Analyzer implements the function call operator () and there it calls analyze()

- in function analyze - "DAQ\_Acquire(0, main\_func, NULL)" - main\_func is passed as callback to DAQ (if pcap is the DAQ module - pcap\_dispatch)

- main\_func is Snort::packet\_callback, which calls process\_packet, which calls main\_hook

- main\_hook can be set to be one of snort\_ignore, snort\_inspect, snort\_log - for us it is snort\_inspect

- snort\_inspect calls InspectorManager::execute

- InspectorManager::execute calls all "eval" funtions of the inspectors in the framework\_policy

- Each inspector has a function named "eval", which is passed a Packet\* as its only argument

**PORT SCAN:**

- port\_scan inspector "eval" performs a lookup in a hashtable and updates the table

- if "port\_scan" inspector detects port scanning over a certain threshold, it alerts

**FLOW CONTROL:**

- StreamBase is the inspector, it holds an instance of FlowControl (member StreamBase::flow\_con), and it's eval calls flow\_con->process\_X for packet of protocol X (== TCP/UDP/ICMP/etc..)

- FlowControl::process\_tcp processes each TCP packet, gets its "flow" object from tcp\_cache.

and calls FlowControl::process(flow, packet)

- FlowControl::process checks flow's flow\_state if needed, and if it's Flow::INSPECT, it calls flow->session->process(packet) which is TcpSession::process (tcp\_session.cc)

- tcp\_cache: FlowControl::init\_tcp allocates contingous memory for flow cache: tcp\_mem = calloc(fc.max\_sessions, sizeof(Flow)), then all Flow objects are puched into tcp\_cache

**Inspector assignment**

1. on new tcp connection (NewTcpSession in tcp\_session.cc) inpector is assiged according to flow->gadget

2. flow->gadget is assigned on binder::eval() --> binder::apply() --> stuff.apply\_service() [binder.cc] accordinng to get\_gadget(flow)

3. get\_gadget(flow) uses s=get\_protocol\_name() and then InspectorManager::get\_inspector(s)

**TCP Stream Pattern Searching**

1. noticed http requests/responses are validated (e.g checked that they start with "HTTP/") in validate() function

2. validate() is called by NHttpStartCutter::cut() (nhttp\_cutter.cc) [which is common to all classes inheriting NHttpStartCutter - e.g NHttpRequestCutter, NHttpStatusCutter]

3. cut() is called by NHttpStreamSplitter::scan (nhttp\_stream\_splitter.cc) which is a common interface to all StreamSplitter (inhreting StreamSplitter)

4. StreamSplitter::scan() is called by paf\_callback (paf.cc) <-- paf\_check (paf.cc) <-- flush\_pdu\_ackd, flush\_pdu\_ips(tcp\_reassembly.cc)

5. flush\_pdu\_ackd, flush\_pdu\_ips is responsible for iterating all tcp segments from seglist and scanning the reassembled buffer

**TCP Tracker (defined in tcp\_reassembly)**

1. tracker has all segments that were received on a specific connection in seglist

**HASHTREES IN SNORT**

Many plugins use hashtrees as the structures that hold the plugin's data. There are two common hashtree implementatiuon in the code - sfxhash and zhash.

The differences between them are unclear - but in docs it is stated that zhash is based on sfxhash.

sfxhash hashtree can also hold the data within the tree (and not only a pointers to data) - zhash holds only pointers to user-managed data.

The key size is constant, data size can be constant (for sfxhash) or pointers (zhash and sfxhash with datasize = 0)

hashtrees have the regular api, including get/add/remove/erase\_all - "get" in hashtree terminology also creates new node if doesnt exist, as apposed to "find"

* 1. **מתווה שינויי הקוד**

להלן הסבר קצר על השינויים שנעשו בקוד של snort

* + 1. **שיתוף state סריקת פורטים**

ה-hashtree של portscan שמור כמשתנה portscan\_hash בקובץ ps\_detect.cc:77

זהו עץ מסוג sfxhash שמוגדר ב-sfxhash.cc

העבודה כללה את הוספת הפונקציות:

sfxhash\_load\_from\_db

sfxhash\_save\_to\_db

שאמורות לקחת hashtree, לבצע לו סריאליזציה ולשמור אותו ב-redis (וכמובן, את התהליך ההפוך).

משם התהליך יחסית פשוט – נמצאה פונקציה שבה עוברת כל פקטה (הפונקציה ps\_detect), ושם מדדנו את האינטרוול בעזרת זמן הפקטה, ואם זמן האינטרוול עבר ה-hashtree נשמר/נטען כתלות במצב ה-snort (האם הוא active).

ה-snort מחליט באותו רגע האם הוא active לפי הדגל /tmp/master (שהוא 1 כאשר המכונה היא ה-master).

* + 1. **שיתוף state של התחברויות (Flows)**

ה-hashtrees שמצאנו שמכילים את המידע הנ"ל מוגדרים ב-flow\_control.cc (tcp\_cache, udp\_cache,…)

ישנם שישה cache-ים כאלה. הוספו הפונקציות

load\_cache (flow\_control.cc:205) הוא הפחות טריוויאלי – כי צריך להוסיף Flows פיקטיביים שלא קיימים (כדי ש-snort יידע לחסום פקטות משם, על סמך ה-Flow Cache ששיתפנו עוד לפני שהיו פקטות). לא פשוט להוסיף Flows פיקטיביים כי מקושרים אליהם אובייקטי session של snort ובאופן כללי flow הוא אובייקט מורכב, אבל אפשר להתחמק באלגנטיות אם מסמנים את ה-flow כאחד שהוא פיקטיבי -  
(יצרנו דגל בשביל זה באובייקט FLOW - current\_flow->new\_from\_cache = true). לאחר מכן מתייחסים אל הפקטה כאל פקטה מ-Flow חדש במקום אחר בקוד – ו-snort בעצמו מאתחל כמו שצריך את האובייקט. הקסם קורה ב-FlowControl::process – למקום הזה מגיעות גם פקטות ראשונות ב-Flow חדש ו-snort יוצר בשבילם את אובייקט ה-Flow וממלא אותו – אבל רק כאשר הוא לא מצא אותם ב-Flow Cache. כאשר אני משתף את ה-Flow Cache – יהיו פקטות ש-snort יתייחס אליהם כחלק מ-Flow מוכר, אבל אובייקט ה-Flow שלהם לא יהיה מאותחל מכיוון שכל מה שעשינו לו זה לסמן BLOCK או ALLOW.במקרה כזה ניעזר בדגל שהמצאנו – new\_from\_cache והוא יסמן לנו להתייחס לפקטה כאל פקטה מ-Flow חדש למרות שהוא נמצא ב-Cache.

save\_cache – גם כאן - פעם באינטרוול.

כעת, כל פקטה שמשנה את ה-state של ה-Flow (שימוש בפונקציה Flow::set\_state שנמצאת בFlow.h:189 היא הדרך היחידה לבצע שינוי של ה-state) – תגרום לשמירה ב-cache ולעדכון ה-redis.

* 1. **הקמת הסביבה**
     1. **הקמת הרשתות הפיזיות**

סביבת ה-HA הוקמה בעיקר ממחשבי Ubuntu Server 16.04, כאשר המחשב התוקף היה עם מערכת ההפעלה kali linux (מערכת הפעלה גם כלי תקיפה מובנים כגון סורק פורטים), ושרת ה-redis וה-server היו ממערכת ההפעלה lubuntu (הפצת Ubuntu הדורשת מעט משאבים). בדיעבד – אין צורך בגיוון מערכות ההפעלה וניתן להשתמש ב-Ubuntu Server עבור כל המחשבים בסביבה.

הסביבה הוקמה בעזרת מכונות וירטואליות על מחשב פיזי בודד, בווירטואליזציה של VMWare. הרשתות השונות הוקמו על בסיס lan segments (טכנולוגיה של VMWare המאפשרת חלוקת רשתות פשוטה.

כל מכונה חוברה לכמה lan segments לפי התרשים הבא (כל lan segment מיוצג ע"י צבע).



בנוסף, כל מכונה חוברה עם כרטיס רשת נוסף לאינטרנט בתצורת NAT של VMWare.

ברשת ה-NAT של VMWare חבר גם המחשב המארח (host), ננצל זאת על מנת לשתף קבצים בין המחשב המארח לבין המכונות הווירטואליות ובין המכונות. כדי לעשות זאת ניצור mount במכונות לתיקיה משותפת (share) על המחשב המארח. במקרה שלנו – המחשב המארח היה windows. יצרנו עליו share והתחברנו אליו מכל המכונות בעזרת הוספת mount קבוע לרשימת ה-mount-ים (הוספת שורה ב-/etc/fstab). השורה שהוספנו היתה

//192.168.170.1/proj /mnt/proj cifs nounix,noserverino,username=share,password=share,rw,file\_mode=0777,dir\_mode=0777,uid=1000,gid=1000

(השורה מצורפת בקובץ share.txt)

כאשר //192.168.170.1/proj הוא הנתיב ל-share במחשב המארח, אשר מכיל את הפרויקט כמו שהוא כשמבצעים clone מ-

https://github.com/ElonN/snort-ha

/mnt/proj הוא הנתיב ל-mount ב-linux.

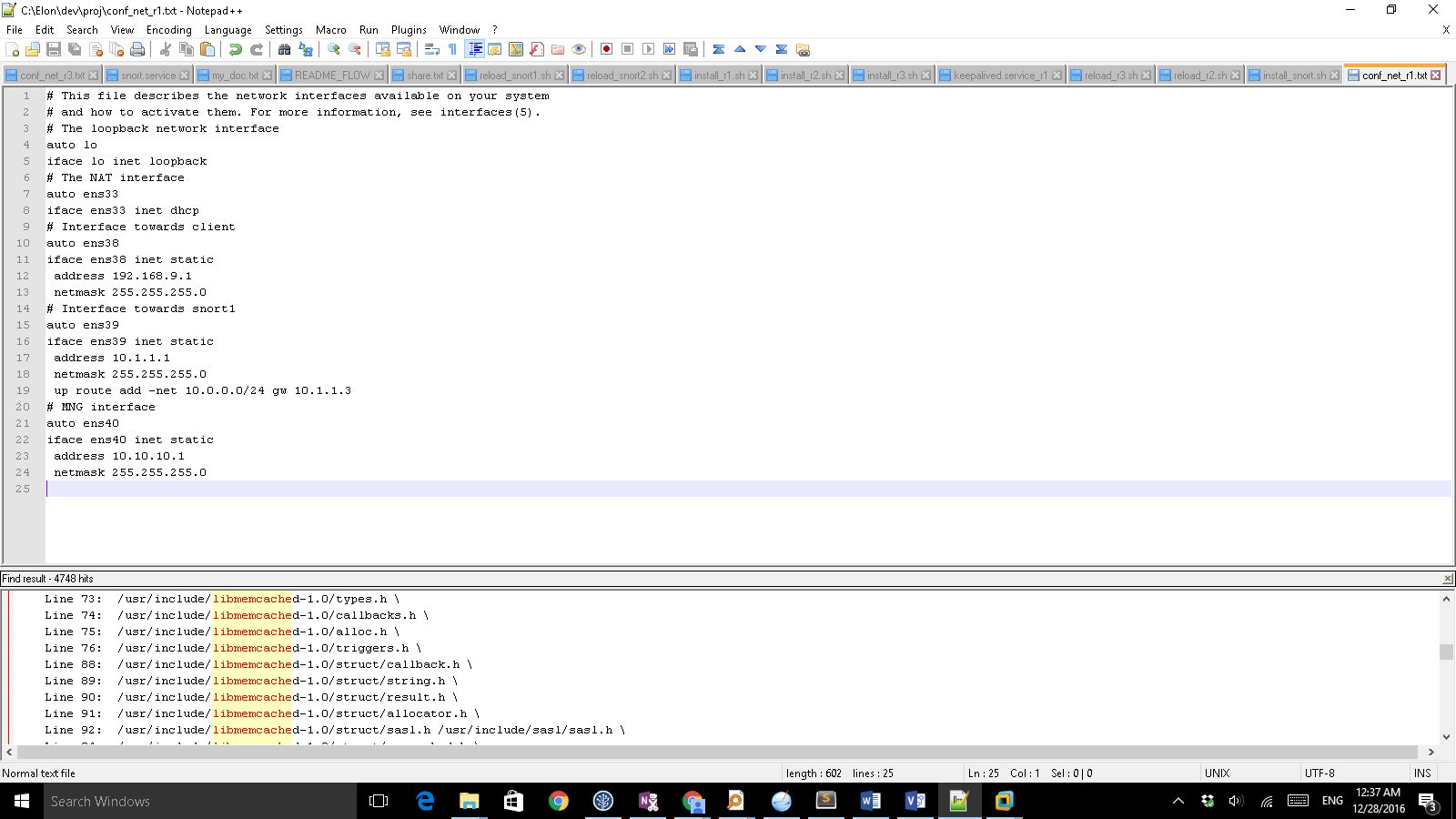
לאחר מכן יש לבצע את הפקודה

mount –a

על מנת לטעון את ה-mount מתוך הקובץ /etc/fstab. אין צורך לבצע פקודה זו שוב מכיוון שבכל הפעלה של המחשב מערכת ההפעלה תטען את כל ה-mount-ים מהקובץ.

* + 1. **התקנת העמדות**

הדבר הראשון שעלינו לעשות הוא להיכנס לקונפיגורציות conf\_net\_XXX ולשנות שם את שמות ה-interface בהתאם למה שנמצא במכונות הוירטואליות. למשל – ניקח לדוגמא את הקובץ conf\_net\_r1.txt. לפי הכתובות שאנו שמים בכל interface ניתן להבין לאיזו רשת הוא אמור להיות שייך. נוודא כי אכן יש תאימות בין שם ה-interface (ens38,ens39 וכו') שכתובים בקונפיגורציה לשם ה-interface שבו אמורה להיות כתובת זו לפי השרטוט. ז"א יכול לקרות מצב שבו בסביבה שהקמנו, ה-interface ב-r1 שפונה כלפי הרשת עם ה-client נקרא ens39, ובקונפיגורציית רשת של r1 (conf\_net\_r1.txt) אנו רואים כי ה-interface הפונה לכיוון ה-client (נזהה אותו לפי כתובת 192.168.9.X) נקרא ens38. במצב כזה - צריך לתקן זאת בקופיגורציה ולכתוב שם ens39 במקום ens38.



לאחר שווידאנו שכל שמות ה-interface מדויקים, ושהנתיב /mnt/proj על המכונות מפנה ל-share שלנו, נוכל להשתמש בסקריפטי ההתקנה שהכנו –

install\_XXX.sh עבור הרכיב XXX מתוך {r1, r2, r3, snort1, snort2, client, server}.

לאחר מכן אם נרצה לפתח את הקוד של snort או אחד הרכיבים האחרים במערכת (לדוגמא keepalive) – נסיים שלב בפיתוח ונריץ את reload\_xxx שאמור רק להעתיק מחדש את כלל הקונפיגורציות.

**חשוב: כלל הסקריפטים אמורים לרוץ ב-sudo ולאחריהם צריך לבצע reboot.**

* + 1. **שירותים (systemd services)**

בסביבה זו אנו משתמשים בשירותים על מנת להריץ אפליקציות בעת עליית המחשב בהרשאות גבוהות. קובץ שירות הוא מהצורה XXX.service והוא מכיל את הפקודה שנרצה להריץ ועוד פרמטרים כגון האם לווא שהתהליך רץ תמיד ולפעילו במחדש בעת קריסה (Restart=always), וכל כמה זמן להפעיל (RestartSec=1).

כדי להפעיל שירות, נעתיק את XXX.service לתיקייה /etc/systemd/system ונריץ

Systemctl enable XXX.service

לאחר מכן השירות רשום וניתן להפעיל מחדש את המחשב.

אנחנו משתמשים בצורה זו על מנת להריץ את keepalived ב-init של r1,r2,r3 וגם על מנת להריץ snort ב-snort1, snort2.

מכיוון ש-snort רץ כ-service, אנחנו משתמשים בפיצ'רים מובנים של systemd כגון watchdog אוטומטי. בנוסף – קבלת תקלות וקריאת לוג הם עניינים מורכבים כשזה service. פתרון שמצאנו לכך – שמרנו את לוג של הריצה בנתיב ב-/tmp/snort\_log וניתן לבצע tail –f על הקובץ כדי לקרוא אותו באופן רצוף.

נניח שקמפלתנו מחדש snort ואנחנו רוצים לגרום לו לרוץ – יש להרוג את ה-snort הישן ובאופן אוטומטי ה-snort מהתיקיה /mnt/proj/snort- 3.0.0-a2 ירוץ.

עוד חריגה – תהליך ההפעלה מחדש של snort מתבצע רק כאשר יש קובץ בשם /tmp/flag\_snort. אחרת, אם הורגים snort הוא נשאר מת.