
Undetectable Generic Aimbot

מאת אוראל דמרי (oreldm)

הקדמה

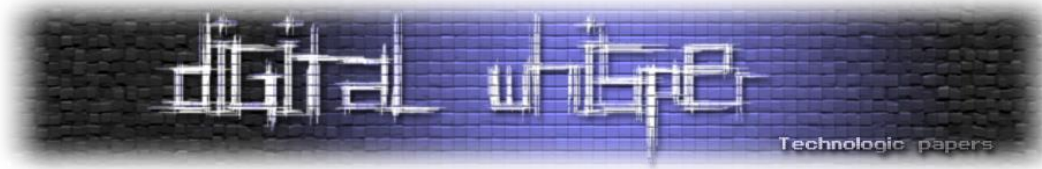
ב-2018, בטורניר eXTREMESLAND באסיה, קרה אחד הרגעים המביכים בתולדות קאונטר סטרייק: השחקן המקצוען forsaken [נתפס בשידור חי](#) מול קהל בלאן, כשדמין לוקח לו את המחשב והוא מנסה נואשות למחוק אימבוט שהוסווה כקובץ תמים. שנים קודם, בגל ה-bans VAC של 2014, כמה מהשמות הגדולים בענף (ביניהם KQLY) ננעלו על אותו חטא. וכל פעם שראיתי כתבה כזו, במקום לחשוב "איזו בושה", חשבתי משהו אחר לגמרי: אלה נתפסו. מי עוד לא נתפס?

השאלה הזו הובילה אותי למחשבה הנדסית פשוטה: כדי לענות עליה באמת, צריך קודם לבנות צייט שאי-אפשר לתפוס - לא "קשה לתפוס", אלא כזה שאין לו טביעת רגל בכלל - ורק אז לנסות לתפוס אותו. שלב התפיסה הוא מאמר שלם בפני עצמו, ונשמור אותו לפעם הבאה. המאמר הזה עוסק בשלב הראשון, שלב-התקיפה: איך נראה אימבוט בלתי-נראה, ולמה הוא כזה כאב ראש לכל מערכת הגנה. אני כותב אותו מזווית הגנתית - כי אי-אפשר לבנות זיהוי טוב בלי להבין את התקיפה עד הסוף.

הרעיון בכלליות: אימבוט המפוצל לשלושה חלקים, כשכל חלק מנטרל אחד משלושת עוגני הזיהוי שעליהם נשען כל אנטי-צייט. העיניים חיות מחוץ למחשב המשחק (ראייה ממוחשבת על מכונה נפרדת - אין גישה לזיכרון), הידיים מזריקות קלט דרך שכבת-הרדיו של העכבר עצמו (אין התקן זר), והתנועה נבנית ממודל קינמטי של יד אנושית (אין תנועה רובוטית). שלושתם יחד מכבים את שלושת העוגנים בו זמנית - וזה בדיוק מה שהופך את הארכיטקטורה למטרידה.

ומילה על הפרופורציות, כי הן מפתיעות. שני החלקים שנשמעים זוהרים - ה-LLM והתנועה האנושית - לקחו לי יחד כחודשיים. החלק השלישי, הזרקת-העכבר, לקח לי שנתיים. הרדיו הוא לב העבודה כאן: הוא דרש לצלול ל-firmware, לאסמבלי של M-Cortex ולהנדסה-לאחור מאפס. בניגוד לכתיבת קוד - שם כל מודל AI ישמח להשלים לך את השורה הבאה - ברגע שנכנסים לעולם החומרה, אתה לבד. אין Stack Overflow שמכיר את הבאג הספציפי שלך, ואין מודל שיסביר לך למה הדונגל פשוט מתעלם ממך. לכן, בהתאם, רוב המאמר מוקדש לרדיו.

אני כותב את המאמר מזווית של מהנדס תוכנה, ולכן אסביר מושגי בסיס ברדיו במהלך המאמר. חשוב לציין שעבור מפתח-לממש היום את הרעיון באופן בסיסי-העומק הטכני אינו הכרח. עם זאת, מהכיוון ההגנתי-חשוב להבין כל שלב לעומקו כדי לחשוב ולמצוא וקטורי תפיסה. שלב התפיסה - וגם ערימת הרעיונות שנכשלו בדרכי-יחכו למאמר ההמשך. עכשיו, לעבודה.



אנטי-ציט: שלושת עוגני-הזיהוי

אנטי-ציט הוא מערכת-אבטחה שרצה לצד המשחק כדי לוודא שאתם - ולא תוכנה במקומכם - משחקים. מערכות מוכרות כמו Vanguard Riot, Cheat-Anti Easy ו-BattleEye רצות לרוב כדרייבר ברמת-הקרנל (0-ring, השכבה הכי-מיוחסת במערכת-ההפעלה) כדי לראות כמה שיותר. בעולם ה-CS (קאונטר סטרייק) התחרותי, Cheat-Anti FACEIT נחשב לאחד המחמירים והחזקים שיש. כמעט כל מה שהמערכות האלה עושות נופל לאחת משלוש קטגוריות, ושלושתן יחד הן ה"מסגרת" של המאמר הזה.

וכמה מחמיר FACEIT? עד כדי כך שהוא עלול להעניש על עצם הרצת כלים "חשודים" ברקע - למשל Wireshark או IDA פתוחים בזמן משחק - ואף אוסר על הרצת המשחק בתוך מכונה-וירטואלית (VM), כשניסיון לעקוף את החסימה הזו מסתיים בבאן. זו לא פאראנויה מוגזמת. זו בדיוק רמת-הרצינות שמולה הארכיטקטורה שנתאר צריכה לעמוד. למי שרוצה לצלול לאיך מערכת כזו בנויה מבפנים [בגיליון 187 יש מאמר נפלא](#) של דור נאמני שמפרק את Cheat-Anti Easy. ממליץ בחום לחזור לאותו גיליון.

עוגן 1 - זיכרון (Memory)

קריאה של מרחב-הכתובות של תהליך המשחק: חיפוש חתימות של ציטים ידועים, זיהוי DLL-ים שהוזרקו, hooks על פונקציות (למשל inline hooks/IAT), וקריאות/כתיבות חשודות למבני-הנתונים של המשחק (מיקומי שחקנים, בריאות וכו'). אם הציט שלכם קורא את מיקומי היריבים מהזיכרון כדי לחשב זווית - כאן תופסים אותו.

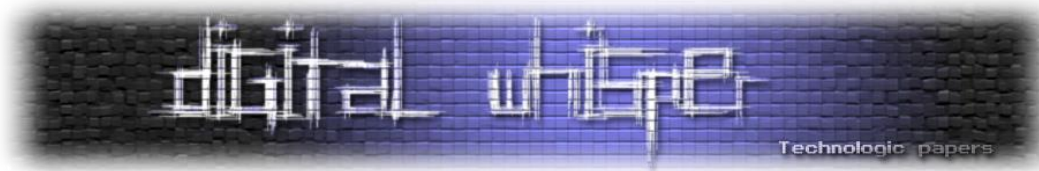
עוגן 2 - התקן-הקלט (Input)

המערכת יודעת אילו התקני-HID מחוברים, מה ה-PID/VID שלהם (מזהי היצרן והמוצר של כל התקן-USB), ואיך נראים דוחות-הקלט שלהם. הזרקת-קלט תוכניתית (SendInput - פונקציית מערכת-ההפעלה להזרקת אירועי-קלט) בולטת מיד. גם עכבר מבוסס-Arduino או התקן DMA שמדבר USB-HID הוא בסופו של דבר התקן שאפשר לספור, לטבוע (fingerprint) ולפעמים לזהות לפי תזמונים או descriptor חשוד.

זה לא תיאורטי: Vanguard, האנטי-ציט של Valorant, כבר ביצע Ban Waves ממוקדים נגד ציטים מבוססי-Arduino - עד כדי זיהוי מזהי-ה-USB האופייניים של לוחות-Arduino נפוצים ופסילתם. השורה-התחתונה: כל התקן-קלט זר, חכם ככל שיהיה, הוא משטח-תקיפה שאפשר לטבוע ולתפוס. וזו בדיוק הנקודה שהחלק השני שלנו עוקף - כפי שנראה, הוא לא מוסיף התקן בכלל.

עוגן 3 - התנהגות (Behavior)

גם אם לא נראה דבר בזיכרון ולא זוהה התקן חשוד - עדיין אפשר לנתח את הטלמטריה של איך אתם משחקים: התפלגות זמני-תגובה, חדות הנעילות (snaps), חלקות המסלולים, והמתאם בין רגע-הופעת-היריב-על-המסך לבין תחילת-התנועה. זה המישור הכי קשה לתקוף, וגם המישור שהחלק השלישי שלנו (התנועה האנושית) נועד להתמודד איתו.



שווה להבין למה המרוץ הגיע עד 0-ring. בהתחלה אנטי-צ'יט רץ ב-mode-user, וזה נכשל: צ'יט שרץ בקרנל יושב מעל מי שמנסה לתפוס אותו ויכול לשקר לו. אז ההגנה ירדה גם היא לקרנל, והתוקפים ירדו עוד - לצ'יטים מבוססי DMA שקוראים את הזיכרון דרך חומרה חיצונית (PCIe), ולצ'יטים מבוססי-hypervisor. הקו התחתון של המרוץ הזה: מי שיושב בשכבה הנמוכה יותר מנצח. הארכיטקטורה שנתאר לא מנסה לנצח את המרוץ הזה - היא פשוט יוצאת מהמחשב לגמרי. וקשה מאוד לתפוס משהו שלא נמצא אצלך.

מה זה אימבוט, ולמה בדרך-כלל תופסים אותו

אימבוט הוא צ'יט שמכוון בשבילכם: מזהה יריב ומזיז את הכוונת אליו. הסוגים נבדלים בדיוק בשאלה איזה עוגן הם מפעילים:

- **מבוסס זיכרון:** קורא מיקומים מהמשחק (screen-to-world). מדויק להחריד - ונתפס בעוגן-הזיכרון.
- **מבוסס פיקסלים/צבע:** מחפש את צבע-המתאר של היריב. פשוט, אך שביר ותלוי-משחק.
- **מבוסס ML:** מזהה יריבים כמו שאדם מזהה. גנרי, ואם רץ על מכונה נפרדת - לא נוגע בזיכרון. זה החלק הראשון שלנו.
- **מבוסס חומרה:** מפריד את המוח מהידיים. המשפחה שאליה הפרויקט הזה שייך - רק שאנחנו לוקחים את הידיים צעד רחוק יותר.

הרעיון: שלושה חלקים נגד שלושה עוגנים

העיניים (YOLO): רצות על מחשב נפרד שמקבל את הווידאו דרך Splitter HDMI וכרטיס-לכידה. אפס גישה לזיכרון ← עוגן-הזיכרון עיוור.

הידיים (הזרקה אלחוטית): מתחזות לעכבר הקיים בשכבת-הרדיו של 2.4 GHz במקום להוסיף התקן ← עוגן-הקלט עיוור.

הכיסוי (תנועה אנושית): כל תיקון-כיוון נבנה ממודל קינמטי של יד ← העוגן ההתנהגותי לא מוצא חתימה טריוויאלית.

End-to-end loop: external eyes -> aim math -> human motion -> radio injection



no game memory access | no new input device | no robotic motion

[תרשים 1: הלולאה המלאה]

Undetectable Generic Aimbot

www.DigitalWhisper.co.il



חלק א' - העיניים: עיבוד-תמונה ו-YOLO

הקוד של חלק זה:

<https://github.com/Oreldm/screen-target-detection>

Splitter HDMI - לראות בלי להיראות

הבסיס של החלק הזה הוא הפרדה פיזית. HDMI Splitter פאסיבי מכפיל את יציאת-הווידאו: פלט אחד הולך למסך, השני לכרטיס לכידה שמחובר למחשב שני. הנקודה החשובה מבחינת אבטחה: מקור-ה-HDMI לא יכול לדעת שחובר אליו צרכן שני. אין ערוץ חזרה שמדווח משהו מקשיב. המידע היחיד שחוזר למקור הוא ה-EDID (תיאור יכולות-המסך), והמפצל פשוט מציג EDID אחד מוסכם. כלומר מחשב-המשחק משדר את התמונה שלו לעולם, בדיוק כמו עכבר אלחוטי ששולח את התנועה שלו לאוויר - ואי-אפשר לדעת מי מאזין. המחשב השני רואה בדיוק את מה שהשחקן רואה, אבל מבחינת מחשב המשחק הוא לא קיים.

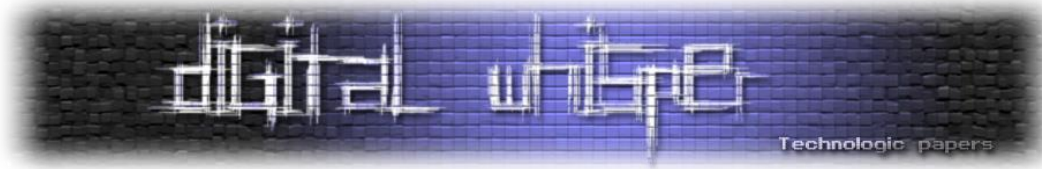
מה זה YOLO? YOLO (Once Look Only You) הוא משפחת-מודלים לזיהוי-אובייקטים בזמן-אמת: רשת נוירונים קונבולוציונית שמקבלת תמונה שלמה ומחזירה, במעבר-קדימה יחיד, רשימת תיבות-תוחמות עם מחלקה ורמת-ביטחון. המימוש הנפוץ הוא של Ultralytics. תפקידו כאן: העיניים. קישור: ultralytics.com.github. הרעיון המקורי: [al et Redmon, 2016](#).

הערה כנה מראש: הספרייה הזו כבר כתובה, ואפשר פשוט להשתמש בה - לא באמת חייבים להבין לעומק איך YOLO עובד כדי להריץ אותו על כמה תמונות. אבל כדי לבנות משהו טוב באמת - מדויק, מהיר, וכזה שרץ גם על מחשב חלש ובזמן-אמת - הייתי צריך להבין מה קורה מתחת למכסה-המנוע. אז נסביר בקצרה, ומי שמכיר ML ידלג.

כמה מושגי-יסוד ב-ML למי שלא מהתחום: רשת קונבולוציונית (CNN) היא רשת-נוירונים שלומדת מסננים שמזהים דפוסים בתמונה (קצוות, צורות, ובסוף - ראש). Inference (הסקה) הוא פשוט הפעלת הרשת המאומנת על תמונה חדשה כדי לקבל תשובה. ולשם השוואה - CNN-R Faster הוא גלאי דו-שלבי: קודם הוא מציע אזורים חשודים בתמונה ואז מסווג כל אחד. זה מדויק אבל איטי. YOLO, לעומתו, עושה הכול במעבר אחד - ולכן הוא מהיר מספיק לוודאו ח'י, וזו הסיבה שבחרנו בו.

איך YOLO עובד, ברמה הנמוכה

ההבדל המרכזי מגלאים דו-שלביים כמו CNN-R Faster - שקודם מציעים אזורים חשודים ואז מסווגים כל אחד בנפרד - הוא ש-YOLO עושה הכול בבת-אחת: התמונה מוקטנת לרזולוציה קבועה (בדרך-כלל 640x640) ועוברת ברשת פעם אחת, וכל אזור בפלט מקבל ניחוש של תיבה עם רמת-ביטחון. בסוף יש שלב קטן שנקרא Non-Max Suppression (NMS) שמאחד תיבות חופפות ומשאיר לכל יריב תיבה אחת, ומסננים לפי סף-ביטחון (למשל 0.55). כי הכול קורה במעבר יחיד, זמן-החישוב יורד לבודדי-מילישניות גם



על כרטיס-מסך צנוע - מספיק לוודאו חי. את המודל מאמנים על כמה אלפי צילומי-מסך מתויגים, כך שהוא לומד איך יריב נראה - ולכן אותה גישה עובדת על מגוון משחקים בלי לגעת בקוד של אף אחד. ונקודה עקרונית שקל לפספס: **הגישה הזו גנרית לכל משחק-ירי**, כי ראש הוא ראש. אין שום היגיון לאמן מזהה נפרד לראש של CS2 ולעוד אחד לראש של Valorant - בדיוק כפי שאדם מזהה ראש בכל משחק בלי ללמוד כל משחק מחדש, כך גם YOLO. מאמנים מזהה-ראשים אחד, גנרי, והוא עובד על פני משחקים שונים. רק אחר-כך, אם בכלל, מוסיפים אילוצים פשוטים - למשל בדיקת צבע-המדים או מתאר-היריב - כדי לוודא שמכוונים לראש של יריב ולא של חבר-לקבוצה. וזה, בהשוואה לרדיו, החלק הקל.

הארכיטקטורה של YOLO, בגסות

בגדול אפשר לחשוב על YOLO כשלושה שלבים: שכבות שמחלצות מהתמונה מאפיינים חזותיים (מקצוות פשוטים ועד צורות מורכבות), שכבות שממזגות מאפיינים בגדלים שונים - כדי לתפוס באותה מידה ראש קטן-ורחוק וגם גדול-וקרוב - ולבסוף שכבה שמנבאת לכל אזור בתמונה תיבה, וכמה בטוחה הרשת שיש שם אובייקט. אין צורך לרדת מכאן לפרטים. העיקרון היחיד שחשוב לזכור הוא שהכול קורה במעבר אחד על התמונה, ומכאן המהירות.

על האימון: אוספים כמה אלפי צילומי מסך, מתייגים תיבות סביב ראשי-היריבים (אפשר מחלקה נפרדת לראש ולגוף), ומריצים tuning-fine על משקולות-בסיס. כמה טיפים מהשטח: איזון בין תמונות עם-יריב לבין תמונות ריקות מפחית positives-false. אוגמנטציה (שינויי-בהירות, טשטוש-תנועה, שינויי-צבע) עוזרת להכליל בין מפות ותנאי-תאורה. וסף הביטחון נבחר לפי הטרייד אוף בין נעילה מהירה לבין קפיצות שווא. אימון בסיסי מגיע לדיוק גבוה תוך שעות בודדות על GPU יחיד (בערך פי-מאה פחות זמן ממה שלקח לי הרדיו).

```
#training (Ultralytics) - github.com/ultralytics/ultralytics
from ultralytics import YOLO
model = YOLO('yolov8n.pt') # 'nano' = fastest at inference
model.train(data='heads.yaml', epochs=100, imgsz=640, augment=True)
# heads.yaml points to images + YOLO-format labels. classes: [enemy_head]
```

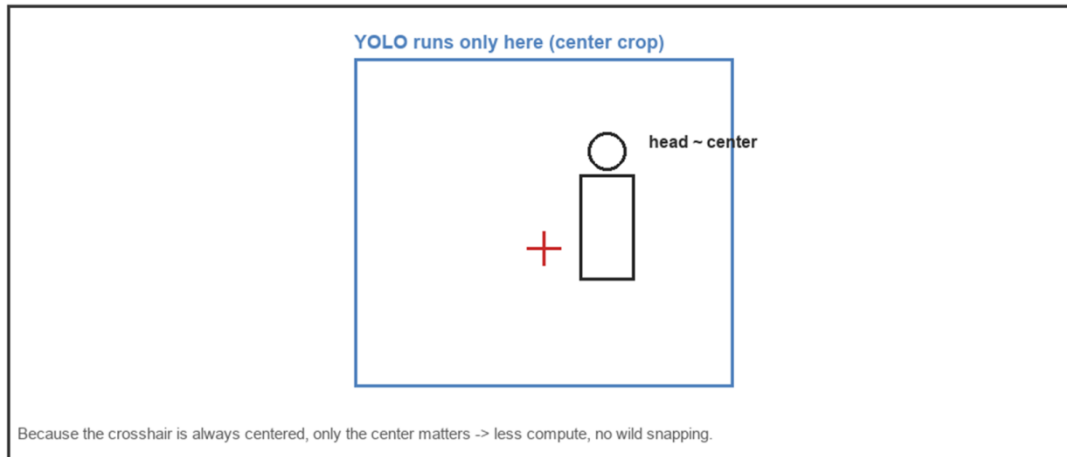
קצת אופטימיזציות: לעבד רק את מרכז-המסך, להריץ על ה-GPU

וכאן טריק נחמד ששווה זהב. הכוונת במשחק-ירי נמצאת תמיד במרכז המסך. מכאן שלא צריך להריץ את YOLO על הפריים המלא (0x1080192) - מספיק לחתוך אזור מרכזי קטן (למשל 0x32032 סביב המרכז) ולהריץ עליו בלבד. זה נותן שני יתרונות בבת-אחת. ראשית, חיסכון: קלט קטן יותר ל-inference (השלב שבו המודל המאומן "חושב", מעבד את התמונה בפועל ומנחש היכן נמצאות המטרות בזמן אמת) פירושו מעבר-קדימה מהיר בהרבה ו-FPS גבוה יותר, וזה קריטי בתקציב-הזמן. שנית, ואולי חשוב יותר, יציבות: אם נעבד את כל המסך, הכוונת תרצה לקפוץ אל כל ראש שמופיע בפינה - קפיצות פרועות שגם מזיקות למשחק וגם צורחות בוט! לכל עוגן התנהגותי. אבל ממילא אתה מכוון אל היריב שמול הכוונת, כלומר אל



מה שקרוב למרכז - אז חיתוך למרכז הוא לא רק אופטימיזציה, הוא גם התנהגות נכונה יותר. חסכתי חישוב, וגם קיבלתי תנועה שפוייה. שני ציפורים.

Full frame 1920x1080 (from HDMI capture)



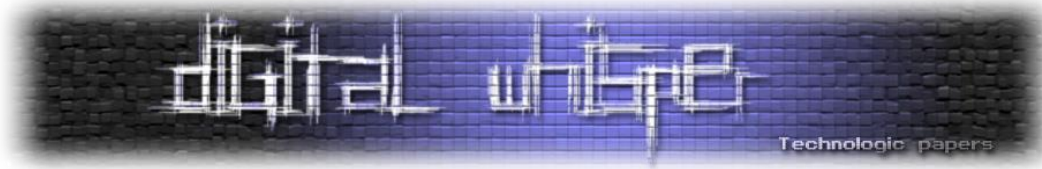
[תרשים 2: עיבוד רק על מרכז-המסך - פחות חישוב, בלי קפיצות אל ראשים בפניה]

זה גם חשבון, לא רק אינטואיציה: עלות החישוב של רשת קונבולוציונית גדלה בקירוב ריבועית עם ממדי-הקלט. מעבר מקלט של $0x64064$ לקלט של $0x32032$ מוריד את מספר-הפיקסלים פי-ארבעה, וחותר נתח נכבד מזמן-ה-inference - כלומר יותר FPS, שזה קריטי כשמנסים לעמוד בקצב-פריימים. מכיוון שהכוונת ממורכזת ממילא והמטרה הרלוונטית תמיד קרובה למרכז, אנחנו פשוט לא משלמים על זיהוי אזורים שלעולם לא נירה אליהם. פחות חישוב, יותר יציבות, אותה תוצאה - נדיר לקבל את שלושתם יחד. אופטימיזציה משלימה: להריץ את ה-inference על כרטיס-מסך של NVIDIA. YOLO רץ על ה-GPU דרך CUDA, ואפשר לדחוס אותו עוד עם TensorRT (ממיר את המודל לגרסה מותאמת-חומרה, לרוב ב-FP16), מה שמוריד את זמן-ה-inference אל מתחת למילישנייה בודדת. השילוב - חיתוך-מרכז + GPU של NVIDIA - הוא מה שמאפשר להריץ את כל חלק הראייה בזמן אמת גם על מחשב-משני צנוע (במקרה שלי- תוצאה חלקה על i7 שיושב על לפטופ משנת 2011).

מפיקסל ל-count של עכבר

כדי להזיז את הכוונת לנקודת המטרה שה-YOLO זיהה, ממירים את המרחק בפיקסלים לזווית לפי ה-FOV, ואז מתרגמים אותה ל-counts ישירים שהמנוע מקבל. התרגום ל-counts תלוי ב-sensitivity של המשחק ובמקדם הסיבוב של המנוע ($m_yaw = 0.022$ ב-Source), ללא תלות ב-DPI החומרתי של העכבר או ב-eDPI. מי שרוצה את הנוסחה המלאה יצא אותה בקוד שלמטה:

```
#detect + aim math (source: github.com/OreIdm/screen-target-detection)
det = model(center_crop, conf=0.55, iou=0.45)[0] # single forward pass, ~1-3 ms
enemy = nearest_to_crosshair(det.bboxes.xywh) # crop is already centered
px, py = enemy[0], enemy[1] - 0.35*enemy[3] # aim at the head line
f = (W/2) / tan(radians(FOVx)/2)
counts = degrees(atan2(px - Wc/2, f)) / (sensitivity * 0.022) # -> deltas to inject
```



חלק ב' - הידיים: הזרקת-עכבר ו-Ghidra

הקוד של חלק זה (פריים-ההזרקה ל-Crazyradio + כלים):

<https://github.com/Oreldm/crazyradio-esb-inject>

כלים: MouseJack (מחקר של Bastille, 2016) הראה הזרקת-קלט לדונגלים אלחוטיים לא-מוצפנים ב-GHz2.4 Crazyradio 2.0 (של Bitcraze) הוא דונגל-USB סביב שבב Nordic nRF52840, המשמש כרדיו-התוקף. LOGITacker (של RoganDawes) הוא firmware מחקרי ל-nRF5 עם מצב promiscuous ומחסינת-ESB, ששימש בסיס לקוד שלנו. Ghidra הוא כלי הנדסה-לאחור החינמי של ה-NSA. קישורים בסוף המאמר.

איך רדיו-העכבר עובד באמת

נעצור על הרדיו, כי בתור מהנדס תוכנה בלי רקע מעמיק בחומרה - כאן נמצא כל הקושי. עכבר אלחוטי משדר בפס ה-GHz2.4 - אותו פס של WiFi ו-Bluetooth - המחולק לכ-80-100 ערוצים במרווחי מגה-הרץ. כל ביט הופך לגל באמצעות GFSK: זו שיטת-אפנון פשוטה שבה 1 ו-0 מיוצגים על-ידי הזזה קטנה של תדר-הנשא מעלה או מטה (Keying Shift-Frequency = FSK), כשה-G (Gaussian) פשוט מחליק את המעברים כדי לא לפזר אנרגיה לערוצים שכנים. אין כאן מודם מתוחכם - יש כאן שינוי-תדר זעיר, ומקלט שמנחש לכל סימבול מאיזה צד של הנשא הוא היה. קצב טיפוסי: 1 או 2 מגה-ביט לשנייה.

מעל השכבה הפיזית יושב ShockBurst Enhanced (ESB) (פרוטוקול של חברת Nordic לשליחת נתונים), וכך נראית חבילה על האוויר, משמאל לימין: Preamble (בייט אחד, 0xAA או 0x55, שמאפשר למקלט לנעול את השעון שלו על הקצב). Address (3 עד 5 בייטים - המקלט מקשיב אך ורק לכתובת שלו). PCF (Field Control Packet, 9 בייטים: 6 ביט לאורך-ה-payload, 2 ביט PID ו-ביט אחד של ack-no). Payload (0-32 בייטים - דוח-העכבר עצמו). ולבסוף CRC (1-2 בייטים). זהו. אין הצפנה, אין חתימה, אין מפתח.

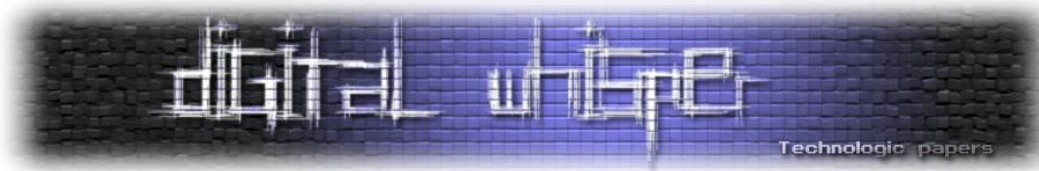
Enhanced ShockBurst (ESB) — the actual on-air packet

Preamble	Address	PCF	Payload	CRC
1 byte clock sync	3-5 bytes the only 'auth'	9 bits len PID no-ack	0-32 bytes the mouse report	1-2 bytes error check

The dongle accepts ANY well-formed packet carrying its address -> a second radio with the same address is, byte-for-byte, 'the mouse'. GFSK modulation, 2.4 GHz, ~1 MHz channels, 1-2 Mbit/s.

[תרשים 3: מבנה חבילת-ה-ESB על האוויר - Preamble, Address, PCF, Payload ו-CRC]

שתי נקודות הופכות את זה לפגיע. ראשית, ESB הוא חסר חיבור: הכתובת היא כל האימות שקיים - כל חבילה תקינה שנושאת את כתובת הדונגל תתקבל, ולא חשוב מי שידר אותה. שנית, כברירת מחדל המקלט מאשר (ACK) כל חבילה שקיבל, והמשדר חוזר ומשדר את אותה חבילה עד שמתקבל אישור. הפרט הקטן הזה - השידורים-החוזרים - יחזור להטריד אותנו בהמשך, כשהזרקה אגרסיבית תיראה למקלט כמו הצפה.



ולמה בכלל אין הצפנה? בגלל latency. הצפנה עולה זמן עיבוד, ובעכבר גיימינג כל מילישנייה קובעת - לכן היצרנים מוותרים עליה בשכבת התנועה. אגב, זו בדיוק הסיבה ששחקני-פרו רבים משחקים עם עכבר אלחוטי אך דווקא עם מקלדת חוטית: הם רודפים אחרי כל פיסת-latency, והעכבר האלחוטי המודרני מספיק מהיר. את תעבורת המקלדת תמיד מצפינים- כיוון שעלולות להשלח מהמקלדת 800מאות, אז שחקני פרו לא יכולים לאפשר לעצמם להשתמש בזה. הבהרה חשובה: במאמר הזה התמקדתי דווקא ב-Unifying Logitech, ולא במקרה - זהו פרוטוקול נפוץ מאוד, מתועד היטב במחקרי MouseJack, והכי חשוב - ה-payload שלו גלוי ולא מאומת. הוא המקרה-הנקי שבו הרעיון עובד במלואו, ולכן הבחירה הטבעית להוכחת-הקונספט (ה-RE של פרוטוקול סגור כמו HyperSpeed Razer מגיע בהמשך, כדוגמה לשיטה).

מילה על Whitening

Data Whitening זה ערבול של הביטים ממש רגע לפני שהרדיו שולח אותם לאוויר. המקלט בצד השני לא מחובר למשדר עם כבל, אז אין לו שעון חיצוני שיגיד לו בדיוק באיזו מילי-שנייה לקרוא כל ביט. כדי לא לפספס את הקצב, הוא מקשיב לאות שבאוויר ותופס את התזמון לפי הקפיצות המהירות בין אפס לאחד.

הבעיה מתחילה כשיש במידע רצף ארוך של אפסים. מבחינת הרדיו זה בעצם שקט מוחלט, קו ישר בלי שום קפיצות או שינויים. כשיש שקט כזה, למקלט אין במה להיאחז, השעון הפנימי שלו מתחיל לזייף ולהיסחף, והוא מאבד סנכרון ולא יודע מתי לקרוא את הביט הבא.

כדי למנוע את זה, המשדר מערבב את המידע עם תבנית ביטים קבועה שמלאה בשינויים. הערבול שובר את השקט, מכריח את האות לקפוץ כל הזמן, וכך המקלט נשאר מסונכרן על הקצב בלי בעיה.

למרות שהמידע מתערבל ונראה במבט ראשון כמו ג'בריש, זו ממש לא הצפנה. אין פה סיסמה, מפתח סודי או ניסיון להסתיר משהו. התבנית שמערבבים איתה גלויה וכתובה בספר ההוראות של השבב. כל מי שקולט את השידור ומפעיל עליו את אותה פעולת ערבול בדיוק, מקבל את המידע המקורי בחזרה תוך שנייה. הייתי צריך לבדוק האם יש Whitening, ובאופן כללי- איך נראית התקשורת.

איך בכלל מוצאים את הכתובת

קודם כל - כתובת של מה? כל עכבר והדונגל שלו חולקים כתובת ייחודית: מעין שם פרטי שסוכם ביניהם בזמן הצימוד, שרק הם מכירים, והדונגל מקשיב אך ורק לה. כדי להתחזות לעכבר, חייבים לגלות את הכתובת הזו - והיא לא מתפרסמת בשום מקום, אז צריך ללכוד אותה מהאוויר. הבעיה: שבבי ה-RF לא באמת יודעים להקשיב להכול (promiscuous mode). הפתרון החכם של MouseJack: מגדירים לרדיו כתובת קצרה ומזויפת (למשל שני בייטים של 0xAA) ומכבים את בדיקת ה-CRC, כך שהרדיו נדלק כמעט על כל שידור שעובר ומעביר לנו בייטים גולמיים. ואז, בתוכנה, מחליקים את הביטים הצידה שוב ושוב (shifting-bit) ובודקים מתי ה-CRC16 של ESB פתאום מסתדר - וברגע שהוא מסתדר, מצאנו פריים אמיתי, ובתוכו את הכתובת. פרט אחרון שמסבך: הביט החמישי של הכתובת קיים רק על האוויר. ה-USB לעולם לא חושף אותו, אז אי-אפשר פשוט לשאול את הדונגל בנימוס מי מדבר איתך?

החומרה: Crazyradio, nRF52840 ומגבר-החזית

הרדיו-התוקף הוא Crazyradio 2.0 של Bitcraze - דונגל-USB סביב nRF52840 Nordic (אותו SoC שמריץ המון עכברים ומקלדות אלחוטיים, ולכן אותו שבב יכול גם להיות עכבר) יחד עם מגבר-חזית nRF21540 (Module End-Front: מגבר-הספק לשידור ומגבר-רעש-נמוך לקליטה). כדי להתחזות לבייט לעכבר ספציפי לא מספיק ה-API המקורי - צריך שליטה מלאה על ה-framing, אז צרבנו firmware משלנו (פורט של LOGITacker אל ה-nRF52840). מכשולים אמיתיים בדרך: על Windows הרדיו לא זוהה עד שקושר לו דרייבר WinUSB ב-Zadig. ה-SDK של Nordic התקמפל רק מול toolchain ספציפי. ואין דיבאגר על הלוח, אז כל צריבה עברה דרך UF2 bootloader. פרט קטן שעלה שעות: אחרי כל צריבה חייבים לנתק ולחבר פיזית את הדונגל, אחרת ה-firmware הישן ממשיך לרוץ ונראה כאילו הצריבה נכשלה. אלה בדיוק הדברים שאף מודל AI לא יגלה לך.



[תמונה 1: CrazyRadio 2.0 שעבר שנתיים מאתגרות]

דוח-העכבר של UnifyingLogitech, בייט-אחר-בייט

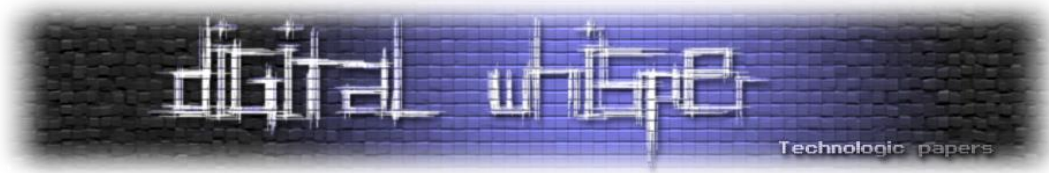
ה-payload של עכבר Unifying Logitech הוא עשרה בייטים, עם checksum בסוף שמאפס את סכום-הפריים (8 סיביות). הכתובת היא כל מה שמפריד בין העכבר למישהו אחר, ולכן ברגע שיש לנו אותה, הזרקה של אותם עשרה בייטים היא - מבחינת הדונגל - פשוט עוד דיווח תנועה מהעכבר. הקלט המוזרק מתמזג אדיטיבית עם היד האמיתית כבר בדונגל: יש עכבר אחד, שלעיתים זז מעט יותר ממה שהיד עשתה.

Logitech Unifying mouse report (10 bytes) | X,Y = 12-bit deltas packed across bytes 4..6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
dev idx	0xC2	btn	--	X lo	Ylo Xhi	Y hi	0	0	cksum

[תרשים 4: פריסת עשרת הבייטים של דוח-Unifying - דלתות ה-Y/X ארזות בבייטים 4-6]

על עכבר Logitech M720 (עכבר-Unifying נפוץ) מעל מקלט Unifying, זה עבד מקצה-לקצה. הסריקה הפרומיסקואית שתיארנו מצאה את הכתובת המלאה על האוויר. אבל למצוא את הכתובת זה רק חצי - עדיין צריך לדעת איפה בתוך עשרת-הבייטים יושבות דלתות ה-Y/X. אצל ה-M720 גילינו שהן יושבות דווקא בבייטים 4-6 (עם בייט מת ב-3), ולא במקום הספרותי המקובל. גיליתי זאת בדרך הבאה: הזרקתי



ערך ידוע לכל בייט בתורו, תוך קריאה בזמן-אמת של מיקום הסמן ממערכת ההפעלה, ובדקתי איזה בייט מזיז את הסמן ולאיזה כיוון ובכמה. ככה מיפיתי את כל פורמט הדוח אמפירית, בלי מפרט וללא ניחושים.

```
#Unifying inject frame (source: github.com/Oreldm/crazyradio-esb-inject)
pl.data[0] = 0x00. // device index
pl.data[1] = 0xC2. // mouse report type
pl.data[2] = 0x00. // buttons
pl.data[3] = 0x00. // unused (the byte the textbook gets wrong)
pl.data[4] = ux & 0xFF. // X low 8
pl.data[5] = ((uy & 0x0F) << 4) | ((ux>>8)&0x0F). // Y-lo nibble | X-hi nibble
pl.data[6] = (uy >> 4) & 0xFF. // Y high 8
logitacker_unifying_payload_update_checksum(pl.data, pl.length).
```

פרוטוקול סגור? מפרקים את ה-firmware ב-Ghidra

עכבר לא-מוצפן זה נחמד, אבל מה עם פרוטוקול קנייני שאין לו מפרט פומבי (למשל Razer HyperSpeed)? כאן נכנס Ghidra. אקדים ואומר- המאמר, והקוד המצורף- מתמקד בתקיפה מול Unifying Logitech. עם זאת- אני חושב שיהיה מעניין לשמוע קצת איך לפרק עכבר אחר עם פרוטוקול אחר. אבל לפני שמפרקים - צריך להשיג את ה-firmware, וזו הפתעה נעימה בפני עצמה. שרשרת- החילוף:

- ה-updater הרשמי הוא קובץ SFX NET (extracting-self) - פורקים אותו כ-ZIP רגיל.
- בתוך ה-ZIP יושב blob של resources, ובו image ה-firmware של המכשיר.
- וההפתעה: ה-image שמור כ-Intel HEX גלוי - טקסט קריא, בלי הצפנה ובלי מעטפת-חתימה סביב ה-payload.
- מפרסרים את ה-HEX ל-binary גולמי, ומזינים ל-Ghidra.

העובדה ש-firmware נשלח כ-Intel HEX גלוי הפכה את שחזור-ה-PHY לעניין של שעות במקום של מכשיר-glitching יקר. זה כשלעצמו לקח הגנתי - payload חתום ומוצפן היה מעלה משמעותית את מחיר ההנדסה לאחר. טוענים ל-Ghidra כ-Binary Raw, בוחרים Cortex:32:LE:ARM (זה M-Cortex, מערכת-פקודות 2-Thumb) וכתובת בסיס נכונה - קריטי, כי כל ה-pointer-ים ב-firmware הם אבסולוטיים, ובסיס שגוי מבטיח שאף xref לא יתיישב.

כמה מושגים לפני שנצלול, לקוראים שזו להם היכרות ראשונה עם הנדסה-לאחור:

- **דיסאסמבלר** (כמו Capstone) הופך בייטים גולמיים בחזרה לפקודות-אסמבלי קריאות. 2-Thumb היא מערכת-הפקודות הדחוסה של מעבדי M-Cortex (הפקודות באורך 2 או 4 בייטים, לא קבוע).
- **דיקומפיילר** (מה ש-Ghidra נותן מעל הדיסאסמבלר) מרים את האסמבלי צעד נוסף - לקוד דמוי-C קריא. וסריקת-resync (המתוארת מיד) היא הטריק שמאפשר לדיסאסמבלר להתאושש אחרי שנתקל בנתונים באמצע הקוד.



הבעיה: איבוד סנכרון בדיאסמבלי ליניארי

בארכיטקטורת Thumb-2 אין פקודת mov אחת שיכולה לטעון ערך של 32 סיביות. לכן, הקומפיילר "מחביא" קבועים כגושי דאטה (literal pools) ישירות בתוך הקוד.

כשמבצעים Linear Disassembly ונתקלים בגוש דאטה כזה, ה-decompiler מנסה לקרוא אותו כפקודות אסמבלי, נתקע ומאבד את הסנכרון עם גבולות הפקודות הבאות.

הפתרון: סריקת סנכרון מחדש (Resync Sweep)

Decompile בלולאה עם Capstone במצב Thumb. ברגע שהפירוק נתקע (כי פגשנו דאטה), קופצים 2 בייטים קדימה וממשיכים הלאה. כך מדלגים מעל גושי הדאטה, מסתנכרנים שוב על הקוד, וכל הגישות לרגיסטרים צפות מיד:

```
#resync Thumb sweep (source: github.com/Oreldm/crazyradio-esb-inject)
import capstone
md = capstone.Cs(capstone.CS_ARCH_ARM, capstone.CS_MODE_THUMB)
pos = 0
while pos < len(d)-3:
    got = False
    for ins in md.disasm(d[pos:], pos):
        got = True
        if ins.mnemonic.startswith(('str', 'ldr')) and '#0x5' in ins.op_str:
            print(hex(ins.address), ins.mnemonic, ins.op_str) # RADIO reg writes
            pos = ins.address + ins.size
    if not got: pos += 2 # hit data -> resync
```

אימות כתובת הבסיס של הרדיו

רגיסטרי הרדיו ב-RF52840n יושבים בכתובת בסיס 0x40001000. איך מוודאים שזו אכן כתובת הרדיו ולא פריפריאל אחר?

הקוד ב-firmware מחשב את מספר ה-interrupt (ה-IRQ) מתוך כתובת הבסיס, באמצעות חילוץ ביטים 12 עד 19:

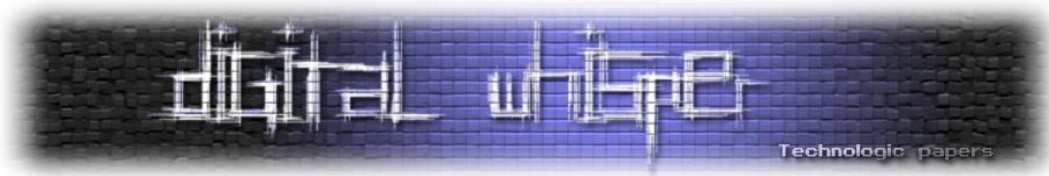
$$(0x40001000 \gg 12) \& 0xFF = 1$$

ב-RF52840n, ה-interrupt שממופה למספר 1 הוא תמיד RADIO_IRQ. התוצאה 1 מתקבלת אך ורק עבור כתובת הבסיס הזו, מה שמוכיח חד-משמעית שמדובר ברדיו.

קריאת הערכים מה-Decompiler:

אחרי שמזהים את כתובת הבסיס, פותחים את הפונקציה ב-Ghidra ורואים את הקונפיגורציה שנכתבת ישירות לרגיסטרי הרדיו:

```
#Ghidra decompiler output (fw_dongle.bin)
iVar5 = *param_1; // *param_1 = RADIO base
*(uint*)(iVar5 + 0x508) = (uint)(ushort*)(param_2 + 8); // FREQUENCY = cfg[8]
*(uint*)(*param_1+0x510) = (uint)(ushort*)(param_2 + 10); //MODE=cfg[10] -> 2 Mbit
// CRC setup elsewhere: CRCINIT=0xFFFF, poly 0x1021 (16-bit); no DATAWHITEIV write
```



- **MODE - 0x510** - קצב שידור (2Mbit) ואפנון.
- **FREQUENCY - 0x508** - ערוץ התדר.
- **הגדרות CRC**: מאותחלות ל-0xFFFF עם פולינום 0x1201 (bit CRC-16) ללא כתיבה ל-DATAWHITEIV (משמע ללא Whitening)

CodeBrowser: fw_dongle.bin		Ghidra 11.3.2 — ARM:LE32:Cortex	
Listing		Decompile: FUN_0001d0ec	
0001d11a	79 7b ldrb r3,[r7,#0x5]	undefined4 FUN_0001d0ec(int *param_1,int param_2,int param_3)	{
0001d11c	b9 7b ldrb r2,[r7,#0x6]		{
0001d11e	39 89 ldrh r1,[r7,#0x8]	int iVar5; ushort uVar2;	
0001d120	c0 f8 0c 35 str.w r3,[r0,#0x50c] ; TXPOWER	piVar7 = (int *) (DAT_0001d1e0 + ...);	
0001d124	c0 f8 04 25 str.w r2,[r0,#0x504] ; PACKETPTR	if ((char)piVar7[1] == ' '){	
0001d128	c0 f8 08 15 str.w r1,[r0,#0x508] ; FREQUENCY	*piVar7 = param_3;	
0001d12c	ba 7a ldrb r2,[r7,#0xa]	iVar5 = *param_1; // RADIO base	
0001d12e	fa 7a ldrb r1,[r7,#0xb]	uVar2 = *(ushort *) (param_2 + 8);	
0001d132	42 ea 01 21 orr.w r1,r2,r1,ls1 #0x8	*(uint *) (iVar5 + 0x50c) = ...; // TXPOWER	
0001d136	c0 f8 10 15 str.w r1,[r0,#0x510] ; MODE <-- 2 Mbit	*(uint *) (iVar5 + 0x508) = uVar2; // FREQUENCY	
0001d13a	10 68 ldr r0,[r2]	*(uint *) (*param_1 + 0x510) =	
0001d13c	4f f0 00 01 mov.w r1,#0x0	(uint) * (ushort *) (param_2 + 10); // MODE = 2 Mbit	...
		}	}

[תרשים 5: מסך CodeBrowser Ghidra אמיתי - משמאל חלון הפירוק (Listing) של הפונקציה שכותבת לרגיסטרי הרדיו, ומימין ה-decompiler שמרים אותה לקוד דמוי-C. כאן ראינו את הכתיבות ל-FREQUENCY, MODE ו-CRC]

התוצאה: HyperSpeed הוא, בשכבת-הפיזיקה, בעצם ESB רגיל - GFSK Mbit2, CRC16 עם פולינום 0x1021 והתחלה 0xFFFF, וללא whitening. ולמה בכלל בזבזנו זמן על whitening? כי הנחנו שהוא קיים. בנינו force-brute שלם שמנסה את כל 128 האפשרויות כדי להלבין בחזרה את מה שלכדנו, ושרפנו על זה שעות - ואז ה-RE של ה-firmware גילה שאין בו ולו כתיבה אחת לרגיסטר ה-whitening. כלומר, אין whitening בכלל. ה-force-brute רדף אחרי רוח-רפאים. השיעור נצרב עמוק: אל תנחש איך שכבת-הרדיו מתנהגת כשאתה יכול פשוט לקרוא אותה מתוך ה-firmware. את הסיפור המלא של ה-RE נשמור למאמר-ההמשך, אבל השיטה כאן היא הליבה.

נחזור להזריק ל-Logitech Unify Dongle

ההזרקה הראשונה עבדה והייתה בלתי-שמישה: העכבר האמיתי מגגם כל-כך שלא יכולתי להשתמש בו. שלושה גורמים נפרדים, כל אחד התגלה בניסוי ולא בקריאה. הראשון: שידור עם ACK ועם failover על כל הערוצים היה, מבחינת הפס, בדיוק כמו jamming - סערת-שידורים-חוזרים שהציפה את הלינק. עברנו ל-ACK-no עדין עם קצב מבוקר. ההזרקה השנייה: נהיגה בערוץ-הרדיו מתוך ה-thread של ה-CLI תקעה את ה-firmware לגמרי (הערוץ חייב להיות בבעלות מכונת-המצבים של הרדיו, לא להידחף מבחוץ). השלישי: כשה CrazyRadio נשאר במצב-שידור בזמן idle זה הציף את הפס ופגע בקליטה - התיקון: המגבר משדר רק בזמן פרץ-הזרקה, ובשאר הזמן יושב בקליטה שקטה. אחרי שלושת התיקונים, ההזרקה חלקה

לחלוטין, והעכבר האמיתי ממשיך לעבוד לצידה כאילו כלום. (בכנות- המעבר מעכבר שמרקד ריקוד חתונה- לכזה שזז חלק כמו חמאה היה החלק המספק ביותר בפרויקט).

כדי להבין למה ההזרקה הראשונה הרעישה כל-כך, שווה לחזור ל-ESB: כשמבקשים אישור-קבלה (ACK), המשדר חוזר ומשדר את אותה חבילה שוב ושוב עד שמתקבל אישור, ובמצב-failover אף מדלג בין ערוצים בחיפוש אחר המקלט. שילבנו את זה עם קצב-שידור גבוה, והתוצאה הייתה עשרות שידורים-חוזרים לכל דוח בודד - הצפה שמתחרה עם העכבר האמיתי על אותו מדיום, בדיוק כמו jamming. הפתרון: לוותר על האישור (ack-no - לשלוח ולשכוח), לפזר את הפרצים בזמן, ולוודא שהמקלט לא סופר את החבילות שלנו ככפילויות. פתאום, במקום קרבות ובלגנים בטווח התקשורת, יש שני משתתפים מנומסים שחולקים אותו - והסמן זז חלק.

וכדי לסבר את האוזן למה שנתיים: לפני שהגעתי להזרקה-אלחוטית נקייה, ניסיתי כמעט הכול. ניסיתי להזריק חשמל ישירות אל מגעי-המתגים של העכבר. ניסיתי לפתוח עכברים אחרים ולחפש בהם נקודת-כניסה בין החיישן ל-MCU. התחלתי לבנות עכבר שחושף API ברשת נפרדת משלו. ואפילו שקלתי לבנות משטח נע שיזיז את העכבר פיזית מלמטה. חלק מהרעיונות נכשלו בצורות יצירתיות, וכמה עכברים לא שרדו את המחקר. כל אלה - על השיטות, הכשלונות והלקחים - יחנו למאמר-ההמשך. כאן רצינו את הפתרון שעובד.

עד לכאן ההזרקה לעכבר, עכשיו נעבור לחלק ג- אלגוריתם התנועה.



[תמונה 2]

חלק ג' - הכיסוי: אלגוריתם התנועה



הקוד של חלק זה (תנועה אנושית, קינמטיקה):

<https://github.com/Oreldm/humanized-mouse-motion>

סימולטור שכתבתי שמדמה תנועת עכבר אנושית:

<https://oreldm-dev.web.app/humanized-mouse>

נניח שיש מטרה (חלק א') ואמצעי-הזרקה בלתי-נראה (חלק ב'). אם נזריק את התיקון כטלפורט בפריים אחד, או כקו-ישר במהירות קבועה, ניפול בעוגן ההתנהגותי. תנועת-יד אנושית אינה ליניארית ואינה מיידית. יש לה חתימה קינמטית מובחנת, וכדי להיות בלתי-נראים צריך לייצר אותה. התחום הזה נחקר רבות באקדמיה ואלה המודלים המרכזיים, מתוך עשרות שנות מחקר בבקרה-מוטורית:

1. חוק Fitts (1954, Fitts): זמן-התנועה גדל לוגריתמית עם היחס מרחק/גודל-מטרה: $a + b \log_2(1 + W/D)$.
2. מסלול מינימום-זעזוע (Hogan & Flash, 1985): תנועה מהירה עוקבת פרופיל-מהירות פעמוני-סימטרי - פולינום מסדר-חמישי, שהנגזרת שלו היא $v(t) \propto (t-1)^2 \cdot t \cdot 30$ (בזמן מנורמל).
3. תת-תנועות (Woodworth, 1899, al et Meyer, 1988): תנועה בליסטית שמגיעה לכ-90% מהדרך, ואז תיקון/תיקונים קטנים במשוב-ראייה.
4. רעש תלוי-אות (Wolpert & Harris, 1998): השונות המוטורית גדלה עם המהירות, $\sigma \propto |v|$. ומעליו רעד פיזיולוגי ב-Hz8-12 (Marsden & McAuley, 2000).

ועכשיו, לקצת מתמטיקה.

הנה הנוסחה:

$$s(\tau) = 10\tau^3 - 15\tau^4 + 6\tau^5$$

סתם... זאת באמת הנוסחה! (למיטיבי לכת: זה פולינום מסדר חמישי שממזער את ה"זעזוע" - הנגזרת השלישית של המיקום - כדי להבטיח שמתחילים ומסיימים בעצירה מוחלטת וללא טלטלות).

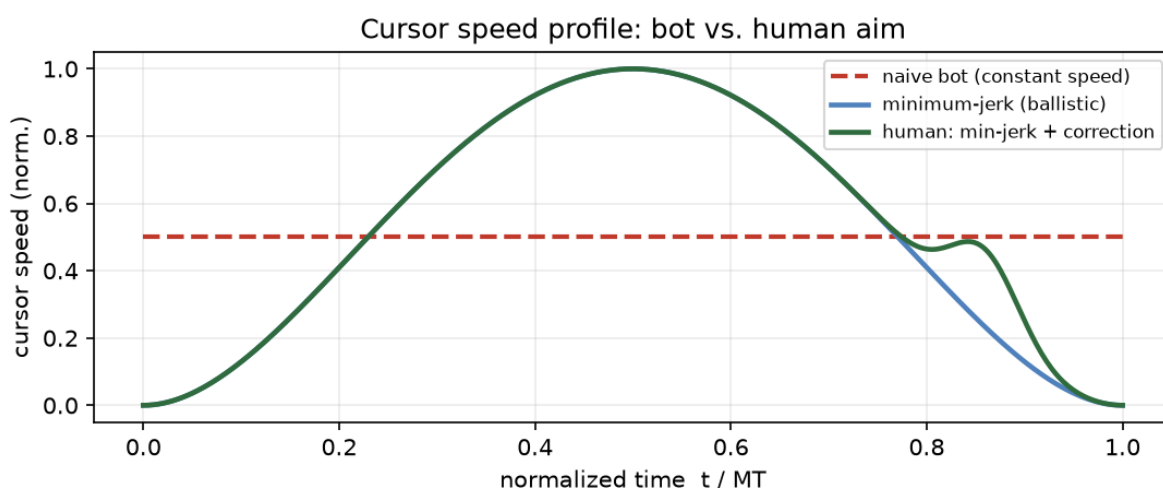
אבל מה שכל המתמטיקה הזו אומרת בתכלס הוא פשוט מאוד:

אם ניקח את הנוסחה הזו ונהפוך אותה לנוסחת מהירות, נקבל את הדבר הבא:

$$v(\tau) \propto 30\tau^2(1 - \tau)^2$$

במילים של בני אדם? הנוסחה הזו מציירת על הגרף פעמון מושלם וסימטרי.

היא מסבירה שכדי לייצר את התנועה הכי חלקה וטבעית שאפשר, חייבים להתחיל לאט, להאיץ ברכות עד לשיא המהירות בדיוק באמצע הדרך, ואז להאט באותה עדינות עד לעצירה המוחלטת. זה כל הסוד!



[תרשים 6: פרופיל-המהירות של הכוונת - בוט נאיבי (קו-ישר) מול מינימום-זעזוע ותנועה אנושית עם תיקון]

בהינתן ה-cursor speed profile היעד מחלק א', מייצרים מסלול מינימום-זעזוע שנדגם בקצב דוחות-העכבר (עד 1000Hz), מוסיפים תת-תנועת-תיקון, רעש תלוי-מהירות ורעד, ושולחים את זה כרצף של דלטות דרך ה-CrazyRadio מחלק ב'. הפרמטרים נדגמים מהתפלגות ולא נקבעים קבוע - כי קביעות היא בדיוק מה שמסגיר בוטים:

```
#Humanized motion (source: github.com/Oreldm/humanized-mouse-motion)
MT = A + B*log2(D/W + 1) # Fitts. A,B sampled per shot
tau = t/MT. s = 10*tau**3 - 15*tau**4 + 6*tau**5 # minimum-jerk position 0..1
path = outer(s, delta*0.9) # primary ballistic sub-movement
path += normal(0, K*abs(gradient(path))) # signal-dependent noise (sigma ~ |v|)
path += tremor(t, 8..12 Hz) # physiological tremor
deltas = diff(path) + corrective_submovement() # per-tick -> RF frames
```

וכאן העוקץ ההגנתי, שהוא לב-המחקר: אותם מודלים בדיוק שמייצרים אנושיות הם הכלים שבהם אנטי-ציט התנהגותי תופס בוטים - התפלגות זמני-תגובה אחידה מדי, מתאם חשוד בין הופעת-היריב לתחילת-התנועה, פרופיל-זעזוע חלק מדי, ואנטרופיה נמוכה של המסלולים לאורך זמן. לזייף אנושיות ולזהות זיוף אנושיות הם שני צדדים של אותה משוואה.

שילוב: הלולאה המלאה ו-latency budget

בזמן-אמת הלולאה היא: כרטיס-הלכידה מוסר פריים ← חותכים את מרכז-המסך ← YOLO מזהה יריב ומחזיר פיקסל-מטרה ← מתמטיקת-הכיוון ממירה לפער-cursor speed profile ← מחולל-הקינמטיקה בונה מסלול אנושי נדגם ב-1000Hz ← הרצף מוזרק כפריימי-ESB בייט-זהים אל הדוגל, שם הוא מתמזג עם היד. Latency budget: לכידה בערך פריים אחד, inference על חיתוך-המרכז כ-3ms, חישוב-הכיוון והקינמטיקה מתחת ל-1ms, וההזרקה בקצב דוח של 1ms. שום שלב לא נוגע בזיכרון-המשחק, לא מוסיף התקן-קלט, ולא מייצר תנועה גיאומטרית-מושלמת. החוזק אינו באף חלק בנפרד - YOLO סטנדרטי,



MouseJack ידוע, והמודלים הקינמטיים פורסמו - אלא בכך ששלושתם יחד מנטרלים את שלושת העוגנים בו-זמנית:

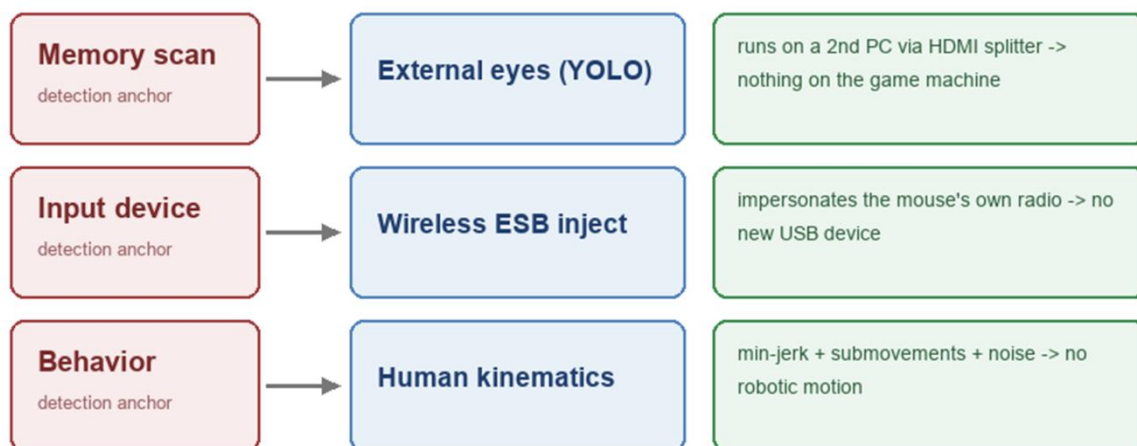
```
#The full loop (high-level skeleton - no game hooks, no weights)
while True:
    frame = capture_card.read()           # ~1 frame from the HDMI splitter
    crop = center_crop(frame, 320)       # crosshair is always centered
    det = yolo(crop, conf=0.55)[0]        # ~1-3 ms on a modest GPU
    enemy = nearest_to_crosshair(det.bboxes)
    if enemy is None: continue
    dx, dy = target_to_counts(enemy_head_px) # pixels -> angle -> mouse counts
    path = humanize(dx, dy)               # min-jerk + submovements + noise
    for (cx, cy) in path:                  # streamed at ~1 kHz
        crazyradio.inject_esb(mouse_addr, cx, cy) # byte-identical mouse frame
```

הבחנה אחרונה על התזמון: במצב שבו נרצה לבצע tracking אחרי הראש הכול חייב להסתיים בתוך מחזור-פריים, אחרת הכוונת מפגרת אחרי היריב. במצב של-flick - נעילה חד-פעמית ברגע-הלחיצה-התקציב סלחני בהרבה, כי מחשבים תנועה בודדת בלבד. בשני המקרים, הזרקת-הפריימים בקצב kHz1 היא מה שמאפשר למסלול-התנועה להיראות רציף וטבעי אצל המקלט.

למה כל כך קשה לתפוס את זה?

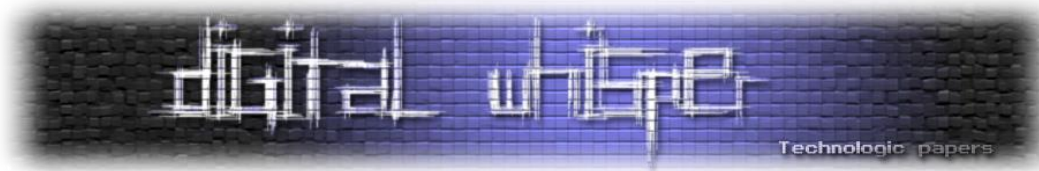
שווה לעצור ולראות איך שלושת החלקים, יחד, מכבים את שלושת העוגנים בבת-אחת - כי זו הנקודה שהופכת את הארכיטקטורה למטרידה, וגם מסבירה למה אפילו אנטי-ציט ברמת-הקרנל, או FACEIT על כל חומרתו, מתקשה כאן.

Three anti-cheat anchors -> three parts -> why each anchor goes blind



[תרשים 7: מיפוי שלושת עוגני-הזיהוי לשלושת חלקי-התקיפה, ולמה כל עוגן מתעורר]

עוגן הזיכרון עיוור כי פשוט אין מה לראות: המוח (YOLO) יושב על מחשב פיזי אחר, שמקבל את הווידאו דרך HDMI Splitter. אין תהליך זר על מחשב-המשחק, אין קריאת זיכרון, אין hook, אין DLL מוזרק. הדרייבר של האנטי-ציט סורק מכונה נקייה לחלוטין, ומוצא - כלום.



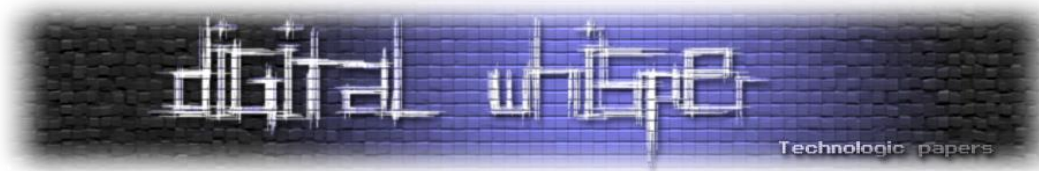
עוגן-הקלט עיוור כי אין התקן חדש: הקלט אינו מגיע דרך SendInput ולא דרך התקן-USB נוסף שאפשר לספור ולטבוע, אלא דרך שכבת-הרדיו של העכבר הקיים - מתחת למחסנית-ה-USB של המחשב, וממוזג אדיטיבית עם היד האמיתית כבר בתוך הדונגל. מבחינת המערכת יש התקן-HID אחד בלבד, החוקי, שסתם זז קצת יותר ממה שהיד עשתה.

ועוגן-ההתנהגות עיוור כי התנועה אנושית מעצם-בנייתה: מסלול מינימום-זעזוע, תת-תנועות-תיקון, רעש תלוי-מהירות ורעד. הסימן היחיד שנשאר אינו קלט זר אלא שארית סטטיסטית עדינה - וזה כבר לא משחק שמנצחים בו על המחשב המקומי, אלא בצד-השרת ולאורך אלפי אירועים. וזה, בדיוק, הנושא של המאמר הבא.

הגנה: לאן ההגנה עוברת

- ניתוח התנהגותי-סטטיסטי בצד-השרת: לא לחפש קלט זר אלא את שאריות-הקינמטיקה. לזייף התפלגות אנושית מושלמת לאורך אלפי אירועים קשה בהרבה מלזייף תנועה בודדת - וכאן ההגנה עדיין מנצחת.
- שרשרת-קלט אמינה (input trusted) ואטסטציה חומרתית: נתיב מאומת מהחיישן דרך ה-firmware אל המשחק, שחותם או מונה כל דוח - אז גם הזרקה בייט-זהה נחשפת. Logitech עצמה עושה זאת במקלדות המוצפנות שלה. היעדר-האימות בדוחות-העכבר הוא כל הפגיעות.
- הצפנה ואימות ברמת-הלינק (כמו BLE עם אבטחת-שכבת-קישור) במקום ESB חסר-חיבור.
- לזכור שהתקיפה החומרתית (HDMI Splitter + רדיו) קשה מאוד לזיהוי מקומי - כי אין לה טביעת-רגל על מחשב-המשחק. זה בדיוק מה שהופך את המישור ההתנהגותי בצד-השרת לקו-ההגנה האמיתי.
- יש עוד דרכים יותר יצירתיות- שאותן אשמור למאמר הבא ☺

ומה בעצם מודדים בצד-השרת כדי לתפוס? שאריות קינמטיקה. התפלגות זמני-התגובה (אדם רועש. בוט אחיד מדי). המתאם בין רגע-הופעת-היריב-על-המסך לבין תחילת-התנועה (חשוד כשהוא הדוק מדי). פרופיל-הזעזוע (חלק או אחיד יתר-על-המידה). ספירת תת-התנועות והתפלגותן. ואנטרופיית-המסלולים לאורך זמן. אף אחד מהסימנים האלה אינו הוכחה בפני עצמו - אבל שילובם, על-פני אלפי אירועים, הוא כבר סיפור אחר. את בניית מודל-הזיהוי הזה, ואת התוצאות שלו מול האימבוט שתואר כאן, נשמור למאמר-ההמשך. כי הרי בשביל זה בנינו אותו מלכתחילה: כדי לתפוס אותו.



כמה זה עולה, ולמה זה משנה

מילה על הנגישות, כי היא חלק בלתי-נפרד מהאיום: הרכיבים לא יקרים. Crazyradio עולה עשרות בודדות של דולרים, מפצל-HDMI וכרטיס-לכידה עולים מעט, ומחשב שני יכול להיות ישן וצנוע. אין כאן ציוד-מעבדה - יש כאן ערכה שאפשר להרכיב על שולחן המטבח. וזו בדיוק הסיבה שהאיום מטריד: הוא לא דורש גישה פיזית לקורבן, לא דורש להריץ שום דבר על מחשב-המשחק, והמחסום העיקרי הוא ידע - לא תקציב. המחסום הזה, אגב, הוא בעיקרו הרדיו: את ה-LLM ואת התנועה כל מתכנת יכול לשחזר בשבועיים, אבל את החלק החומרתי צריך באמת ללמוד. (ובדיוק בגלל שהמחסום הוא ידע ולא כסף, פיצלנו את הקוד לשלושה repositories נפרדים - אין מהם אימבוט מוכן להורדה.)

סיכום

הצגנו רעיון שלם שעובד מקצה-לקצה: אימבוט בשלושה חלקים משלימים - עיניים חיצוניות מבוססות-YOLO (עם חיתוך-מרכז שחוסך חישוב ומייצב את הכוונת) שאינן נוגעות בזיכרון, ידיים אלחוטיות שמתחזות לשכבת-הרדיו של העכבר במקום להוסיף התקן (עם צלילה ל-Ghidra כדי לפצח פרוטוקול סגור), ותנועה שנבנית ממודל קינמטי של יד. כל חלק מנטרל עוגן-זיהוי אחר, והחיבור ביניהם הוא מה שהופך את הארכיטקטורה למטרידה - ולכן ראוייה למחקר הגנתי. השורה-התחתונה אופטימית דווקא למגן: אותה מתמטיקה שמייצרת אנושיות היא גם מה שחושף אותה, וזה אחד המישורים שאליה ההגנה צריכה לעבור. מקווה שנהנתם! ☺

על הכותב

אוראל דמרי (oreldm) - אני קורא את Whisper Digital מאז הגליון הראשון ב-2009, ומרגש אותי לכתוב כאן לראשונה. בוגר יחידה 81, מהנדס תוכנה וכיום כותב אוטומציה ב-CrowdStrike. כפי שראיתם, מוכן להקריב קצת חומרה בשם המדע.

אפשר ליצור איתי קשר ב-[linkedin](https://www.linkedin.com/in/oreldm/) או באתר <http://orel.dev>.

תודה רבה לאפיק קסטיאל ולאלי קסקי על התמיכה בכתיבת המאמר!



קוד, כלים ומקורות

במכוון פיצולנו את הקוד לשלושה repositories נפרדים - אחד לכל חלק - כדי שאיש לא יקבל אימבוט מוכן מהמדף: אין אף repo יחיד שמחבר עיניים, ידיים ותנועה, וכל חלק דורש ידע וחומרה משלו. זו בחירה מכוונת של הפחתת-נזק.

תנועה אנושית (קינמטיקה):

<https://github.com/Oreldm/humanized-mouse-motion>

הזרקת-ESB ל-Crazyradio (frame + כלים):

<https://github.com/Oreldm/crazyradio-esb-inject>

זיהוי-מטרות על המסך (YOLO):

<https://github.com/Oreldm/screen-target-detection>

כלים פומביים

- Ultralytics YOLO - <https://github.com/ultralytics/ultralytics>
- MouseJack (Bastille) - <https://github.com/BastilleResearch/mousejack>
- LOGITacker - <https://github.com/RoganDawes/LOGITacker>
- Ghidra (NSA) - <https://github.com/NationalSecurityAgency/ghidra>
- Capstone (disassembler) - <https://github.com/capstone-engine/capstone>
- Crazyradio 2.0 (Bitcraze)- <https://www.bitcraze.io/products/crazyradio-2-0>

מקורות אקדמיים - קינמטיקה של תנועת-יד

- Fitts, P. M. (1954). The information capacity of the human motor system. J. Exp. Psychology, 47(6)
- Flash, T., & Hogan, N. (1985). The coordination of arm movements. J. Neuroscience, 5(7)
- Meyer, D. E., et al. (1988). Optimality in human motor performance. Psychological Review, 95(3)
- Harris, C. M., & Wolpert, D. M. (1998). Signal-dependent noise determines motor planning. Nature, 394
- McAuley, J. H., & Marsden, C. D. (2000). Physiological and pathological tremors. Brain, 123(8)