digital whisper
Technologic Papers

חבוי בגלוי מול תוכנות Anti Cheat

מאת אריאל טרי

הקדמה

כבר שנים רבות מתנהל מאבק מתמשך בין מפתחי תוכנות Anti-Cheat למשחקי מחשב לבין אלה המנסים לעקוף אותן. במהלך 'משחק החתול והעכבר' הזה, שני הצדדים פיתחו מגוון רחב של שיטות וטכניקות. כל צד מנסה להתגבר על היתרונות של הצד השני, מה שמוביל להתפתחות מתמדת בתחום. -

במאמר זה, נחקור כיצד ניתן לנצל סביבות משחק המשתמשות בקוד המהודר בשיטת JIT (Just-in-time) כדי לעקוף מנגנוני הגנה. נתחיל בהסבר בסיסי על אופן פעולתן של תוכנות Anti-Cheat נפוצות, כולל סוגי החסימות שהן מיישמות.

למשל - ברוב פתרונות ההגנה כיום, מדוע נדרש מאיתנו שימוש בקוד אשר רץ ברמת ה-Kernel של מערכת ההפעלה, על מנת לכתוב ולקרוא מזיכרון משחק המחשב?

לאחר מכן, נתמקד במשחק מוכר-Escape from Tarkov, המשתמש בתוכנת ה-Anti Cheat בשם Battleye בתור שכבת הגנה למשחק. המשחק מהודר באמצעות מנוע JIT - מונח שנבין מה הוא אומר בהמשך, ומדוע ניצול הדבר יכול להקשות על אותה תוכנת הגנה לאתר ולדווח על קוד זדוני, אשר עוזר לרמאים בקצה להמשיך מבלי שיעלו ברדאר. בנוסף, חשוב לציין כי כל המידע המופיע במאמר זה נועד למטרות למידה בלבד, ואיני אחראי על כל שימוש לרעה בו.

מהי תוכנת Anti Cheat

תוכנות נגד רמאות הן תוכנות (ולעיתים גם דרייברים, נגע בזה מעט בהמשך), המותקנות ורצות לצד משחק המחשב בצד הלקוח. בנוסף, הן לרוב כוללות גם מערכות ניטור בצד השרת של ספק המשחק או ספק התוכנה המגנה, אשר מזהות התנהגות חשודה של השחקן. ניתן לחשוב עליהן כמעין "אנטי-וירוס" שמיועדת במיוחד עבור משחקי מחשב. עם זאת, בהשוואה לתוכנות אנטי-וירוס ו- EDR (Endpoint Detection and Response), תחום הניטור של תוכנות נגד רמאות מצומצם יותר, כיוון שהן יודעות בדיוק אילו פעולות שייכות למשחק שעליו הן מגנות. בעוד שתוכנות אנטי-וירוס צריכות להתמודד עם זיהוי תחת כל תהליך במערכת ההפעלה.

סוגי תוכנות רמאות

קיימים שני סוגי רמאויות, במאמר זה נתייחס לסוג השני:

- **חיצוני (External):** תוכנה אשר אך ורק כותבת/קוראת מזיכרון המשחק, לרוב ללא עריכת קוד המשחק.
- **פנימי (Internal):** קובץ DLL הטעון במרחב הזיכרון של המשחק, לרוב יערוך ישירות את פונקציות המשחק.

דרכים בסיסיות למנוע רמאות

עבור כותבי תוכנות ההגנה, קיימות מספר דרכים למנוע, לסכל ולדווח על תוכנות רמאות אשר אינן מתוחכמות. דוגמאות לכמה דרכים יכללו בדיקה מחזורית של מפת הזיכרון של תהליך המשחק, ווידוא שלא קיים בו שום קוד הרצה המסומן כ-Executable, אשר אינו רץ בדרך כלל במשחק. ניתן גם להשוות את זיכרון ה-Read only sections של המשחק בזמן ההרצה, אל מול המידע שכתוב בקבצי המשחק השמורים בכונן הדיסק, בכדי לוודא ששום גוף צד שלישי לא ערך את קוד המקור של המשחק.

דרך נפוצה בהן משתמשות חברות ההגנה (כגון Battleye) אשר מריצות דרייבר (בפשטות - פיסת קוד אשר רצה ברמת ה-Kernel של מערכת ההפעלה), היא לבצע קריאה לפונקציה הנמצאת ברמת ה-Kernel בשם-ObRegisterCallbacks. אותה הפונקציה, כפי שכתוב בתיעוד המופיע באתר-MSDN, מבצעת הוספה של פונקציית Callback עבור פעולות המבוצעות על אובייקטים כגון תהליכים, Threads, ו-Handle-ים:

Learn / Windows / Windows Drivers / API / Kernel / Wdm.h /

ObRegisterCallbacks function (wdm.h)

Article • 02/25/2022 Feedback

In this article

- Syntax
- Parameters
- Return value
- Remarks

Show 2 more

The **ObRegisterCallbacks** routine registers a list of callback routines for thread, process, and desktop handle operations.

Syntax

C++ Copy

```
NTSTATUS ObRegisterCallbacks(
  [in]  POB_CALLBACK_REGISTRATION CallbackRegistration,
  [out] PVOID                     *RegistrationHandle
);
```

כלומר, פעולות המבוצעות על אותם אובייקטים, יגרמו לקריאה של פונקציית ה-Callback אותה הגדיר הדרייבר אשר מחפש התנהגות חשודה. יכולת זו בהחלט נחשבת כחזקה, משום שהיא מאפשרת הפלה מלאה של בקשות ליצירת Handle מכל תהליך שרץ ברמת ה-Usermode. כך יכולים המגנים לסנן בקלות בקשות ליצירת Handle המאפשר גישה לכתיבה וקריאת הזיכרון למשחק המחשב, במידה והעבירו אחד מהפרמטרים הבאים: **PROCESS_VM_OPERATION**, **PROCESS_VM_READ**, **PROCESS_VM_WRITE**.

לכן, כתיבה וקריאה לזיכרון המשחק מרמת ה-Usermode, אינה אפשרית (לפחות ללא ניצול חולשה אזוטרית כלשהי). ובכן, מה ניתן לעשות בנושא? הרי בסופו של יום כתוקפים, נאלץ לכתוב זיכרון שרירותי למשחק, בשביל להריץ קוד זדוני. בגלל סיבה זו, נצטרך להשתמש בדרייבר אשר רץ ברמת ה-Kernel, ואז תהיה לנו גישה מלאה לטווח זיכרון המשחק, ונוכל לכתוב ולקרוא ממנו בחופשיות.

בשביל לפתור בדיוק את הבעיה הזו, החלטתי ללכת במסע לאחר חולשה בדרייבר מגורם צד שלישי, בכדי לממש את שיטת BYOVD (Bring your own vulnerable driver). על אותה שיטה כבר ניתן לקרוא במאמר "מעקף DSE בעזרת שיטת BYOVD" מאת שי גילת, ולכן אני בוחר שלא להעמיק בה מעבר לנדרש, אך כן אסביר בקצרה כיצד מבצעים אותה, וכיצד היא תורמת לנו.

ניצול דרייבר בעל חולשה

השיטה BYOVD מתבססת על טעינה של דרייבר בעל חולשה למערכת ההפעלה. דבר זה מאופשר על ידי מערכת ההפעלה ווינדוס, במידה והדרייבר הינו חתום על ידי גורם מוסמך, גם אם מדובר בדרייבר צד שלישי שלא נוצר על ידי החברה מייקרוסופט. לאחר שהדרייבר טעון, ניתן לנצל את אותה חולשה. כמובן שהניצול עצמו תלוי בסוג החולשה, אך ברוב המקרים יהיה מדובר בחולשת Buffer overflow כלשהי, או פשוט פונקציה אשר מבצעת גישה ישירה לזיכרון המחשב עם כתובת שמספקים לה, וכותבת או קוראת מידע בתוך גוף הפונקציה. בכל מקרה, נצטרך לבצע שליחת בקשה לפונקציית-IOCTL אשר קיימת בדרייבר הפגיע, נעשה זאת ברמת ה-Usermode.

לאחר שמצאנו דרייבר פגיע, נוכל לבצע תקשורת בעזרת הפונקציה **DeviceIoControl,** הנמצאת בספרייה kernel32.dll.

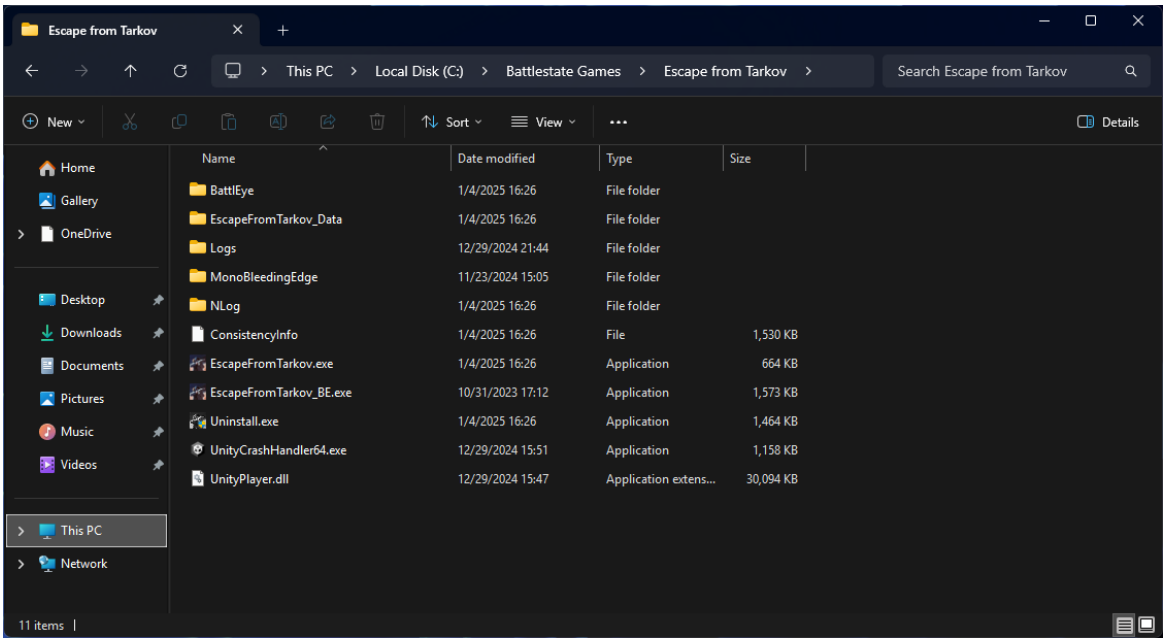
אחסוך מן המאמר את הדרייבר החולשתי אותו מצאתי וניצלתי, משום שדיווחתי על כך לחברה שיצרה אותו. בנוסף, קיימים מאות דרייברים דומים בעלי חולשות דומות באופן פומבי, אותם ניתן למצוא בקלות רבה באתרים כמו loldrivers, וגם פרויקטים באתר GitHub כמו KDMapper.

כעת יש לנו גישה ישירה לכתוב ולקרוא בחופשיות מזיכרונו של תהליך המשחק. בגלל שנרצה להריץ קוד אשר "מוחבא" תחת המשחק עצמו, ומבצע פעולות זדוניות כלשהן, לא נסתפק רק בזאת. משום שאנחנו צריכים להבין היכן כדאי לאחסן את הקוד הזדוני שלנו, מבלי להיתפס בסביבת המשחק, וכיצד נריץ אותו מלכתחילה.

תקיפת המשחק Escape from Tarkov

כפי שציינתי בהקדמה של המאמר, לאורך המאמר נתמקד במשחק-Escape from Tarkov. אותו משחק יצא לאור בשנת 2017 ונוצר על ידי החברה Battlestate Games. המשחק הוא בין הלקוחות הגדולים, אם לא הכי גדול, של חברת ההגנה Battleye.

נוכל למצוא את קבצי המשחק בתיקייה C:\Battlestate Games\Escape from Tarkov. תמיד חשוב לבדוק את קבצי המשחק בתור הצעד הראשון, בשביל להבין מול מה אנחנו עובדים, כלומר, באיזה מנוע משחק המפתחים בחרו להשתמש. לכתוב קוד זדוני עבור משחק שמשתמש במנוע כמו Source engine של החברה Valve, שקיים עבורו תיעוד רב, או Godot שהינו מנוע שכל הקוד שלו פומבי וזמין לקריאה, יהיה הרבה יותר קל. זאת משום שיש מידע רב על אותם מנועים באינטרנט, לעומת מנוע שאינו שכיח, אותו אולי אפילו מפתחי המשחק פיתחו בעצמם:



במידה ויש לכם מעט הכירות עם מנועי משחקי מחשב, או אפילו בכללי יצא לכם להריץ משחקים חינמיים בטלפון או במחשב, קשה מאוד לפספס את הקבצים UnityPlayer.dll, ו-UnityCrashHandler64.exe אשר מעידים על כך ש-Unity הוא המנוע בו השתמשו מפתחי המשחק. בשביל להבין באיזה שפה נכתב המשחק, נוכל להתייעץ עם גוגל, ומהר מאוד נגלה שכאשר יוצרים פרויקט חדש של משחק בעזרת המנוע, הוא מאפשר בחירה בין שתי אופציות, משחק אשר משתמש במנוע בשם mono, או ב-il2cpp. לפי התיקייה בשם MonoBleedingEdge, ניתן למצוא שיוך לאופציה הראשונה של מנוע mono. אם נחפש על כך באינטרנט, נגלה שהוא מנוע קוד פתוח, המאפשר סביבה לקימפול והרצה של קוד #C. מפתחי מנוע המשחקים Unity בחרו להשתמש ב-Mono כדי לאפשר למפתחי משחקים לכתוב את קוד המשחק שלהם בשפת #C.

לעומת זאת, il2cpp הינה משתמשת בשיטת ההידור Ahead of Time, אשר תעביר את כל הקוד הליך קומפילציה עוד כשמפתח המשחק החליט להריץ את הקוד, ולא מבוצע שום הליך כזה בתוכנה אותה מריץ משתמש הקצה.

מידע זה הוא מאוד בסיסי, אך הכרחי עבור תוקף, משום שכעת ניתן להבין כי עבודה אל מול קוד המשחק תהיה פשוטה בעזרת כלים כמו dnSpy אשר מאפשרים דיבוג, עריכת קוד וקריאת קוד עבור בינארים הנועדו להרצת קוד DOTNET, דבר זה כולל את #C.

מטרת המאמר אינה להדריך כיצד לכתוב קוד רמאות זדוני, אלא להתמקד בהצגת טכניקות להחבאת קוד בזיכרון המשחק, תוך שימוש בסביבת Mono שסקרנו קודם. נבחן כיצד טכניקות אלו מתמודדות עם פתרונות ההגנה הנפוצים במשחקים, אך המאמר לא יכלול הוראות מפורטות לכתיבת קוד רמאות, שכן מידע כזה כבר זמין בשפע באינטרנט.

עם זאת, אותו הידע יכול לשמש אותנו להחבאת הקוד הזדוני שלנו במשחק. כיצד? ובכן, מנוע mono הינו מנוע הפועל בצורת ההידור JIT (Just in Time). כלומר, כאשר המפתח מוסיף קוד חדש למשחק, הוא נבנה לשפת ביניים כלשהי (IL), אשר עוברת הידור סופי לקוד הנקרא Native, במקרה שלנו - Assembly מסוג x86. קוד זה יקומפל אך ורק בזמן הרצה, ולשיטה זו קיימים יתרונות רבים, למשל - הקוד היחידי שצריך להיות מקומפל ונוכח בזיכרון, הוא קוד שמשומש. קוד שלא שומש זמן מה, יכול להיות "משוחרר" מן הזיכרון במידה ואינו נחוץ. יתרון חשוב ביותר הוא, בגלל שהקוד עובר הליך קומפילציה רק בזמן הרצת התוכנה, התוכנה אשר מהדרת את הקוד, יכולה לזהות Feature-ים מיוחדים במחשבי קצה של משתמשי התוכנה, ולבצע אופטימיזציות שונות עבור אותו מעבד, או רכיב חומרתי כלשהו. לכן נגיע למצב בו נוצר קוד מסוים עבור מחשב א', אך קוד שונה לחלוטין עבור מחשב ב', בהינתן חומרה שונה בין שני המחשבים.

ניצול סביבות הרצה המשתמשות ב-JIT

יכול להיות שחלקכם כבר הבחינו בבעייתיות שבדבר, כפי שצוין בתחילת המאמר, מוצרי הגנה רבים יבדקו שהקוד שאותו מריץ המשחק הינו באמת משויך לקבצים השמורים בכונן הדיסק שאותו מריץ תהליך המשחק. אך ברגע שהמשחק רץ תחת מנוע JIT כלשהו, המשימה נהפכת לקשה במיוחד, משום שהקוד שרץ יכול להיות להשתנות בין מחשבים שונים, ובין גרסאות המנוע, במידה ומעודכן יום אחד.

חשוב להבין שאין גודל קבוע של קוד Native שמערכת ההגנה יכולה להשתמש כנקודת ייחוס לזיהוי חריגות. זאת מכיוון שהקוד Native שקיים בזיכרון הוא רק הקוד שהמנוע נדרש להריץ כחלק מתהליך הקומפילציה. תהליך זה מתרחש באופן דינמי: רק הקוד הנחוץ עובר קומפילציה, וזאת רק בפעם הראשונה שהוא נקרא לשימוש. תהליך הקומפילציה הזה מתבצע באמצעות הפונקציה mono_compile_method. פונקציה זו מוגדרת בטבלת ה-Export Address Table (EAT) של הקובץ mono-2.0-bdwgc.dll, שהוא חלק מרכזי במנוע Mono. כתוצאה מכך, גודל הקוד Native בזיכרון משתנה בהתאם לצרכי התוכנית, מה שמקשה על מערכות הגנה לזהות קוד זדוני על סמך גודל בלבד.

לכן, בגלל שעבור מוצר ההגנה בו משתמש המשחק, יצירת בדיקה אשר תבדוק אם קוד מסוים משויך ליוצרי המשחק או לא, מהווה אתגר קשה במיוחד לפתור בגלל סביבת ההרצה. נוכל לנצל זאת לטובתנו דרך אלקוצי זיכרון אשר נראים זהים לאלו שמנוע ה-JIT אוגר בהן קוד משחק.

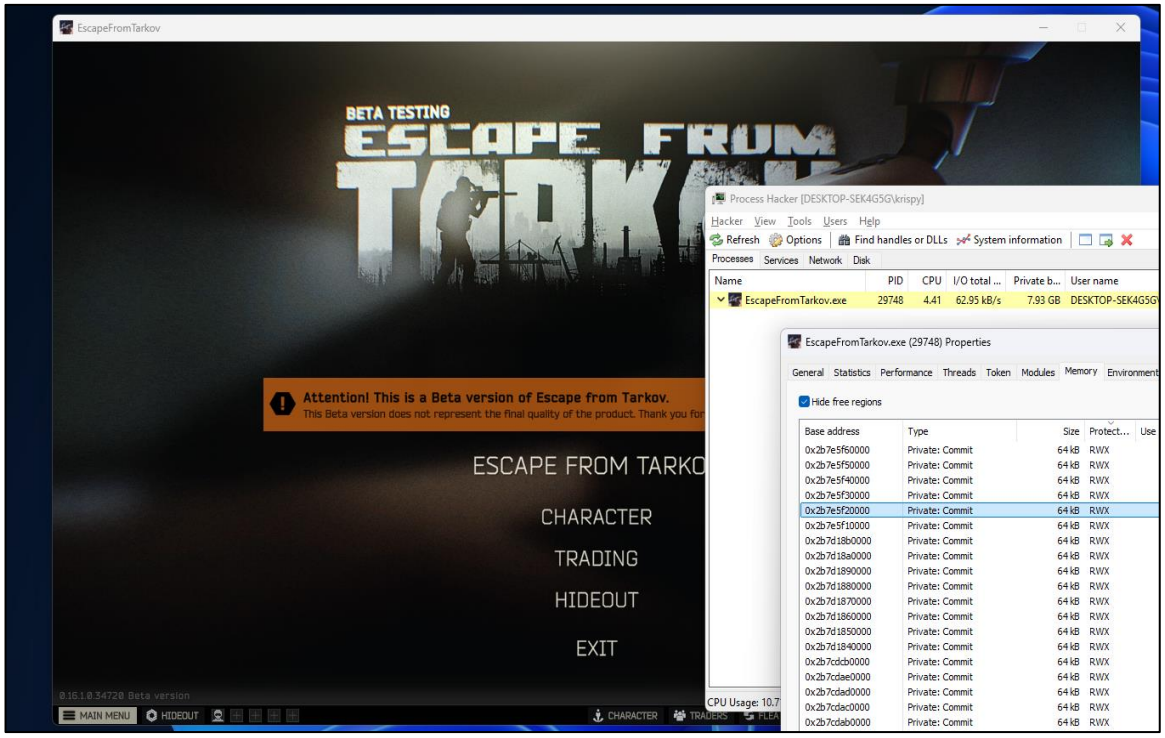
כל התהליך ידרוש מאיתנו, התוקפים, שלושה שלבים:

1. קריאה מזיכרון המשחק, בכדי להבין לאן ניתן לכתוב את ה-Shellcode שיאלקץ זיכרון חדש, אשר מוחבא כזיכרון שנראה אמיתי.
2. קריאה של המיקום בו הקוד שלנו אילקץ זיכרון, וכתיבה של ה-DLL הזדוני שלנו לאותה הכתובת.
3. הרצת קוד נוספת של ה-Entry point השמור ב-DLL הזדוני שלנו, בכדי שהוא ירוץ. נקודות בונוס במידה ומדובר ב-Thread שלא יאט שום דבר במשחק או יגרום להתנהגות חריגה :)

שלב ראשון - לאן כותבים קוד?

הדבר הראשון שעלינו לעשות הוא להריץ קוד המבצע אלקוץ זיכרון שנראה תואם לזה של המשחק. אך לאן נכתוב את אותו הקוד? ובכן, בוא נריץ את המשחק ונבין כיצד ואיפה הוא שומר קוד שעבר הליך קומפילציה.

קודם כל, נצטרך להריץ את המשחק ללא Battleye, בכדי שנוכל להריץ כלים כמו Process hacker, אשר יאפשרו לנו גישה למפת הזיכרון של המשחק. בשביל להריץ את המשחק ללא שום תוכנת הגנה, השתמשתי בכלי BattlEyent שזמין בפורום UnknownCheats וב-Github. כל מה שנדרש הוא הרצה של הכלי, ולאחר מכן הרצה רגילה של המשחק, הכלי יוצר Hook פשוט על הפונקציה שמבצעת את תהליך האתחול של Battleye. לכן נוכל להתחבר רק ללובי משחק מקומי במידה ואנחנו משתמשים באותו הכלי, אך השיטה עדיין משמשת כדרך נהדרת עבור בדיקות כמו זו שאנחנו רוצים לבצע כרגע:



כעת יש לנו גישה מלאה לזיכרון המשחק וכל הפרטים עליו, משום שהוא רץ ללא ההגבלות ש-Battleye מציב לנו. אם נגלול בטבלת ה-Memory בכלי Process hacker, נשים לב כי יש המון מיפויי זיכרון בגודל 0x10000 (64KB), אותם מיפויים בעלי הרשאות RWX (Read Write Execute), דבר זה הגיוני משום שהקוד ה-Native שסביבת הקימפול של-JIT יצרה הוא מאוד דינאמי, וקוד ישן יכול להימחק, קוד חדש יכול להתווסף וכו'. לכן ניתנות הרשאות גבוהות לאלקוצי הזיכרון ששומרות קוד Native.

כל האלקוצים נראים כמו מקום טוב לכתוב אליו את ה-Shellcode שלנו, שיבצע אלקוצי זיכרון חדשים המסווים את עצמם בתור אותם קוד מהמשחק כפי שרואים בתמונה.

כעת, כשבחרנו מיקום לכתוב אליו, נשאר רק להבין איך אנחנו יכולים למצוא את אותן כתובות בעצמנו. ובכן, בגלל שאותן כתובות מנוהלות על ידי המנוע Mono, סביר להניח כי מקום טוב להתחיל בו יהיה קובץ הספרייה הגדול שמנהל חלק נרחב מניהול הזיכרון והקימפול של Mono אותו ציינתי לפני-mono-2.0 bdwgc.dll.

למרות ש-Mono מוגדר כפרויקט קוד פתוח, הגרסה בה משתמשים מפתחיה של מנוע Unity היא גרסה אשר עברה שינויים על ידיהם. את אותם שינויים אני לא מכיר, ולכן אעדיף שלא להסתמך על הקוד הפתוח, אלא לפרוס את הבינארי עם כלי כמו Ghidra או IDA, ולהשתמש בקבצי ה-Symbol שמפתחי Unity משאירים בשרת שלהם. לאחר מעט מחקר שמתי לב שהשימוש במשתנה הגלובאלי mono_root_domain הוא נפוץ ביותר, והוא בעצם אוגר מידע רב לגבי החלקים שמנהלים את המנוע, למשל איפה שזיכרון נשמר:

```
                    mono_root_domain                          XREF[181]:   mono_runtime_quit:18009c4ae(R),
                                                                           mono_domain_create_appdomain_int...
                                                                           mono_domain_try_unload:1800a474c...
                                                                           mono_get_root_domain:1800a71b0(R...
                                                                           mono_mlist_alloc:1801baa65(R),
                                                                           mono_mlist_prepend:1801baf4d(R),
                                                                           mono_mlist_append:1801bb3ad(R),
                                                                           mono_class_create_runtime_vtable...
                                                                           mono_unhandled_exception_checked...
                                                                           create_internal_thread_object:18...
                                                                           mono_thread_attach_internal:1801...
                                                                           mono_thread_current:180201a0d(R),
                                                                           mono_threads_attach_coop_interna...
                                                                           mono_threads_attach_coop_interna...
                                                                           mono_gc_alloc_vector:18026f7f8(R...
                                                                           mono_unity_thread_fast_detach:18...
                                                                           mono_unity_thread_fast_detach:18...
                                                                           mono_unity_root_domain_mempool_c...
                                                                           mono_jit_thread_attach:18028ec97...
                                                                           mono_jit_thread_attach:18028ece3...
                                                                           [more]
180751020 00 00 00        _MonoDom...  00000000
          00 00 00
          00 00
```

המשתנה מצביע למבנה נתונים בשם-"_MonoDomain", וניתן למצוא אותו ב-Offset 0x751020 מתחילת מיפוי זיכרון התוכנה. כאשר גוללים דרך התוכן שמכיל מבנה הנתונים אליו המשתנה מצביע, ניתן לראות שהמידע האחרון שנשמר הוא מצביע בשם-memory_manager, ב-Offset 0x248:

Structure Editor - _MonoDomain (mono-2.0-bdwgc.dll) [CodeBrowser: mono:/mono-2.0-bdwgc.dll]

Edit Help

Structure Editor - _MonoDomain (mono-2.0-bdwgc.dll)

Offset	Length	Mnemonic	DataType	Name
0xa8	0x8	_MonoAssembly *	_MonoAssembly *	entry_assembly
0xb0	0x8	char *	char *	friendly_name
0xb8	0x8	_GHashTable *	_GHashTable *	proxy_vtable_hash
0xc0	0x28	_MonoInternalHashTable	_MonoInternalHashTable	jit_code_hash
0xe8	0x30	mono_mutex_t	mono_mutex_t	jit_code_hash_lock
0x118	0x4	int	int	num_jit_info_table_duplicates
0x120	0x8	_MonoJitInfoTable *	_MonoJitInfoTable *	jit_info_table
0x128	0x8	_MonoJitInfoTable *	_MonoJitInfoTable *	aot_modules
0x130	0x8	_GSList *	_GSList *	jit_info_free_queue
0x138	0x8	char **	char **	search_path
0x150	0x8	_MonoMethod *	_MonoMethod *	create_proxy_for_type_method
0x158	0x8	_MonoMethod *	_MonoMethod *	private_invoke_method
0x160	0x8	_GHashTable *	_GHashTable *	special_static_fields
0x208	0x4	int	int	assembly_bindings_parsed
0x210	0x8	_MonoImage *	_MonoImage *	socket_assembly
0x218	0x8	_MonoClass *	_MonoClass *	sockaddr_class
0x220	0x8	_MonoClassField *	_MonoClassField *	sockaddr_data_field
0x228	0x8	_MonoClassField *	_MonoClassField *	sockaddr_data_length_field
0x230	0x8	_GHashTable *	_GHashTable *	ftnptrs_hash
0x238	0x8	_GHashTable *	_GHashTable *	method_to_dyn_method
0x240	0x4	int	int	throw_unobserved_task_except...
0x244	0x4	uint	uint	execution_context_field_offset
0x248	0x8	_MonoMemoryManager *	_MonoMemoryManager *	memory_manager

זה כבר כיוון עבור מה שאנחנו מחפשים! בוא נראה מה מבנה הנתונים "_MonoMemoryManager" מכיל בתוכו:

Structure Editor - _MonoMemoryManager (mono-2.0-bdwgc.dll) [CodeBrowser: mono:/mono-2.0-bdwgc.dll]

Edit Help

Structure Editor - _MonoMemoryManager (mono-2.0-bdwgc.dll)

Offset	Length	Mnemonic	DataType	Name
0x0	0x8	_MonoDomain *	_MonoDomain *	domain
0x8	0x4	int	int	collectible
0xc	0x4	int	int	is_generic
0x10	0x4	int	int	freeing
0x18	0x30	_MonoCoopMutex	_MonoCoopMutex	lock
0x48	0x8	_MonoMemPool *	_MonoMemPool *	mp
0x50	0x8	_MonoCodeManager *	_MonoCodeManager *	code_mp
0x58	0x8	_GPtrArray *	_GPtrArray *	class_vtable_array
0x60	0x8	_MonoGHashTable *	_MonoGHashTable *	type_hash
0x68	0x8	_MonoConcGHashTable *	_MonoConcGHashTable *	refobject_hash
0x70	0x8	_MonoGHashTable *	_MonoGHashTable *	type_init_exception_hash

ניתן לראות שב-Offset 0x50 הוא מכיל מצביע למנהל קוד של המנוע.

כשמביטים במבנה נתונים של ניהול הקוד שבתמונה הבאה:

Structure Editor - _MonoCodeManager (mono-2.0-bdwgc.dll) [CodeBrowser: mono:/mono-2.0-bdwgc.dll]

Edit Help

Structure Editor - _MonoCodeManager (mono-2.0-bdwgc.dll)

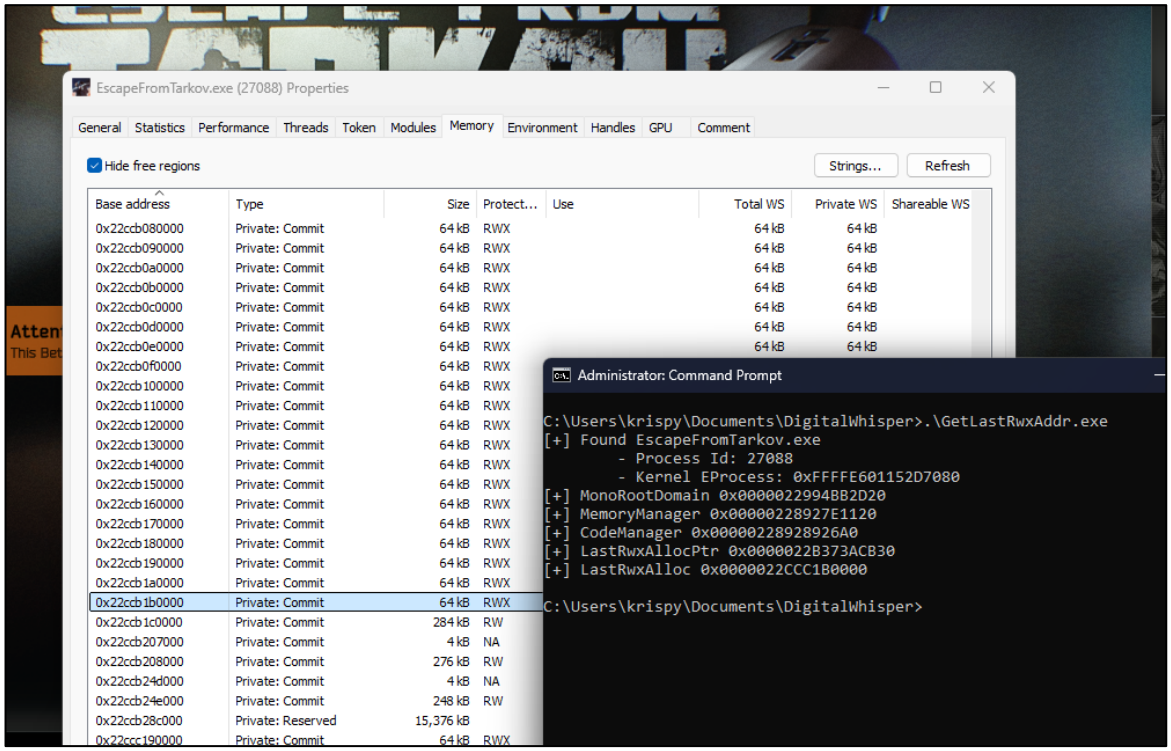
Offset	Length	Mnemonic	DataType	Name
0x0	0x8	_CodeChunk *	_CodeChunk *	current
0x8	0x8	_CodeChunk *	_CodeChunk *	full
0x10	0x8	_CodeChunk *	_CodeChunk *	last
0x18	0x1	int:1	int:1	dynamic
0x18	0x1	int:1	int:1	read_only

מגלים שהוא מחזיק רשימה מקושרת של CodeChunks, מושלם! כעת נוכל פשוט לקרוא את פרק הקוד האחרון שברשימה, ב-Offset 0x10 כפי שמודגש בתמונה, ואותו משתנה אמור להצביע למצביע של קוד RWX שניתן לכתוב אליו את ה-Shellcode שלנו מרחוק.

אז אם נצטרך לסכם את הקריאות שלנו, זה יראה כך:

1. קריאה של ה-mono_root_domain בכתובת mono-2.0-bdwgc.dll+0x751020
2. קריאה של ה-memory_manager בכתובת mono_root_domain+0x248
3. קריאה של ה-code_manager בכתובת memory_manager+0x50
4. קריאה של המצביע לכתובת האחרונה, בכתובת code_manager+0x10
5. קריאה נוספת לכתובת משום שהיא מצביעה למצביע, לכן - last_rwx_alloc+0x00

לאחר קריאה של אותו הזיכרון בתוכנה שכתבתי ב-C, ובעזרת ניצול של הדרייבר בעל החולשה, ניתן למצוא את הכתובת, ולוודא שהיא נכונה עם בדיקה קצרה מול Process Hacker:



והנה הקוד, בדיוק כפי שתואר לפני:

```
std::uint8_t* MonoBaseAddress = reinterpret_cast<std::uint8_t*>(Driver->GetModuleBase("mono-2.0-bdwgc.dll"));
const std::uintptr_t MonoDomain = Driver->ReadProcessMemory<std::uintptr_t>(reinterpret_cast<std::uintptr_t>(MonoBaseAddress) + 0x751020);
const std::uintptr_t MemoryManager = Driver->ReadProcessMemory<std::uintptr_t>(MonoDomain + 0x248);
const std::uintptr_t CodeManager = Driver->ReadProcessMemory<std::uintptr_t>(MemoryManager + 0x50);
const std::uintptr_t LastRwxAllocPtr = Driver->ReadProcessMemory<std::uintptr_t>(CodeManager + 0x10);
const std::uintptr_t LastRwxAlloc = Driver->ReadProcessMemory<std::uintptr_t>(LastRwxAllocPtr);
std::printf("[+] MonoRootDomain 0x%p\n", MonoDomain);
std::printf("[+] MemoryManager 0x%p\n", MemoryManager);
std::printf("[+] CodeManager 0x%p\n", CodeManager);
std::printf("[+] LastRwxAllocPtr 0x%p\n", LastRwxAllocPtr);
std::printf("[+] LastRwxAlloc 0x%p\n", LastRwxAlloc);
if (!LastRwxAlloc)
    return EXIT_FAILURE;
```

מגניב, עכשיו אנחנו יודעים לאן אפשר לכתוב את ה-Shellcode שלנו, שיבצע אילקוץ זיכרון נוסף עבורינו דרך אותו Code manager.

מפני שהשלב השני דורש מאיתנו שקוד ירוץ, עדיין לא סיימנו עם השלב הראשון. יש לנו שרשרת קריאה שמביאה אותנו למיקום RWX בו נשים את ה-Shellcode, וזה נהדר, אבל אנחנו עדיין צריכים להבין איך אנחנו בכלל נריץ קוד.

הרצת קוד שרירותי

יש המון דרכים להריץ קוד ראשוני מרחוק, אנחנו יכולים לבצע Patch לפונקציה שנקראת באופן מחזורי, או למצביע של פונקציה שנקראת כל כמה זמן, שנמצא ב-data section. עם זאת, רציתי ללכת עם דרך קצת יותר מגניבה, ובטוחה, שלא נופלת לאותם race conditions שנוכל להגיע אליהן במידה ונערוך קוד של המשחק. ובגלל זה בחרתי ללכת עם דרך מקורית שפורסמה על ידי החוקר אלון לבייב תחת השם- Pool Party. יש מאמר מקיף ומלא שנכתב על ידי אלון, ולכן אחסוך מכם את הפרטים הטכניים, משום שהשימוש בשיטה אינה ספציפית עבור המאמר הזה, ושיטות אחרות יכולות להיות משומשות באותה מידה. עם זאת, מה שנדרש מאיתנו בשביל להשתמש בשיטה הוא ליצור ולכתוב מבנה נתונים בשם TP_DIRECT לזיכרון המשחק, אשר מחזיק שדה בשם Callback, המצביע לכתובת בה קיים הקוד אותו נרצה להריץ. לאחר מכן נצטרך לשכפל Handle בעל הרשאות IO_COMPLETION_ALL_ACCESS (יש עוד כמה דרכים של הטכניקה שעובדות, אך בחרתי להשתמש ב-IO Completion, שנחשב כגרסה מספר 4 ב-GitHub של הטכניקה).

לאחר מכן נקרא לפונקציית Win API בשם ZwSetIoCompletion, ונעביר לה מצביע למבנה נתונים TP_DIRECT שיצרנו מוקדם יותר. לאחר הקריאה לפונקציה הזו, תהליך ה"קורבן", שבמקרה שלנו הוא משחק המחשב, יריץ את הקוד אליו מבנה הנתונים מצביע בעזרת Worker thread שקיים בתהליך.

לבסוף הקוד הדגמה שלנו יראה כך:

```
DuplicateHandle(
    TargetProcess,
    reinterpret_cast<HANDLE>(IoCompletionHandle),
    GetCurrentProcess(),
    &HijackedIoHandle,
    IO_COMPLETION_ALL_ACCESS,
    false,
    NULL
);

// Write TP_DIRECT structure with a callback pointing to our code
const std::uintptr_t BreakPointAddress = LastRwxAlloc + sizeof(TP_DIRECT);
const std::uintptr_t TpDirectAddress = LastRwxAlloc;
const TP_DIRECT TpDirect = { .Callback = reinterpret_cast<void*>(BreakPointAddress) };
Driver->WriteProcessMemory<TP_DIRECT>(TpDirectAddress, TpDirect);

// Write our actual code, here it's just an "int 3" instruction as a test
Driver->WriteProcessMemory<BYTE>(BreakPointAddress, 0xCC);

// Trigger the remote callback using ZwSetIoCompletion
const NTSTATUS SetIoCompletionResult = ZwSetIoCompletion(HijackedIoHandle, reinterpret_cast<PTP_DIRECT>(TpDirectAddress), 0, 0, 0);
if (SetIoCompletionResult != ERROR_SUCCESS)
    std::printf("[!] Call failed (Error status 0x%d)\n", TpDirect.Callback, SetIoCompletionResult);
```

ניתן לראות שאנו כותבים את מבנה הנתונים TP_DIRECT, ולאחר מכן את הקוד אותו אנחנו רוצים להריץ. בתור הדגמה אני מריץ אך ורק את ההוראה int 3, אשר מוסיפה מפסק לתוכנה.

לפני ההרצה, ווידאתי שהמשחק רץ תחת התוכנה WinDbg, בשביל שיהיה קל לאמת שהפסיקה באמת מתרחשת.

כפי שניתן לראות בתמונה הבאה, באמת עצרנו בהוראה int 3, וניתן לראות במחסנית ה-Thread שאנחנו רצים תחת הפונקציה TppWorkerThread:

```
000002b5`327f0034 0000          add     byte ptr [rax]
000002b5`327f0036 0000          add     byte ptr [rax]
000002b5`327f0038 48007f32      add     byte ptr [rdi+
000002b5`327f003c b502          mov     ch, 2
000002b5`327f003e 0000          add     byte ptr [rax]
000002b5`327f0040 0000          add     byte ptr [rax]
000002b5`327f0042 0000          add     byte ptr [rax]
000002b5`327f0044 0000          add     byte ptr [rax]
000002b5`327f0046 0000          add     byte ptr [rax]
000002b5`327f0048 cc            int     3
000002b5`327f0049 54            push    rsp
000002b5`327f004a b302          mov     bl, 2
000002b5`327f004c 0000          add     byte ptr [rax]
000002b5`327f004e ffe0          jmp     rax
```

```
C:\Users\krispy\Documents\DigitalWhisper>.\ExecBreakpoint.exe
[+] Found EscapeFromTarkov.exe
        - Process Id: 29712
        - Kernel EProcess: 0xFFFFE6011810B080
[+] MonoRootDomain 0x000002B210F02D20
[+] MemoryManager 0x000002B10EAF7E10
[+] CodeManager 0x000002B10EB33DE0
[+] LastRwxAllocPtr 0x000002B2CB1BBE50
[+] LastRwxAlloc 0x000002B5327F0000
```

Stack

Frame Index	Call Site	Child-SP	Return Address
[0x0]	0x2b5327f0048	0xd85977f688	0x7ffa94751f...
[0x1]	ntdll!TppWorkerThread+0x5a0	0xd85977f690	0x7ffa9267e8d7
[0x2]	KERNEL32!BaseThreadInitThunk+0x17	0xd85977f9f0	0x7ffa947dfbcc
[0x3]	ntdll!RtlUserThreadStart+0x2c	0xd85977fa20	0x0

כתיבת ה-Shellcode שמאלקץ זיכרון

עכשיו כשיש לנו כתיבה וקריאה של זיכרון, והרצת קוד שרירותית, אנחנו יכולים לאלקץ זיכרון חדש במשחק שמדמה נראות של קוד לגיטימי שמשומש על ידי המשחק. על מנת להסוות את הזיכרון שלנו כזיכרון של המשחק, נצטרך להבין כיצד המשחק מאלקץ ושומר את הזיכרון עבור עצמו. ב-Windows, קיימת פונקציה ב-Win API אשר משמשת עבור בקשת זיכרון ממערכת ההפעלה, שמה הוא VirtualAlloc (קיימת גם גרסה מורחבת בשם VirtualAllocEx, אך לא נראה שהיא בשימוש):

VirtualAlloc function (memoryapi.h)

Article • 02/05/2024 Feedback

In this article

- Syntax
- Parameters
- Return value
- Remarks
- Show 2 more

Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.

To allocate memory in the address space of another process, use the VirtualAllocEx function.

Syntax

C++ Copy

```
LPVOID VirtualAlloc(
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

כאשר נחפש היכן הפונקציה נקראת בעזרת Ghidra, נמצא את הפונקציה הגנרית mono_valloc אשר משמשת כמעטפת ל-VirtualAlloc, שהמנוע קורא לה בכל אלקוץ זיכרון:

```
Decompile: mono_valloc - (mono-2.0-bdwgc.dll)
1
2 void * __cdecl mono_valloc(void *param_1,ulong64 param_2,int param_3,MonoMemAccountType param_4)
3
4 {
5   int iVar1;
6   void *pvVar2;
7
8                    /* 0x72370  1215  mono_valloc */
9   if ((alloc_limit != 0) && (alloc_limit <= total_allocation_count + param_2)) {
10    return (void *)0x0;
11  }
12  iVar1 = mono_mmap_win_prot_from_flags(param_3);
13  pvVar2 = (void *)VirtualAlloc(param_1,param_2,0x3000,iVar1);
14  LOCK();
15  allocation_count[(int)param_4] = allocation_count[(int)param_4] + (longlong)(int)param_2;
16  UNLOCK();
17  LOCK();
18  total_allocation_count = total_allocation_count + (longlong)(int)param_2;
19  UNLOCK();
20  return pvVar2;
21 }
```

כעת ניתן לחפש איזה פונקציה קוראת לפונקציה הזו, ולאחר מכן איזה פונקציה קוראת לה, וכן הלאה. כך נוכל בסוף למצוא את הפונקציה שיוצרת buffer חדש הנועד לשמירת קוד Native עבור המשחק.

ניתן למצוא שהפונקציה codechunk_valloc קוראת ל-mono_valloc. אותה הפונקציה נקראת על ידי new_codechunk, אשר נקראת על ידי הפונקציה העיקרית: mono_code_manager_reserve_align:

```
Decompile: mono_code_manager_reserve_align - (mono-2.0-bdwgc.dll)
1
2 void * __cdecl
3 mono_code_manager_reserve_align(_MonoCodeManager *CodeManager,int param_2,int param_3)
4
5 {
6   _CodeChunk *p_Var1;
7   _CodeChunk *p_Var2;
8   void *pvVar3;
9   ulonglong uVar4;
10  _CodeChunk *p_Var5;
11  uint uVar6;
12
13  uVar4 = (ulonglong)(param_3 - 1);
14  if ((CodeManager->field_0x18 & 2) != 0) {
15    monoeg_assertion_message("* Assertion at %s:%d, condition `%s\' not met\n");
16  }
17  if (0x10 < param_3) {
18    monoeg_assertion_message("* Assertion at %s:%d, condition `%s\' not met\n");
19  }
20  if ((CodeManager->field_0x18 & 1) != 0) {
21    dynamic_code_alloc_count = dynamic_code_alloc_count + 1;
22    dynamic_code_bytes_count = dynamic_code_bytes_count + (longlong)param_2;
23  }
24  p_Var1 = CodeManager->current;
25  if (p_Var1 == (_CodeChunk *)0x0) {
26    p_Var1 = new_codechunk(CodeManager,param_2);
27    CodeManager->current = p_Var1;
28    if (p_Var1 == (_CodeChunk *)0x0) {
29      return (void *)0x0;
30    }
31    CodeManager->last = p_Var1;
32  }
```

בעזרת קצת הינדוס לאחור של הקוד, אם נמשיך לרדת באותו שרשרת פונקציות שציינתי לפני, נגלה כי הפרמטר הראשון של הפונקציה משמש עבור כתיבה וקריאה מה-CodeManager שמצאנו לפני. הפרמטר השני בסוף עובר לפרמטר השני של mono_valloc, שקובע את גודל אלוקציית הזיכרון כפי שהופיע בתמונה מ-MSDN. נראה שהפרמטר השלישי משומש עבור מטרות alignment, ונוכל להתעלם ממנו אם נקרא לעוד פונקציה שמוודא שהקוד שמאולקץ הינו aligned (אני מבטיח שזו הפונקציה האחרונה):

```
Decompile: mono_code_manager_reserve - (mono-2.0-bdwgc.dll)
1
2 void * __cdecl mono_code_manager_reserve(_MonoCodeManager *CodeManager,int ReserveSize)
3
4 {
5   void *AllocAddress;
6
7   AllocAddress = mono_code_manager_reserve_align(CodeManager,ReserveSize,0x10);
8   return AllocAddress;
9 }
```

מצאנו את הפונקציה, כעת ניגש לחלון ה-Disassembly בשביל למצוא את המיקום שלה בזיכרון התוכנה:

```
                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    void * __cdecl mono_code_manager_reserve(_MonoCodeManage...
                       assume GS_OFFSET = 0xff00000000
        void *         RAX:8          <RETURN>                          XREF[1]:     18006d486(W)
        _MonoCodeManag...  RCX:8      CodeManager
        int            EDX:4          ReserveSize
        undefined8     RAX:8          AllocAddress                      XREF[1]:     18006d486(W)
                    mono_code_manager_reserve
  18006d480 41 b8 10      MOV        R8D,0x10
            00 00 00
```

עכשיו אנחנו יכולים לכתוב shellcode, אשר יקרא לפונקציה הזו בשביל לאלקץ לנו זיכרון שידמה קוד של המשחק. מכיוון שהגודל הנפוץ ביותר באלוקציות זיכרון קוד הוא 0x10000 בתים, כפי שראינו בתהליך הבדיקה, נשתמש גם אנחנו באותו המספר. בגלל שלא נרצה מגבלת גודל על ה-DLL הזדוני שלנו, נכתוב קוד אסמבלי בסיסי שיקרא לפונקציה כמות מוגדרת של פעמים. נתחיל מקוד לופ בסיסי:

```
1   ; ebx will be used as our allocation counter
2   ; so it should be reset at the start
3   xor ebx, ebx
4
5   LoopStart:
6       ; r9d will be used as the value we will be comparing ebx against
7       xor r9d, r9d
8       ; let's allocate 10 chunks for now
9       mov r9d, 10
10
11      ; If this is the FIRST code chunk we're allocating, then we want to store it's
12      ; return address, because we obviously want the base of all of our chunks,
13      ; to map our malicious DLL at the start. So, if it's the first one, we store rax (return value) in r10
14      cmp ebx, 0
15      jnz SkipFirstAlloc
16      mov r10, rax
17  SkipFirstAlloc:
18      ; Increment the counter and compare, since a chunk allocation has succeeded.
19      inc ebx
20      cmp ebx, r9d
21      jl LoopStart
22      ret
```

אותו הקוד מבצע לולאה אשר תבצע כ-10 חזרות, כל חזרה האוגר ebx יגדל ב-1 ונשווה אותו אל מול r9d בכדי לוודא שהוא לא רץ יותר מ-10 פעמים. כמובן שבגלל שאנחנו קוראים לפונקציית אלקוץ מספר פעמים, ה-return value יישמר באוגר rax כל פעם מחדש, אך אנחנו רוצים לשמור רק את הכתובת שאנחנו מקבלים מהקריאה הראשונה לאותה פונקציה. נוכל לבצע זאת באמצעות קפיצות ו-labels. כפי שניתן לראות בקוד, במידה והלולאה מריצה את האיטרצייה הראשונה, אנחנו נשמור את האוגר rax לתוך r10 בכדי לשמר את המידע שלו.

כעת חסרים לנו שני דברים חשובים, קודם כל עליינו לקרוא לפונקציה שמצאנו עם הפרמטרים הנכונים. שנית, אנחנו צריכים לזכור שהערך שאנחנו שומרים ב-r10 לא מתוקשר חזרה אלינו, ולכן כדאי לשמור אותו במקום שמוגדר לפני הרצת ה-shellcode, בכדי שיהיה קל לקרוא אותו ולדעת היכן נוצרה האלוקצייה הראשונה שלנו.

שני השינויים הללו מתווספים כאן:

```
1   ; ebx will be used as our allocation counter
2   ; so it should be reset at the start
3   xor ebx, ebx
4
5   LoopStart:
6       ; Here we will store the address of
7       ; mono_code_manager_reserve(_MonoCodeManager *CodeManager,int ReserveSize)
8       movabs r8, 0x111111111111
9
10      ; Here we set up the arguments for the function call of mono_code_manager_reserve
11      ; RCX = First arg = Address of CodeManager
12      ; RDX = Second arg = Size of allocation
13      movabs rcx, 0x222222222222
14      mov rdx, 0x10000
15      call r8
16
17      ; r9d will be used as the value we will be comparing ebx against
18      xor r9d, r9d
19      ; Let's allocate 10 chunks for now
20      mov r9d, 10
21
22      ; If this is the FIRST code chunk we're allocating, then we want to store it's
23      ; return address, because we obviously want the base of all of our chunks,
24      ; to map our malicious DLL at the start. So, if it's the first one, we store rax (return value) in r10
25      cmp ebx, 0
26      jnz SkipFirstAlloc
27      mov r10, rax
28  SkipFirstAlloc:
29      ; Increment the counter and compare, since a chunk allocation has succeeded.
30      inc ebx
31      cmp ebx, r9d
32      jl LoopStart
33      ; We use rsi to store the address of where we write our result to, this is decided by the
34      ; PrepareShellcode(...) function. Currently, it just writes the address at the beginning of the RWX allocation
35      ; which we are executing inside of. So that we can read our call result.
36      movabs rsi, 0x333333333333
37      mov [rsi], r10
38      ret
```

בין שורות 8-15 מתבצעת הקריאה לפונקציה, ובין השורות 36-37 אנו מחליטים על כתובת לשמור אליה את המידע. כמובן שהכתובות שמצוינות בקוד אינן לגיטימיות, ולכן נצטרך שהתוכנה שלנו תערוך אותן באופן מקומי לפני הכתיבה וההרצה של ה-shellcode.

עכשיו חסר לנו רק דבר אחד אחרון בקוד, שהוא בטיחות. הרי אנחנו לא רוצים להשתמש באוגרים אשר נדרסים על ידי הקריאה לפונקציה, לכן אני שומר את כל האוגרים הרלוונטיים במחסנית ולאחר מכן מחזיר אותם לאחר הקריאה. החלטתי לבצע את שמירת ה-context הזו על כל האוגרים למעט rax, כי זה נוח:

```
push rbx
push rcx
push rdx
push rsi
push rdi
push rbp
push r8
push r9
push r10
push r11
push r12
push r13
push r14
push r15
movabs rcx, 0x222222222222
mov rdx, 0x10000
call r8
pop r15
pop r14
pop r13
pop r12
pop r11
pop r10
pop r9
pop r8
pop rbp
pop rdi
pop rsi
pop rdx
pop rcx
pop rbx
```

בנוסף, נתקלתי במספר בעיות בגרסאות מסוימות של ווינדוס (למשל 24H2), בהן כאשר אוגרים כמו rbx חורגים מערך מסוים, הקוד אשר מריץ את ה-Worker thread יכשל, או ישאיר את התהליכון בלולאה בלתי נפסקת. זאת משום שהקוד מצפה לערכים שהיו שמורים לפני שערכנו את האוגרים, ולכן חשוב לבצע את אותם ה-push/pop-ים שמבצעים שמירה/החזרת context בתחילת וסוף האסמבלי:

```
push rbx
push rcx
push rdx
push rsi
push rdi
push rbp
push r8
push r9
push r10
push r11
push r12
push r13
push r14
push r15
;
xor ebx, ebx
LoopStart:
pop r15
pop r14
pop r13
pop r12
pop r11
pop r10
pop r9
pop r8
pop rbp
pop rdi
pop rsi
pop rdx
pop rcx
pop rbx
ret
```

זה בהחלט מכער את הקוד, אבל במזל, זה לא משהו שאמור לעניין את המעבד שלנו :)

כעת ניתן להעביר את הקוד שלנו דרך Assembler, אני בחרתי להשתמש ב-Online x86 assembler מהאתר defuse.ca בשביל נוחות, אבל הבחירה לא משנה כלל.

האתר יחזיר לנו Array אותו נוכל להוסיף לתוכנה שלנו:

```
Array Literal:
{ 0x53, 0x51, 0x52, 0x56, 0x57, 0x55, 0x41, 0x50, 0x41, 0x51, 0x41, 0x52, 0x41, 0x53, 0x41, 0x54,
  0x41, 0x55, 0x41, 0x56, 0x41, 0x57, 0x31, 0xDB, 0x49, 0xB8, 0x30, 0xBF, 0x66, 0xCC, 0xF9, 0x7F,
  0x00, 0x00, 0x53, 0x51, 0x52, 0x56, 0x57, 0x55, 0x41, 0x50, 0x41, 0x51, 0x41, 0x52, 0x41, 0x53,
  0x41, 0x54, 0x41, 0x55, 0x41, 0x56, 0x41, 0x57, 0x48, 0xB9, 0x20, 0x2D, 0xD1, 0x24, 0xE1, 0x01,
  0x00, 0x00, 0x48, 0xC7, 0xC2, 0x00, 0x00, 0x01, 0x00, 0x41, 0xFF, 0xD0, 0x41, 0x5F, 0x41, 0x5E,
  0x41, 0x5D, 0x41, 0x5C, 0x41, 0x5B, 0x41, 0x5A, 0x41, 0x59, 0x41, 0x58, 0x5D, 0x5F, 0x5E, 0x5A,
  0x59, 0x5B, 0x45, 0x31, 0xC9, 0x41, 0xB9, 0x1E, 0x00, 0x00, 0x00, 0x83, 0xFB, 0x00, 0x75, 0x03,
  0x49, 0x89, 0xC2, 0xFF, 0xC3, 0x44, 0x39, 0xCB, 0x7C, 0x9E, 0x48, 0xBE, 0x20, 0x2D, 0xD1, 0x24,
  0xE1, 0x01, 0x00, 0x00, 0x4C, 0x89, 0x16, 0x41, 0x5F, 0x41, 0x5E, 0x41, 0x5D, 0x41, 0x5C, 0x41,
  0x5B, 0x41, 0x5A, 0x41, 0x59, 0x41, 0x58, 0x5D, 0x5F, 0x5E, 0x5A, 0x59, 0x5B, 0xC3 }
```

עכשיו נצטרך רק לכתוב לכמה מקומות ב-shellcode לפני שנכתוב אותו לזיכרון המשחק ונריץ אותו.

להזכירכם שלושת הדברים הבאים צריכים להיכתב לזיכרון לפני שהקוד יעבוד: הכתובת של הפונקציה אליה אנו קוראים בשביל לאלקץ קוד, הכתובת של ה-Code manager אותה מצאנו לפני, בשביל להעביר אותה בתור הפרמטר הראשון ש-mono_code_manager_reserve מקבל.

ולבסוף, אנחנו צריכים כתובת בה נוכל לשמור את הערך של האוגר r10 שישמור את הכתובת של ה- Code chunk הראשון שלנו. לאחר מכן, נקרא את הכתובת הזו בשביל לדעת היכן יתחיל המיפוי של ה-DLL הזדוני שלנו:

```
void PrepareShellcode(const std::uint8_t* Rwx, const std::uint8_t* MonoCodeManager, const std::uint8_t* StartAddressReserve)
{
    memcpy(&Shellcode[26], (void*)&StartAddressReserve, sizeof(uintptr_t));
    memcpy(&Shellcode[58], (void*)&MonoCodeManager, sizeof(uintptr_t));
    memcpy(&Shellcode[124], (void*)&Rwx, sizeof(uintptr_t));
}
```

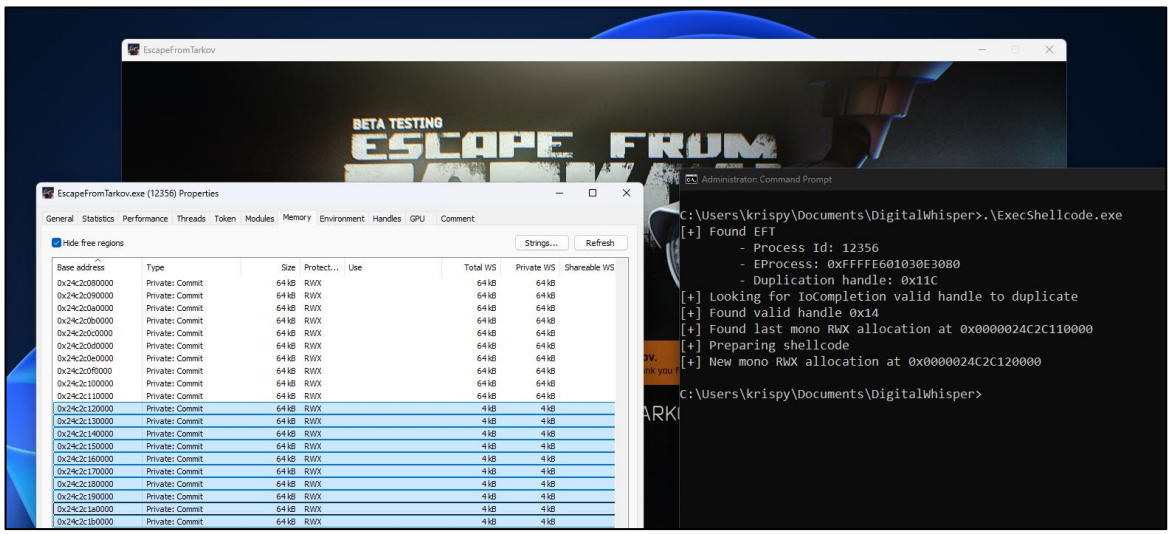
כעת נוכל לקרוא לפונקציה בצורה הבאה:

```
// Store original bytes
*OriginalBytes = Driver->ReadProcessMemory<OriginalBytesData>(LastRwxAlloc, sizeof(OriginalBytesData));

PrepareShellcode(
    reinterpret_cast<std::uint8_t*>(LastRwxAlloc),
    reinterpret_cast<std::uint8_t*>(CodeManager),
    ResolveCodeReserve(MonoBaseAddress)
);
```

עכשיו ניתן להריץ קוד מרחוק שנית, רק הפעם עם ה-shellcode החדש שלנו:

```
const std::uintptr_t TpDirectAddress = DataAddress;
const std::uintptr_t ShellcodeAddress = DataAddress + sizeof(TP_DIRECT);

// Write TP_DIRECT struct which holds the shellcode address to execute
const TP_DIRECT TpDirect = { .Callback = reinterpret_cast<void*>(ShellcodeAddress) };
Driver->WriteProcessMemory<TP_DIRECT>(TpDirectAddress, TpDirect);

// Write shellcode after the TP_DIRECT struct
Driver->WriteProcessMemoryPtr<std::uint8_t*>(ShellcodeAddress, Shellcode, sizeof(Shellcode));

// Execute remote shellcode
const NTSTATUS SetIoCompletionResult = ZwSetIoCompletion(HijackedIoHandle, reinterpret_cast<PTP_DIRECT>(TpDirectAddress), 0, 0, 0);
if (SetIoCompletionResult != ERROR_SUCCESS)
    return false;

const std::uintptr_t NewRwxAllocation = Driver->ReadProcessMemory<std::uintptr_t>(DataAddress);
if (!NewRwxAllocation)
    return false;
```

לאחר הרצה ניתן למצוא את ה-Code chunks החדשים אותם יצרנו:



כעת הדבר היחידי שנשאר לעשות הוא להזריק קובץ DLL דמה, בפועל כל מה שזה אומר זה לטעון קובץ DLL לזיכרון המקומי של התוכנה שלנו, לתקן דברים ב-PE Headers כמו relocations ולמצוא את ה- Entry point בעזרת ה-PE, בשביל שנדע מה הפונקציה הראשונה אליה נרצה לקרוא:

```
// Map cheat dll locally
const std::uintptr_t DllBytes = reinterpret_cast<std::uintptr_t>(CheatBytes.data());
const IMAGE_DOS_HEADER* DosHeader = reinterpret_cast<IMAGE_DOS_HEADER*>(DllBytes);
const IMAGE_NT_HEADERS* NtHeader = reinterpret_cast<IMAGE_NT_HEADERS*>(DllBytes + DosHeader->e_lfanew);

// Validate headers
if (DosHeader->e_magic != IMAGE_DOS_SIGNATURE || NtHeader->Signature != IMAGE_NT_SIGNATURE)
    return false;

const size_t PeHeaderSize = 0x1000;

// We do not map the PE header
DllEntryPoint = NewRwxAllocation + NtHeader->OptionalHeader.AddressOfEntryPoint;
if (!DllEntryPoint)
    return false;

// Allocate zero initialized local memory for DLL setup
void* LocalImageBase = malloc(NtHeader->OptionalHeader.SizeOfImage);
memset(LocalImageBase, 0, NtHeader->OptionalHeader.SizeOfImage);
if (!LocalImageBase)
    return false;

std::printf("[+] Mapping image locally\n");
std::printf("[+] Local image base at 0x%p\n", LocalImageBase);

memcpy(LocalImageBase, DosHeader, PeHeaderSize);

const IMAGE_SECTION_HEADER* CurrentImageSection = IMAGE_FIRST_SECTION(NtHeader);
for (int i = 0; i < NtHeader->FileHeader.NumberOfSections; i++)
{
    std::printf("[+] Mapping %s section at 0x%p of size 0x%x\n",
        CurrentImageSection[i].Name,
        CurrentImageSection[i].VirtualAddress,
        CurrentImageSection[i].SizeOfRawData
    );
    void* LocalSectionAddress = reinterpret_cast<void*>(
        reinterpret_cast<std::uintptr_t>(LocalImageBase) + CurrentImageSection[i].VirtualAddress
    );
    memcpy(LocalSectionAddress, reinterpret_cast<void*>(
        reinterpret_cast<std::uintptr_t>(DosHeader) + CurrentImageSection[i].PointerToRawData),
        CurrentImageSection[i].SizeOfRawData
    );
}
```

כפי שניתן לראות בתמונה, אנחנו קודם כל נטען את קובץ ה-DLL לוקאלית. לאחר מכן ניצור מיפוי עבור כל ה-Sections שנמצאים בקובץ, בשביל למפות את המיקומים של כל Section בכתובת הווירטואלית התואמת שלו:

```
std::printf("[+] Resolving relocations locally\n");
const std::vector<RelocationData> Relocations = GetRelocations(reinterpret_cast<std::uintptr_t>(LocalImageBase));
const std::uintptr_t ImageDifference = NewRwxAllocation - NtHeader->OptionalHeader.ImageBase;
for (const auto& CurrentRelocation : Relocations)
{
    for (std::uint32_t i = 0; i < CurrentRelocation.Count; i++)
    {
        const std::uint16_t RelocationType = CurrentRelocation.Item[i] >> 12;
        const std::uint16_t RelocationOffset = CurrentRelocation.Item[i] & 0xFFF;

        if (RelocationType == IMAGE_REL_BASED_DIR64)
            *reinterpret_cast<std::uintptr_t*>(CurrentRelocation.Address + RelocationOffset) += ImageDifference;
    }
}
```

לאחר מכן נעבור על כל הרילוקאציות שיש בקובץ הבינארי על מנת לתקן אותם לוקאלית. ולבסוף, נוכל למחוק את ה-PE Header משום שהוא לא נחוץ יותר, ורק ישאיר עקבות מוזרות בזיכרון המשחק. נכתוב את כל תוכן הבינארי המתוקן לתחילת ה-Code chunks שאילקצנו לפני:

```
// Clear out PE header locally, we no longer need it
memcpy(LocalImageBase, OriginalBytes, PeHeaderSize);
std::uint8_t* CodeStartAddress = reinterpret_cast<std::uint8_t*>(reinterpret_cast<std::uintptr_t>(LocalImageBase));

// Copy locally mapped image (without PE header) to EFT
std::printf("[+] Copying local image to remote target\n");
Driver->WriteProcessMemoryPtr<std::uint8_t*>(NewRwxAllocation, CodeStartAddress, NtHeader->OptionalHeader.SizeOfImage);
```

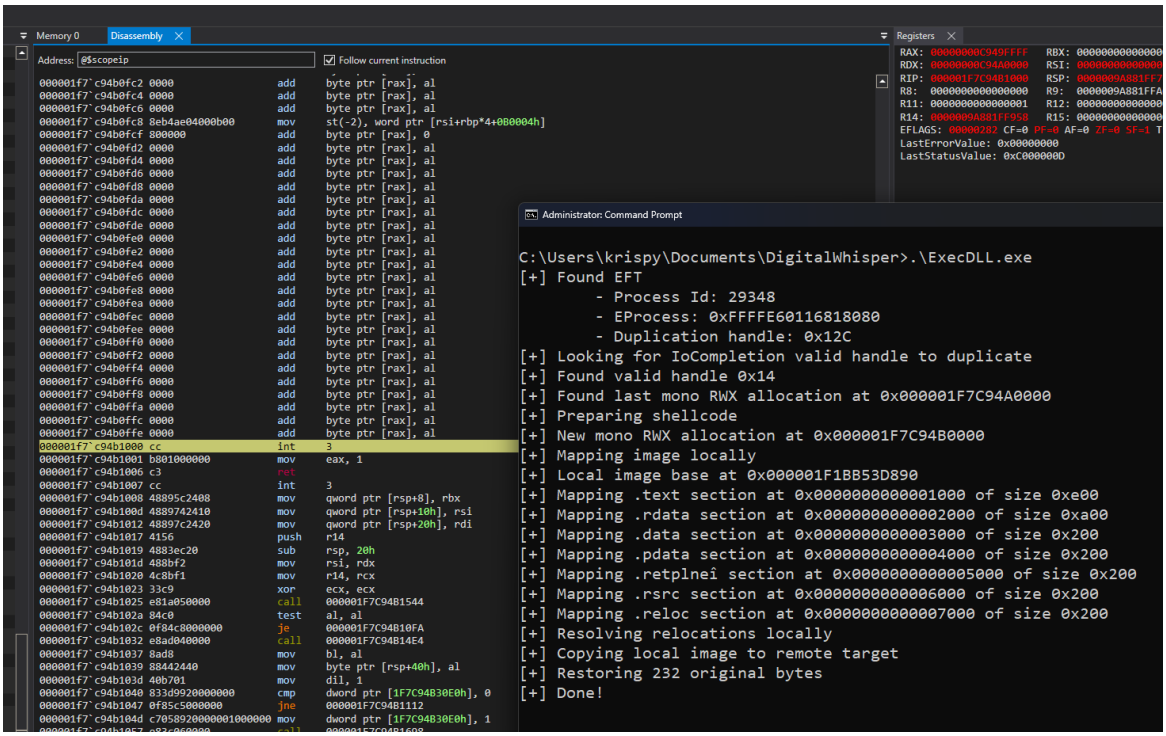
עכשיו נשאר לנו לקרוא ל-Entry point שמיפינו:

```
const std::uintptr_t TpDirectAddress = LastRwxAlloc;
const TP_DIRECT TpDirect = { .Callback = reinterpret_cast<void*>(DllEntryPoint) };
Driver->WriteProcessMemory<TP_DIRECT>(TpDirectAddress, TpDirect);

Sleep(5);

const NTSTATUS SetIoCompletionResult = ZwSetIoCompletion(HijackedIoHandle, reinterpret_cast<PTP_DIRECT>(TpDirectAddress), 0, 0, 0);
if (SetIoCompletionResult != ERROR_SUCCESS)
    std::printf("[!] Failed calling DLL entry point at 0x%p (Error status 0x%d)\n", TpDirect.Callback, SetIoCompletionResult);

Sleep(2000);

// Restore original bytes
std::printf("[+] Restoring %d original bytes\n", sizeof(OriginalBytesData));
Driver->WriteProcessMemoryPtr<OriginalBytesData*>(LastRwxAlloc, OriginalBytes, sizeof(OriginalBytesData));
delete OriginalBytes;

std::printf("[+] Done!\n\n");
```

עכשיו בואו ניצור DLL דמה אשר אותו נטען. תהיה בו פונקציה בודדת שתהיה ה-Entry point, וכל מה שהיא תעשה זה להריץ את ההוראה int 3 כמו שניסינו בהתחלה, וכמובן, שוב פעם נוודא ש-WinDbg בשימוש. נוכל לקמפל את ה-DLL בעזרת הקומפיילר Clang, משום שהוא מאפשר שימוש ב- Inline assembly גם בקימפול עבור ארכיטקטורת x86_64, לעומת הקומפיילר MSVC של Windows:

```
#include <Windows.h>

BOOL WINAPI DllMain(HINSTANCE Instance, DWORD Reason, LPVOID Reserved)
{
    __asm { int 3 }
    return TRUE;
}
```

כך יראה הקוד שלנו, עכשיו, בואו נריץ אותו:

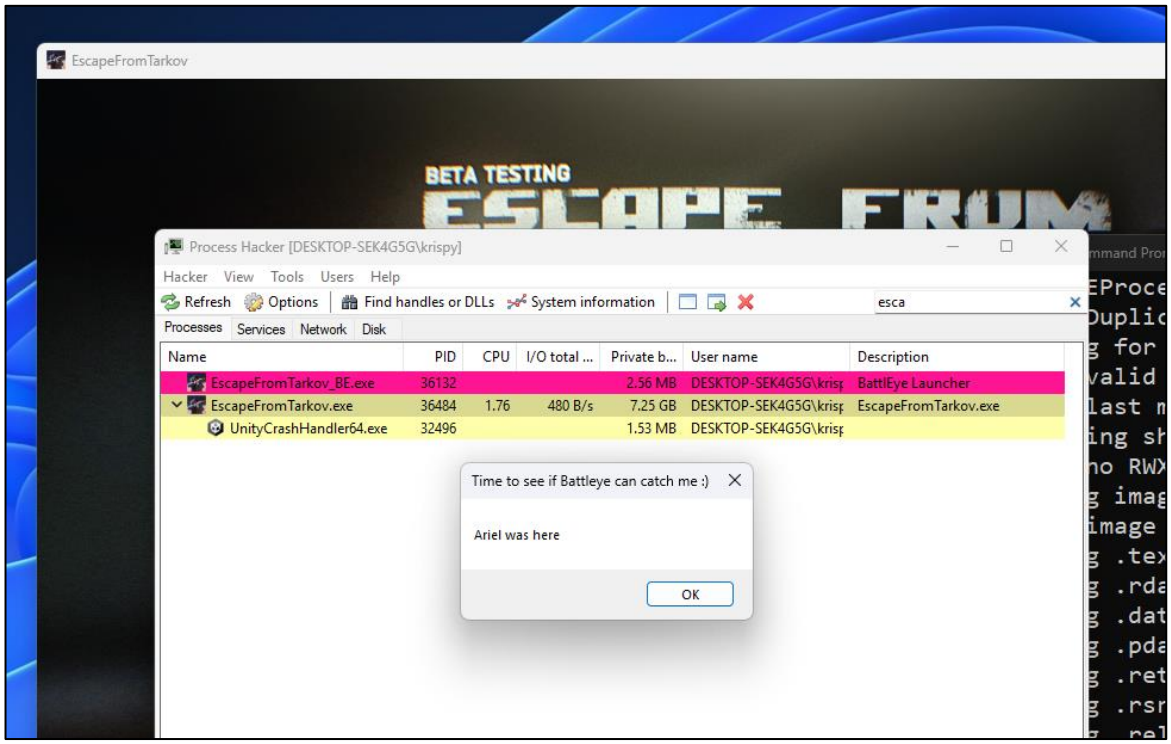


כפי שצופה, אנחנו מריצים int 3 ו-ret ישירות לאחר מכן.

אז מגניב! הכל עובד. הדבר היחידי שחסר הוא תיקון של ה-Import address table של קובץ ה-PE. החלטתי שלא להוסיף קוד עבור זה, משום שאם אתם זוכרים, מחקנו אה-PE header מהזיכרון בזמן תהליך מיפוי ה-DLL, על מנת לא להשאיר עקבות בזיכרון התוכנה. הדבר לא מהווה בעיה משום שניתן לכתוב קוד שבאופן דינאמי מוצא את אותן פונקציות ב-Export address table של DLL-ים אחרים שטעונים בתהליך.

הרצה מול Battleye

כעת ניתן לבדוק ולגלות שהתוכנה עובדת גם כאשר Battleye רץ לצד המשחק:



כפי שניתן לראות, התהליך ש-Battleye יוצר תחת השם EscapeFromTarkov_BE.exe רץ ברקע, וכל הבקשות לקריאת/כתיבת הזיכרון ברמת ה-Usermode לא עובדות. עם זאת, ה-MessageBox שה-DLL שלי מזריק ומריץ עובד בלי שום בעיות, והקוד מוחבא איפשהו בין שאר הקוד של המשחק!

מה ניתן לעשות בכדי להגן מהשיטה הזו?

ראשית, בעזרת Hook-ים במקומות הנכונים, תוכנת ההגנה מסוגלת לבצע מעקב אחר קוד אשר באמת נוצר על ידי מנוע ה-JIT, ובכך לסנן קוד שאינו נבע מן המנוע. עם זאת, הדבר כנראה אינו תרחיש סביר מרוב השימוש הנפוץ בכלל פונקציות המנוע. הוספת מערכת מעקב שכזו תוכל לגרום לצניחה משמעותית בביצועי המשחק, במידה ואינה מפותחת באופן קפדני ביותר.

שנית, במידה וניתן להגיע למצב בו המנוע יהדר קוד זהה עבור כל מכונה שמריצה את המשחק, קיימת אפשרות לזהות קוד אשר אינו שייך למשחק - אך הדבר יגרום לאיבוד יתרון משמעותי בשימוש מנוע הרצה מסוג JIT אשר מאפשר ייעול קוד שתלוי בחומרת המחשב.

בנוסף, ניתן לעקוב לאחר הקוד ה-Native אשר מנוע ה-JIT יוצר, ולגלות שבסך הכל, הוא בעל דפוס שיחסית קל לחזות מראש. רובו המוחלט של הקוד מורכב ממספר הוראות בודדות, ומשתמש לרוב באותם האוגרים עבור פעולות שונות:

```
Memory 0   Disassembly
Address: 0x14862e40000                    Follow current instruction
00000148`62e4027b 488b4010          mov     rax, qword ptr [rax+10h]
00000148`62e4027f 4883c118          add     rcx, 18h
00000148`62e40283 48898518ffffff    mov     qword ptr [rbp-0E8h], rax
00000148`62e4028a 488901            mov     qword ptr [rcx], rax
00000148`62e4028d 90                nop
00000148`62e4028e 49bbc92c891c47010000 mov  r11, 1471C892CC9h
00000148`62e40298 41ffd3            call    r11
00000148`62e4029b 488b8518ffffff    mov     rax, qword ptr [rbp-0E8h]
00000148`62e402a2 488b9510ffffff    mov     rdx, qword ptr [rbp-0F0h]
00000148`62e402a9 4c8b8d08ffffff    mov     r9, qword ptr [rbp-0F8h]
00000148`62e402b0 4c8bc2            mov     r8, rdx
00000148`62e402b3 498b4710          mov     rax, qword ptr [r15+10h]
00000148`62e402b7 488bc8            mov     rcx, rax
00000148`62e402ba 4c898d58ffffff    mov     qword ptr [rbp-0A8h], r9
00000148`62e402c1 4c898550ffffff    mov     qword ptr [rbp-0B0h], r8
00000148`62e402c8 48899548ffffff    mov     qword ptr [rbp-0B8h], rdx
00000148`62e402cf 48898d40ffffff    mov     qword ptr [rbp-0C0h], rcx
00000148`62e402d6 4885c0            test    rax, rax
00000148`62e402d9 0f85d1000000      jne     0000014862E403B0
00000148`62e402df 4d85ff            test    r15, r15
00000148`62e402e2 0f8400040000      je      0000014862E406E8
00000148`62e402e8 48b91884a06148010000 mov  rcx, 14861A08418h
00000148`62e402f2 488d6d00          lea     rbp, [rbp]
00000148`62e402f6 49bb1030891c47010000 mov  r11, 1471C893010h
00000148`62e40300 41ffd3            call    r11
00000148`62e40303 488bc8            mov     rcx, rax
00000148`62e40306 4d85ff            test    r15, r15
00000148`62e40309 0f84d2030000      je      0000014862E406E1
00000148`62e4030f 4c897920          mov     qword ptr [rcx+20h], r15
00000148`62e40313 48898d08ffffff    mov     qword ptr [rbp-0F8h], rcx
00000148`62e4031a 4883c120          add     rcx, 20h
00000148`62e4031e 49bbc92c891c47010000 mov  r11, 1471C892CC9h
00000148`62e40328 41ffd3            call    r11
00000148`62e4032b 488b8d08ffffff    mov     rcx, qword ptr [rbp-0F8h]
00000148`62e40332 48b8e0cbef6148010000 mov  rax, 14861EFCBE0h
00000148`62e4033c 48894128          mov     qword ptr [rcx+28h], rax
00000148`62e40340 48b8906e016248010000 mov  rax, 14862016E90h
00000148`62e4034a 48894140          mov     qword ptr [rcx+40h], rax
00000148`62e4034e 48b8986e016248010000 mov  rax, 14862016E98h
00000148`62e40358 488b5028          mov     rdx, qword ptr [rax+28h]
00000148`62e4035c 48895118          mov     qword ptr [rcx+18h], rdx
00000148`62e40360 488b4020          mov     rax, qword ptr [rax+20h]
00000148`62e40364 48894110          mov     qword ptr [rcx+10h], rax
00000148`62e40368 c6417000          mov     byte ptr [rcx+70h], 0
00000148`62e4036c 488bc1            mov     rax, rcx
00000148`62e4036f 48898d18ffffff    mov     qword ptr [rbp-0E8h], rcx
00000148`62e40376 48898d60ffffff    mov     qword ptr [rbp-0A0h], rcx
00000148`62e4037d 498d4f10          lea     rcx, [r15+10h]
00000148`62e40381 48898510ffffff    mov     qword ptr [rbp-0F0h], rax
00000148`62e40388 488901            mov     qword ptr [rcx], rax
00000148`62e4038b 666690            nop
00000148`62e4038e 49bbc92c891c47010000 mov  r11, 1471C892CC9h
00000148`62e40398 41ffd3            call    r11
```

דוגמא טובה הינה השימוש באוגר r11 עבור פעולות ה-call שמבצע הקוד. גם לאחר חיפוש ממוקד, לא הצלחתי למצוא שום קוד שהמנוע יצר שמשתמש באוגר אחר עבור קריאה. עם זאת, הדבר יהיה קל מאוד לעקיפה עבור תוקף, על ידי שימוש בקומפיילר כמו Clang שמייצר קוד בשפת הביניים LLVM IR, שלאחר מכן יעבור Pass-ים רלוונטיים, אשר יגרמו לכך שכל שימוש בהוראת call תצטרך להשתמש באוגר r11 בתור כתובת (לדוגמא).

בכל מקרה, אלו רק כמה מהדרכים שחשבתי עליהן, וקשה לי להאמין שיכניסו אותן לשימוש, בעיקר בגלל הקושי הפיתוחי או הביצועי עבור המגנים, לעומת כמות הזמן הנמוכה שתיקח לתוקף מנוסה לחמוק מאותן הגנות מחדש.

עם זאת, חשוב לזכור כי הזרקת קוד אל היעד לא פותרת אף תוקף משום צורך בשיטות התחמקות נוספות. תוכנות ההגנה עדיין יבצעו הטמנה של ה-Hook-ים שלהם, בדיקות בסיסיות של זיכרון חשוד, ועוד.

סיכום

תוכנות הגנה, בין אם מדובר בכאלו עבור משחקים, או אנטי-וירוסים ותוכנות EDR למיניהן, לא תמיד מתחשבות, או יכולות להתחשב מספיק, בסביבות בהן רצות. בסופו של יום, הבנה עמוקה של המגבלות איתן התוכנות מתמודדות, עלולה להוביל לשיטות יצירתיות לעקוף ולנתח כל מערכת נתונה. לא דיברתי על כך לאורך המאמר, משום שניתן לכתוב על כך מאמר שלם משלם עצמו, אך לאחר מחקר ממושך של הגנות Battleye במשחק, לא ראיתי שום דיווח שנשלח לשרת, או קוד אשר מחפש התנהגויות חשודות שקשורות לשיטה.

אשמח להודות לשני חברים מקהילות אינטרנטיות שעזרו לי לאורך המחקר, tr1x ו-kert.

על המחבר

שמי אריאל טרי, בן 19 ומשרת ביחידה טכנולוגית בצה"ל. זהו המאמר הראשון שכתבתי, ואני יותר מאשמח לענות על שאלות, לדון ולשמוע כל פידבק עליו בטוויטר או בלינקדאין. מקווה שנהניתם!

מקורות מידע

- https://digitalwhisper.co.il/files/Zines/0xA3/DW163-1-BYOVD.pdf
- https://github.com/TheCruZ/kdmapper
- https://loldrivers.io
- https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-obregistercallbacks
- https://discussions.unity.com/t/mono-vs-il2cpp/703013
- https://github.com/mono/mono
- https://www.youtube.com/watch?v=d7KHAVaX_Rs
- https://www.unknowncheats.me/forum/escape-from-tarkov/481919-battleyent-play-offline-raids-battleye-live-version.html
- https://github.com/mostdefinitelynotnth/battleyent
- https://processhacker.sourceforge.io/downloads.php
- https://github.com/NationalSecurityAgency/ghidra
- https://docs.unity.com/ugs/manual/cloud-diagnostics-advanced/manual/ConnectToSymbolServers
- https://github.com/SafeBreach-Labs/PoolParty
- https://www.safebreach.com/blog/process-injection-using-windows-thread-pools/
- https://defuse.ca/online-x86-assembler.htm