digital whisper Technologic Papers

Valve Anti-Cheat Bypass

מאת אילן ארנגאוז

הקדמה

מאמר זה יהיה סיכום של תהליך מחקר ופיתוח שערך כ-3 שבועות על מערכת מניעת הרמאות **VAC** של חברת Valve. VAC היא מערכת אשר מופעלת במספר משחקים של Valve כגון Counter Strike 2 (המשחק בו התמקדתי, ויצרתי אליו רמאות), Team Fortress 2, ועוד.

במאמר אפרט מעט על תהליך פיתוח הרמאות וגישות שונות לבעיה זו, אך אתמקד בגישות ושיטות למניעת רמאות, ודרכי מחקר ועקיפה של אותן שיטות.

המידע אשר אפרסם יתמקד על המשחק Counter Strike 2 והמערכת VAC (בפרט הגרסה של מערכת ההפעלה Windows) אך ברובו תקף לכלל משחקים ומערכות מניעת רמאות שונות.

בנוסף המאמר יכולל כ"בונוס" מידע מעניין למדי על VAC אשר לא תוכלו למצוא ברשת נכון לכתיבת מאמר זה, אז מומלץ לקרוא עד הסוף.

במאמר זה אניח כי אתם יודעים מה הם Hooks ואיך הם פועלים, מהו ה-WinAPI, מה זה DLL Injection, ורעיונות בסיסיים של אלגוריתמי Pseudorandom.

אציין כי כל המידע אשר נמצא במאמר זה הוא למטרות למידה בלבד, ואני לא אחראי על שימוש לא ראוי בו.

בניית תוכנות רמאות למשחקים

ראשית נתחיל בלחלק את תוכנות הרמאות לשני סוגים:

1. **רמאות חיצונית (External)** - רמאות בה נעשה שימוש בתוכנה חיצונית אשר קוראת וכותבת את זיכרון המשחק.
2. **רמאות פנימית (Internal)** - רמאות בה מזריקים ספרייה לתוך תהליך המשחק אשר יכול לקרוא ולכתוב זיכרון, ואף לקרוא לפונקציות של המשחק ולשנות את הרצתן (Hooking).

כאמור רמאות פנימית היא "חזקה" יותר, מכיוון שמאפשרת לקרוא ישירות לפונקציות המשחק ולשנות את הפונקציונליות שלהן, לדוגמא נוכל לקרוא לפונקציית המשחק אשר מחשבת היכן נמצא מיקום במפה על מסך המשחק (World to Screen), כאשר ברמאות חיצונית נצטרך לעשות חישוב זה בעצמנו (נוכל לעשות הרבה יותר מזה, זו רק דוגמא פשוטה).

החיסרון ברמאות פנימית היא שהיא מוסיפה משטח רחב יותר לזיהוי, ניתן לזהות למשל את ה-Thread בו רצה הספרייה או את הספריה בזכרון התהליך, אפרט קצת על זה בהמשך.

הערה: קיימת שיטה מעניינת למדי לעשות דברים שעושה רמאות פנימית בעזרת רמאות חיצונית על ידי אמולציה של תהליך המשחק והחלפת פעולות זיכרון בקריאות WinAPI, אך זה עדיין איטי וגרוע בהרבה.

כעת נעסוק במימוש של רמאות חיצונית, לצורך העניין אשתמש ברמאות המפורסמת במשחקי יריות "ESP" או "WallHack" כדוגמא, רמאות זאת מאפשרת לרמאי במשחק לראות אנשים דרך קירות מפת המשחק ובכך להשיג מידע שימושי ביותר על מיקומי היריבים שלו על המפה:

[איור: ה-ESP הבסיסי שלי ב-CS2]

יצירת הרמאות מחולקת לשני חלקים:

1. השגת המידע הנחוץ לרמאות (בפרט ל-ESP, מיקומי השחקנים על המפה ועוד)
2. הצגת המידע לרמאי (בפרט ל-ESP, ציור דמויות האויבים על מסך המשחק)

קיים מספר רחב של דרכים להשיג כל אחד מהחלקים האלו, אפרט מעט על העיקריות.

השגת המידע הנחוץ לרמאות

נניח כי אנחנו רוצים להשיג את מיקומי האויבים על המפה, השרת כמובן שולח ללקוח (המשחק אשר רץ אצלנו על המחשב) את המיקומים האלו (לא תמיד, ראו הערה) כדי שיוכל לבצע פעולות נחוצות למשחק כגון להציג את השחקנים על המסך, להשמיע את הקולות אשר הדמויות במשחק עושות וכו'

נוכל לקרוא מידע זה בכל "שלב" שהוא עובר מהגעתו מהשרת, כגון מעברו ברשת, וכתיבתו לזיכרון המשחק. במאמר אתמקד בקריאת המידע מזיכרון המשחק, אך גם "הסנפתו" ברשת היא שיטה לגיטימית ומשומשת.

הערה: בדרך כלל גם בשיטת ההסנפה נצטרך לבצע קריאה כלשהי מזכרון המשחק מכיוון שהמידע אשר מועבר כמעט תמיד יהיה מוצפן, ונרצה להשיג את מפתח ההצפנה כדי לפענח אותו (משימה לקורא: לחשוב למה לא יוכל להסניף גם מפתח זה אלא יצטרך לקרוא אותו מהזכרון של תהליך המשחק, רמז: SSL).

בנוסף, במשחקים מסוימים (כגון Valorant) קיימת שיטה יחסית צפויה אשר לא חושפת את הלקוח למידע אשר לא אמור להיות אצלו, כגון מיקומים של שחקנים רחוקים מעבר לקירות, אלא רק מיקומים של שחקנים שהוא אמור לראות. שיטה זו כמובן לא מושלמת כי צריך לידע את הלקוח יחסית מוקדם על מיקומים בגלל שיקולים כגון Ping שיכול לגרום לבעיות של "הופעה" של שחקנים על המסך, והרצון לאפשר שמיעת צעדים של שחקנים קרובים גם מעבר לקירות.

לקריאת זיכרון זה מתהליך המשחק ישנן שתי שיטות עיקריות:

1. קריאתו בעזרת פונקציות של מערכת ההפעלה מ-Usermode (בפרט בחלונות- WinAPI)
2. קריאתו בעזרת Kernel Driver עם שימוש בפונקציות קיימות לקריאת זיכרון וירטואלי, ולפעמים אף בעזרת חישוב מיקום הזיכרון בזיכרון הפיזי וקריאתו ישירות משם.

במאמר אתמקד בשיטה 1 מכיוון ש-VAC רץ בעצמו ב-Usermode ולכן שיטה 2 תיהיה "חזקה" מידי בשבילו ולא מעניינת כל כך (למרות שגם לה יהיו חלקים מעניינים כגון איך להעלות Driver כזה מבלי לבטל DSE אך זה סיפור ליום אחר).

בשביל לקרוא זיכרון של תהליך אחר (בפרט תהליך המשחק) במערכת ההפעלה חלונות יש ליצור אובייקט לאותו תהליך בשם Handle, אובייקט זה יחזיק מידע על ההרשאות הרצויות לגישה של אותו תהליך (כגון: קריאת זיכרון, כתיבת זיכרון וכו') והוא יהווה כדרך התקשורת שלנו לתהליך המשחק.

ה-Handle יפתח בעזרת הפונקציה של מערכת ההפעלה OpenProcess, ונוכל בעזרתו לקרוא לפונקציות כגון ReadProcessMemory ועוד אשר יתנו לנו גישה לקרוא את זיכרון התהליך, לכתוב אליו ועוד.

כעת נוכל לגשת לזיכרון המשחק ולקרוא את מיקומי השחקנים על המפה מהמערך שלהן בזיכרון, לאחר מכן נוכל לבצע חישובים מתמטיים פשוטים למדי (למשל World to Screen) כדי לדעת היכן לצייר על המסך את הדמויות ונצטרך לעבור להצגת המידע (ציור על המסך).

הערה: מציאת המיקום בזכרון המשחק בו נמצא מערך השחקנים היא לפעמים משימה לא פשוטה הדורשת לרברס חלק מקוד המשחק. לחלופין עבור חלק מהערכים למשל ערך החיים של שחקן, נוכל למצוא את המיקום בעזרת סריקת הזיכרון לערכים רצויים (לדוגמא בעזרת הכלי המפורסם Cheat Engine).

הצגת המידע לרמאי

כעת נעסוק בהצגת המידע לרמאי ובפרט בציור על מסך המשחק, אפרט בקצרה על מספר שיטות לציור מעל מסך המשחק.

ראשית נדבר על השיטה הפופולרית ביותר שבה עושים Hook לפונקציה במשחק אשר אחראית על הציור למסך, ומשנים את תוכן התמונה שהיא אמורה לצייר על המסך, בשיטה זו משתמשים בדר"כ ברמאות פנימית בגלל הצורך לשנות פונקציונליות של פונקציה של תהליך המשחק.

הערה: שיטה זו חושפת אותנו למשטח זיהוי נוסף של שינוי פונקציונליות המשחק, למשל אם נעזרנו ב-Inline Hook, יהיה ניתן לזהות את השינוי בקוד הפונקציה (Intergrity Check), אפרט עוד קצת בנושא זה בהמשך.

שיטה אחרת לציור על המסך היא שימוש בחלון בעל רקע בלתי נראה אשר ימוקם מעל חלון המשחק (Overlay), שיטה זו כמובן חושפת אותן לזיהוי הקיום של חלונות כאלו, אך ניתן לעקוף זאת בקלות יחסית על ידי ציור על חלון כזה אשר קיים כבר ברוב המחשבים כגון: Discord Overlay, Nvidia Overlay וכו' טכניקה זו נקראת בלעז "גניבת חלון" (Overlay Hijacking) והיא קשה יחסית לזיהוי.

פעולת מערכות לזיהוי רמאות

כעת אתחיל לדבר על דרכי הפעולה של VAC בפרט והסיבות לדרכי הפעולה האלו, בהמשך אדבר על שיטות ספציפיות לזיהוי רמאות אשר VAC נוקט בהן, ולאחר מכן רעיונות שונים לדרכי עקיפה של שיטות אלו.

VAC מתחיל לפעול כאשר השחקן נכנס למשחק ומתחבר לשרת המשחק, השרת שולח חבילה שמכילה קטע קוד (בפרט קובץ PE, נקרא לו Module), פונקציה בספריה steamservice.dll אשר רצה בתוך steam.exe שנקרא לה load_vac_module, טוענת את אותו ה-Module לתוך steam.exe, בשיטת Manual Mapping (בשיטה זה מקצים זיכרון בצורה ידנית ומעתיקים את הקוד לקטע זיכרון זה).

לאחר מכן השרת שולח חבילות אשר מורות להריץ את אותו קטע קוד (עם קלט שגם הוא נשלח מהשרת, קלט זה מכיל דברים כגון PID של תהליך שהשרת רוצה לחקור, מפתחות הצפנה ועוד), אשר מבצע סריקה

כלשהי ושולח לשרת את תוצאת הסריקה, לאחר מכן כמובן השרת יכול לנתח את תוצאת הסריקה ולהחליט אם יש סיבה לחשוד, ולנקוט באמצעים הנחוצים.

הערה: אם steam.exe רץ בהרשאות מנהל אז steamservice.dll רץ בתוך steam.exe, אחרת יהיה תהליך נפרד בשם steamservice.exe.

בנוסף, אני מתאר פה את הפעילות של VAC גרסה 3 (האחרונה) בעבר היו גרסאות אחרות כגון VAC 2 שהיו פשוטות יותר וקלות יותר לעקיפה, אך כבר לא משתמשים בהן.

יש מספר סיבות לדרך פעולה דינמית זו, לעומת קיום קובץ סטטי של מערכת ההגנה מרמאות: ראשית, דרך זו מקשה על חקר המערכת מכיוון שאין קובץ ספציפי אשר אותו יש לחקור, ויש למצוא דרך "לצלם" את הזיכרון כדי לחקור את אותם ה-Modules (אפרט עוד בהמשך).

שנית, שיטה זו מעניקה ל-Valve שליטה על הרצת חלקים שונים בזמנים שונים על פי רצונם, בנוסף הם יכולים לבטל Modules, ולהחזיר אחרים לפעולה, וכמובן ליצור חדשים. השליטה הזו מקשה גם היא על

הבנה מלאה של VAC, מכיוון שבכל משחק נטענים Modules שונים בעלי פונקציונליות שונה, ולפעמים יש לחכות שעות לקבלת Module מסויים.

הערה: קיימים מעל 20 Modules שונים אשר נטענים בזמנים שונים, לפעמים גם בתלות ב-Gamemode שהשחקן משחק (למשל ...Competitive, Casual, Deathmatch).

שיטות למחקר חלקים הנטענים דינמית לזיכרון

כאמור ה-Modules מוקצים לזכרון לאחר בקשה מהשרת, נרצה לכתוב אותם על הדיסק על מנת לבצע ניתוח סטטי שלהם, שיטה זאת נקראת Dumping.

נוכל לכתוב אותם על הדיסק על ידי Hook (למשל Inline Hook) לפונקציה load_vac_module אשר אחראית לטעינת ה-Modules, וכתיבת הספרייה על הדיסק.

נרצה כמובן גם לשמור עותק של הקלט שקיבל ה-Module, שיעזור לנו לחקור אותו:

```
char __stdcall load_vac_module_hook(DWORD* module, char a2) {
    cout << "load_vac_module(module=" << module << ", " <<  a2 << ") ";
    cout << "Module[6]: " << module[6];

    if (!module || !module[6])
        return load_vac_module(module, a2);

    std::bitset<8> bits(a2);
    cout << " a2: " << bits << endl;

    DWORD* v3 = (DWORD*)(module[6] + *(DWORD*)(module[6] + 60)); // get dll size from PE headers
    int size = v3[20];

    auto moduleHash = hash_header((void*) module[6]);
    std::stringstream stream;
    stream << std::hex << std::uppercase << moduleHash;
    std::string hashHexString(stream.str());

    string path = "C:\\Users\\ilana\\Desktop\\Volvo Anti Cheat\\dumps\\";
    string dumpPath = path + "dump_" + hashHexString + ".dll";
    string paramsPath = path + "params_" + hashHexString + ".txt";

    // write dump to file
    //...

    // write params to file
    ofstream params;
    params.open(paramsPath);
    params << _parameters;
    params.close();
    return load_vac_module(module, a2);
}
```

[איור: דוגמא ל-Hook ל-load_vac_module]

ניתוח סטטי של Modules

בניתוח הסטטי של של ה-Modules ניתקל במספר מכשולים (Obfuscations) אשר יקשו עלינו בהבנת הפונקציונליות, העיקרי מבינהם הוא הצפנת מחרוזות וטעינה דינמית של פונקציות WinAPI.

ה-Module ישמור מערך גלובלי מוצפן של שמות הפונקציות הרצויות של WinAPI (הוא יקבל אותו Initialized מהשרת), השמות יהיו מוצפנים בשיטת ההצפנה Ice עם מפתח שיתקבל ב-Input מהשרת, לפני ביצוע כל סריקה, ה-Module יפענח את המערך וישתמש בפונקציית GetProcAddress כדי להשתמש בשמות הפונקציות לקבלת מצביעים לאותן פונקציות, לאחר מכן ימלא מערך אחר אשר נקרא לו func_array במצביעים אלו.

כאשר ירצה ה-Module לקרוא לפונקציית WinAPI, הוא ישתמש במצביע אליה ב-func_array.

שיטה זו מקשה תחילה על ניתוח ה-Module כי מצריכה לפענח את מערך השמות על מנת להבין את קוד ה-Module (כי הקריאות יהיו מתוך func_array שתחילה אנחנו לא יודעים את תוכנו):

```
v115 = ((int (__stdcall *)(__int16 *, unsigned int, int, _DWORD, int, int, _DWORD))func_array[90])(
          v80,
          0x80000000,
          7,
          0,
          3,
          128,
          0);
```

[איור: דוגמא לקריאה של פונקציית WinAPI בעזרת func_array]

ניתן לפענח את מערך השמות במספר דרכים למשל פענוח ידני בעזרת אלגוריתם Ice בשימוש המפתח שמתקבל ב-Input, אך אני עשיתי זאת בעזרת Hook לאחר ביצוע הפענוח ב-Module והדפסת המערך המפוענח, לאחר מכן גיליתי שמערך השמות הוא סטטי וזהה לכל ה-Modules ולכן זה היה פתרון מלא לבעיה זו:

```
struct func_array {
 void *LoadLibraryExA;
 void *GetProcAddress;
 void *NtOpenProcess;
 void *FreeLibrary;
 void *GetVolumeInformationW;
 void *GetFileInformationByHandleEx;
 void *QueryFullProcessImageNameW;
 void *GetLastError;
 void *OpenProcess;
 void *CryptMsgGetParam;
 void *OpenSCManagerA;
 void *GetTokenInformation;
 void *CertCloseStore;
 void *WideCharToMultiByte;
 void *GetModuleHandleExA;
 void *SetFilePointerEx;
 void *FindFirstVolumeW;
 void *Module32FirstW;
 void *CryptMsgClose;
 void *GetFileVersionInfoSizeA;
 void *GetCurrentProcess;
 void *GetModuleInformation;
 void *VerQueryValueA;
```

[איור: מבנה שנוסיף ל-IDA ליצירת קוד קריא יותר (לא המבנה המלא)]

```
(function_array.FreeLibrary)(moduleHandles[j]);
```

[איור: דוגמא לקריאה לפונקציית WinAPI לאחר הוספת המבנה לעיל]

בעזרת השימוש במבנה לעיל הקוד יהפוך לקריא הרבה יותר ויהיה ניתן להתחיל להבין את פעילות המערכת, דבר אשר יעזור לנו מאוד הוא השימוש הנפוץ והברור בפונקציות WinAPI שבעזרתן נוכן להבין את המשמעות של משתנים שונים בקוד (לפי הקשר).

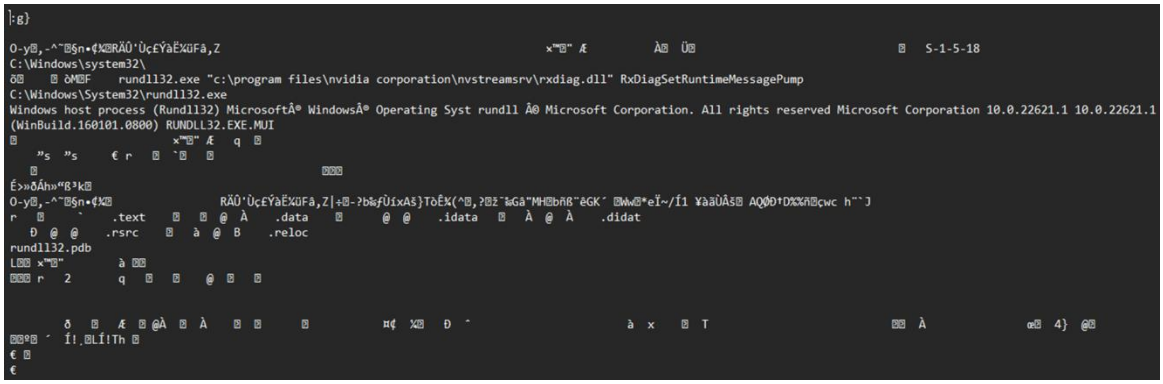
מסקנות מניתוח סטטי של Modules

כעת אפרט כמה מהסריקות העיקריות שמבצעת מערכת VAC, אציין כי יש למעלה מ-20 Modules שונים אשר בחלקם מספר סריקות לכן כמובן לא אפרט את כל הפעילות, אתמקד בשיטות לזיהוי רמאות חיצונית מ-Usermode אך אציין גם סריקות לזיהוי שאר הרמאויות.

ל-Module העיקרי (והמשמעותי ביותר) אשר מזהה רמאות חיצונית נקרא HandleList, כאמור כל רמאות שרוצה לקרוא זיכרון מהמשחק מ-Usermode עושה זאת בעזרת Handle, Module זה מנצל זאת ומשתמש בפונקצייה הלא מתועדת NtQuerySystemInformation עם פרמטר SystemInformationClass של SystemHandleInformation ובכך מקבלת את כל ה-Handles אשר פתוחים במחשב, לאחר מכן היא מסננת את אלו אשר פתוחים למשחק (לפי PID שהתקבל בקלט) ושולחת את ה-PID של התהליכים אשר פתחו אותם יחד עם ההרשאות שלהם (למשל הרשאת קריאה, הרשאת כתיבה, הרשאת שאילתת אינפורמציה ועוד) לשרת.

בעזרת שיטה זו השרת מקבל את האינפורמציה הקריטית והיחידה ככל הידוע לי אשר ניתנת להשגה מ-Usermode על אילו תהליכים קוראים זיכרון מהמשחק. חשוב לציין ש-VAC כן רוצה לתת לחלק מהתהליכים התמימים לקרוא זיכרון מהמשחק כגון svchost.exe או אנטי-וירוס כלשהו, לכן יש צורך בחקירה נוספת על תהליכים אלו אשר מהווים Handle למשחק כן לוודא שהם אכן תהליכים זדוניים.

לאחד ה-Modules אשר אחראיים לעשות חקירה זו על תהליך כלשהו נקרא Process Investigator, Module זה מקבל בקלט מהשרת PID של תהליך שיש לחקור, ומחזיר בפלט את שם וה-Path של קובץ התהליך, שמות Sections של זיכרון התהליך, חתימה רשמית של קובץ התהליך ועוד מידע מעניין נוסף:



[איור: פלט לדוגמא של ה-Process Investigator]

בעזרת מידע זה ומידע מ-Modules אחרים יכול השרת להבדיל בין תהליך תמים שקורא זיכרון מהמשחק לתהליך זדוני של רמאות. עוד Module אשר פעול בצורה דומה ל-HandleList נקרא FileMapping, הוא מחזיר שמות קבצים של תהליכים עם Handle פתוח לתהליך המשחק, ההבדל המשמעותי הוא ש-Module זה לא משיג את המידע בעזרת קריאה ישירה לפונקציות WinAPI, אלא רק מדווח מידע אשר נמצא ב-File Mapping אשר יצרה הספריה steamclient.dll.

הספרייה steamclient.dll טעונה לתוך cs2.exe (תהליך המשחק), steam.exe, GameOverlayUI.exe (ה-Overlay של Steam), והיא משיגה את שמות התהליכים ומכניסה אותם ל-File Mapping אשר ה-Module מדווח לאחר מכן (לא בהכרח ישר אחרי, אלא מתי שהשרת מבקש לטעון אותו).

ההבדל הזה מקשה על עקיפה פוטנציאלית אשר עוקפת ישירות את ה-Modules בגלל העובדה שהסריקה נמצאת בתהליכים אחרים (יהיה ברור יותר כאשר נדבר על שיטות עקיפה של WinAPI Hooking):

```
S\GameManagerService.exe
S\GameManagerService.exe
es (x86)\Steam\steam.exe
e\net6.0-windows\Neo.exe
ows\System32\svchost.exe
ows\System32\svchost.exe
ows\System32\svchost.exe
ows\System32\svchost.exe
ontainer\nvcontainer.exe
K\bin\PresentMon_x64.exe
ows\System32\audiodg.exe
ows\System32\audiodg.exe
```

[איור: פלט לדוגמא של ה-FileMapping Module]

כאמור ישנם הרבה Modules אחרים אשר עושים סריקות שונות בינהם:

- בדיקת Debug על התהליך של steam.exe
- הרצת פקודת cpuid
- בדיקה האם DSE מופעל ובדיקות על Driver-ים אשר רצים:

```
FACEIT
FACEIT
C:\Program Files\FACEIT AC\FACEIT_AC.sys
System Reserved
```

[איור: פלט לדוגמא של Module אשר חוקר Driver רץ]

בנוסף קיימות כמובן בדיקות עבור רמאות פנימית, אשר בקצרה עובדות כך:

ריצה על הספריות הטעונות לתוך התהליך של המשחק על ידי שימוש ב-EnumProcessModules, השגת מידע נוסף כגון שם קובץ מקור ועוד על ידי שימוש ב-VirtualQueryEx.

שיטות לניתוח דינמי של Modules

בנוסף לניתוח סטטי של ה-Modules כלי שימושי מאוד שעוזר במחקר המערכת הוא ניתוח דינמי (דיבוג).

הבעיה בדיבוג של Modules היא שהם נטענים במקום לא ידוע בזיכרון, בזמן לא ידוע תוך כדי משחק לכן נרצה למצוא דרך קלה יותר בה נוכל לדבג אותם בקלות מתי שאנחנו רוצים.

כאמור לעיל יש לנו Dump של הספריות של ה-Modules, והקלטים שהם מקבלים בהתאם, לכן נוכל לרשום תוכנה קצרה שתסמלץ הרצת Modules על ידי טעינתו לתוך הזיכרון של עצמה והרצתו, לאחר מכן נדבג את אותה תוכנה עצמה ובכך את ה-Module לאחר טעינתו:

```
int main()
{
    //string path = "C:\\Users\\ilana\\Desktop\\Volvo Anti Cheat\\dumps\\";
    string path = "C:\\Users\\Ilan\\Desktop\\debug test\\";
    //string path = fs::current_path().string() + "\\";
    string hash;
    cout << "Enter dump hash: " << endl;
    cin >> hash;
    string dumpDLL = "dump_" + hash + ".dll";
    string dumpParams = "params_" + hash + ".txt";
    string paramFile = path + dumpParams;
    string dumpPath = path + dumpDLL;
    auto dumpLPCSTR = dumpPath.c_str();

    cout << "Loading VAC Module: " << dumpDLL << endl;
    HMODULE hModule = LoadLibraryA(dumpLPCSTR);
    if (!hModule)
        cout << "Error executing LoadLibraryA: " << GetLastError() << endl;

    // find runfunc of module (module entry point)
    runfunc module_main = (runfunc) GetProcAddress(hModule, "_runfunc@20");
    int resultSize = RESULT_SIZE;
    DWORD result[RESULT_SIZE] = {0};

    // get params and original input from dumped file
    int paramSize = GetParamSizeFromFile(paramFile);
    int funcId = GetFuncIdFromFile(paramFile);
    void* params = GetParamsFromFile(paramFile, paramSize);
    // for dump1 processId
    //((int*)params)[24] = 2640;
    int retVal1 = module_main(funcId, params, paramSize, result, &resultSize);
    cout << retVal1 << endl;
    return 0;
}
```

[איור: דוגמא לתוכנה אשר טוענת Module, מריצה אותו עם קלט רצוי]

היכולת לדבג את ה-Modules היא כלי שימושי מאוד אשר יתן לנו להתבונן בפעילות ה-Module ולראות ערכים של משנתים שונים ופעילות של פונקציות מסובכות אשר קשה לנתח סטטית, בנוסף כך נוכל לראות את פלט ה-Modules לפני ההצפנה (אדבר על זה בהמשך).

עקיפת מערכות למניעת רמאות ובפרט VAC

עקיפה על ידי שימוש בשיטות לא מזוהות על ידי המערכות

השיטה הפשוטה ביותר והכי פחות מעניינת היא למידת המערכת ושימוש בשיטות עקיפות שהיא לא מזהה (כגון: Kernel-Mode).

במקרה שלנו מ-Usermode ניתן לנסות דברים כמו "לגנוב" Handle של תהליך אחר שלא יהיה חשוד שיהיה לו Handle לתהליך המשחק למשל svchost.exe על ידי פונקציות כמו DuplicateHandle, או על ידי הזרקה לתהליך כזה וביצוע פעולות הזיכרון ממנו, אך אלו דברים שלא בדקתי לכן כדאי לקחת בערבון מוגבל.

אותו דבר תקף גם עבור שיטת הציור שלנו על המסך, יש ללמוד את הזיהוי של המערכת עבור זיהויים כאלו (במקרה שלנו בדיקות ל-Hooks לפונקציות שאחראיות לציור במשחק) ועקיפה בעזרת שיטה לא מזוהה (כגון Overlay Hijacking מתהליך לא חשוד).

עקיפה בעזרת שינוי פונקציונליות המערכת

שיטת WinAPI Hooking

כל מערכת לזיהוי רמאות עובדת בצורה דומה: הלקוח מקבל קלט מהשרת (במקרה שלנו Module וקלט לאותו ה-Module), והלקוח מחזיר פלט לשרת.

המטרה שלנו הוא שאותו הפלט יהיה לא חשוד וכך לשרת לא תיהיה סיבה לחשוד בנו, יש מספר רחב של דרכים ליצור פלט כזה, אפרט על כמה מעניינות.

שיטה אחת לייצור פלט כזה היא בעזרת Hooks לפונקציות של WinAPI שכאמור הם כמעט הדרך היחידה של המערכת (בגלל שהיא Usermode) להשיג מידע על תהליכים אשר רצים במחשב, ועוד מידע אחר.

ה-Hooks שלנו יהיו כמובן עבור תהליך מסויים כיוון שאנחנו רצים ב-Usermode (כנראה Inline או IAT/EAT Hook).

דוגמא אחת לשימוש בשיטה היא למשל לעשות Hook לפונקציה NtQuerySystemInformation בתהליך steam.exe (כי בו רצים ה-Modules) ומחיקת ה-Handles של תהליך הרמאות למשחק, אפשר לדוגמא לעשות Hook גם ל-OpenProcess למניעת חקירה של תהליך הרמאות ודברים כגון VirtualQuery עבור רמאות פנימית.

היתרון העיקרי בשיטה זו היא שאנו לא נוגעים ישירות בפונקציונליות ה-Modules אלא רק מהווים Proxy בינהם לבין ה-WinAPI, ולכן לא נצטרך לעקוף דברים כמו Integrity Checks על ה-Modules.

קיימים שני חסרונות עיקריים לשיטה:

הראשון הוא שכמובן ניתן לזהות אותם Hooks בקלות יחסית על ידי Integrity Checks והשוואה למשל ל-ntdll.dll המקורי של המערכת, או השוואת כתובות במקרה של IAT/EAT Hooks.

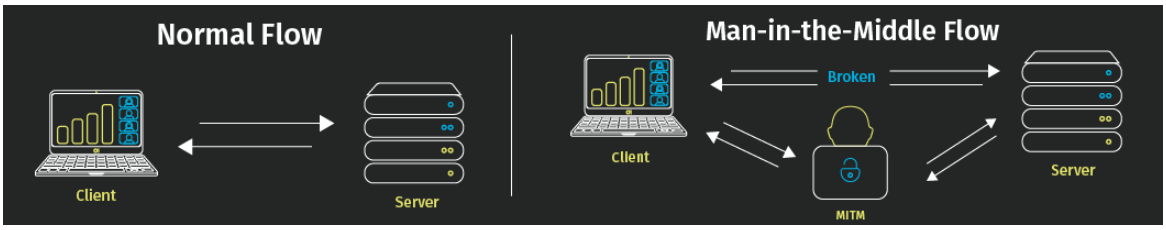
החסרון השני הוא שקיימים חלקים ב-VAC אשר אינם רצים ב-steam.exe כמו למשל ה- Filemapping Module שמדווח מידע אשר נכתב מתוך תהליך המשחק ותהליכים אחרים בנוסף ל-steam.exe, את זה נוכל לעקוף ע"י Hooks גם בתהליכים אלו אך זה חושף אותנו להרבה יותר שיטות זיהוי, ולרוב נרצה גם להזריק לתוך תהליך המשחק על מנת לבצע Hooks אלו (לא תמיד, אפשר גם חיצונית בעזרת כתיבות לזיכרון אך קשה כי נצטרך למצוא\להקצות זיכרון עבור ה-Hooks שלנו) ובכך הפכנו פרקטית לרמאות פנימית.

שיטת ה-Patching\שינוי פונקציונליות ישירה

שיטה זו דומה לשיטה הקודמת, אך בה נשנה את פונקציונליות ה-Modules בצורה ישירה, במקרה של VAC, שיטה זו פותרת את החסרון השני של השיטה הקודמת (קוד אשר רץ בתהליכים אחרים), מכיוון שה-Modules בכל מקרה אחראים על הדיווח ולכן נוכל לשנות את הדיווח).

דרך אחת בה ניתן לשנות את פונקציונליות ה-Module היא לעשות Hook/Patch לקוד ה-Module, אך זה קל לזיהוי בעזרת Integrity Checks בכל הרצת Module (ועושים זאת), ניתן כמובן לעשות Hook גם לבדיקה וזה באמת יעבוד (מעיין חתול ועכבר של Hooks), זה כמובן ידרוש קצת מחקר נוסף של הדרך בה מבוצעים ה-Integrity Checks, במקרה של VAC - פונקציה די פשוטה.

מתקפת Man in the middle



[איור: המחשה של MiTM]

שיטה זו היא השיטה שבחרתי לממש, מימשתי ובדקתי אותה ואני יכול להבטיח שהיא עובדת לפחות נכון לפרסום מאמר זה, אם הייתי Valve הייתי משנה את המערכות עליהן ארחיב בהמשך, אך לפי ההיסטוריה שלהם זה כנראה לא יקרה בקרוב (נראה שכבר פחות אכפת להם מהמערכת צד לקוח, הם מנסים לעשות

משהו צד-שרת אך ביינתים לא רואים את זה כל כך). החולשה העיקרית ב-VAC היא המחסור בזיהוי Hooking עבור פונקציות אשר אחראיות על הרצת ה-Modules (כגון load_vac_module), ונוכל לנצל אותה קשות.

ניתן לחשוב שהוספה פשוטה של בדיקות של Hooks לפונקציות אלו יעזרו מאוד, אך כאמור ניתן לעשות Hook גם לבדיקות עצמן ובכך בעזרת מחקר המערכת (גם אם הוא ארוך וקשה) תמיד יהיה ניתן לעשות זאת.

הערה: קיימות שיטות כגון VEH Hooking שקשה מאוד עד לא אפשרי לזהות מ-Usermode שיוכלו לעזור לנו בכל מקרה.

ניתן לנצל את החולשה הזאת ולממש מתקפה קלאסית של MiTM, לעשות Hook לפונקציה שמריצה Module, ולפני החזרת הפלט לשרת, לשנות אותו כרצוננו, וכך אנחנו יכולים להישאר מתחת לרדאר מבלי לשנות ישירות פונקציונליות של ה-Modules עצמם (חוסך לנו כאב ראש, שיטה פשוטה ואלגנטית).

כעת הבעיה הופכת לבעיה של ייצור פלט לא חשוד עבור קלט מסויים מהשרת (Module וקלט בשבילו), לבעיה זו קיימים כמה פתרונות מעניינים יותר ופחות, עליהם אפרט מעט:

ראשית, ניתן כמובן לחקור את כל ה-Modules ולייצר גרסאות אשר יצרו פלט לא חשוד עבורנו, לשיטה זו קוראים אמולציה והיא הפתרון ה-"מושלם" אך גם הקשה ביותר, מכיוון שהמחקר יכול לקחת זמן הרבה מאוד עבודה, ויש לייצר פלט זהה במדוייק לפלט המקורי (למעט החלקים החשודים), אחרת השרת יחשוד כי פורמט הפלט לא מדוייק.

שיטה מעניינת אחרת היא להריץ את ה-Modules בסביבה בטוחה, למשל בשרת כלשהו, וכך לייצר פלט לא חשוד. החסרון בשיטה זו היא שחלק מהקלטים שמקבלים מהשרת הם תלויי סביבה למשל PID של תהליך המשחק, כמובן שגם קלטים אלו מגיעים בסופו של דבר ממידע שנשלח לשרת מהלקוח ולכן נוכל להתאים כך את כל התקשורות לסביבת השרת, אך גם זה ייקח לא מעט מחקר של התקשורת. (לדעתי שיטה טובה מאוד אך התעצלתי לממשה)

לבסוף יש את השיטה העצלנית-אך-עובדת שהיא להשתמש בפלט שייצר ה-Module האמיתי ורק לנקות את החלקים החשודים, למשל ב-Module אשר מחזיר Handles לתהליך המשחק, למחוק את ה-Handle של הרמאות שלנו וכך לכל Module פוטנציאלי שיכול לזהות את שיטת הרמאות שלנו, גם שיטה זו דורשת מחקר יחסית מעמיק של ה-Modules והפלט שהם יוצרים, אך ניתן לנצל את העובדה שהמערכת תמיד תרצה לחקור תהליכים שנראים לה חשודים, ולכן נוכל לראות מתי תהליך הרמאות נחשד על ידיה, על ידי הסתכלות על הפלטים (כגון אלה שהראתי במאמר, ניתן לחפש אם מתבצע מחקר על תהליך הרמאות שלנו למשל לפי חיפוש שם התהליך).

השיטה האחרונה היא זו שהחלטתי לממש, אך נתקלתי במכשול חשוב: בדיוק למניעת מתקפות MiTM, ה-Modules מצפינים את הפלט שלהם ולכן לא נוכל לנקות אותו כיוון שאנחנו לא יודעים את הפלט האמיתי.

את בעיית ההצפנה ניתן לפתור בשיטת ראש בקיר, לעשות עוד Hook בתוך ה-Module ולקרוא את מפתח ההצפנה, אך זה מוסיף עוד כאב ראש של בדיקות אמינות של ה-Modules שדיברתי עליהן לעיל, ובנוסף זו דרך לא אלגנטית לטעמי (אם כי אולי קלה יותר למשל בעזרת VEH Hooking).

כאשר חקרתי את דרך ההצפנה הבנתי דבר חשוב: ה-Module כמובן כן רוצה שהשרת יבין את הפלט, לכן הוא "מחביא" את מפתח ההצפנה בתוך הפלט (כמובן שלא בדרך פשוטה, אלא בדרך די גאונית לדעתי), אם נוכל להבין איך להשיג את המפתח מתוך הפלט, יהיה לנו פתרון לבעיית ההצפנה ונוכל לבצע מתקפת MiTM בעזרת Hook יחיד בפונקציית הרצת ה-Module (שכאמור אין עליה בדיקות).

משימה לקורא: לחשוב למה לא כדאי להשתמש במשהו כמו SSL במקום שיטה לא מושלמת להחבאת המפתח.

חילוץ מפתח ההצפנה מהפלט

הגענו לחלק שלדעתי הכי מעניין במאמר, המידע שאפרסם בחלק זה לא נמצא בשום מקום ציבורי באינטרנט והוא במלואו תוצאה של המחקר שלי.

שיטת החבאת המפתח

נתבונן בקוד אשר אחראי להצפנת הפלט:

```
v16 = __rdtsc();
initial_seed[1] = 0;
flOldProtect[1] = HIDWORD(v16);
index = 0;
initial_seed[0] = -abs32(v16);
do
  encryption_key[index++] = random(initial_seed);
while ( index < 0x20 );
return_buffer_size = *a4;
v35 = 0;
encryption_key[1] = return_buffer_size;
encryption_key[0] = encryption_key[0] & 0xFFFFFF7F | 0x40;
for ( j = return_buffer_size >> 2; j < 0x422; ++j )
  ret_buff[j] = random(initial_seed);
v28 = dword_75517574;
v20 = sub_75513D08(v39, v29, ret_buff + 1026, &v28, flOldProtect);
ret_buff[1024] = v28;
v21 = 0x10000;
if ( v20 )
  v21 = 0;
v22 = v27;
ret_buff[1025] = flOldProtect[0] | v21;
IceEncrypt((int)ret_buff, v22, 0, (int)v34);
for ( k = 0; k < 0x20; ++k )
  random(initial_seed);
```

[איור: הקוד אשר אחראי ל"החבאת" המפתח בפלט והצפנת הפלט]

כיוון שבאיור לעיל יש הרבה שורות לא קשורות אפשט את החלק החשוב ל-Pseudocode:

```
int seed_arr[34] = {0};
seed_arr[0] = __rdtsc(); // https://c9x.me/x86/html/file_module_x86_id_278.html
int output_arr[0x20];
for (int i = 0; i < 0x20; i++)
    output_arr[i++] = random(seed_arr);

uint64_t ice_key = *((uint64_t *)&output_arr[2]); // ice_key is output_arr[2], output_arr[3]

for (int i = 0; i < 0x10; ++i )
      output_buffer[0x1000 + i] = random(seed_arr);
```

[איור: הקוד המפושט ל"החבאת" המפתח בפלט והצפנת הפלט]

ניתן לראות בקוד שה-Module מייצר מפתח הצפנה (ההצפנה מתבצעת באמצעות אלגוריתם סימטרי המכונה Ice, ולכן מספיק להשיג את המפתח) בעזרת פונקציה לייצור מספרים אקראיים (Pseudorandom number generator) עם Seed כלשהו (ה-Seed הספציפי הוא הפלט של פקודת __rdtsc שמביחנתנו אקראי ואין דרך לצפות אותו), לאחר מכן ה-Module מייצר מספרים נוספים בעזרת אותה פונקציה ומוסיפה אותם לפלט לשרת.

ידוע לנו כמובן שהשרת יודע להבין את הפלט ולכן בהכרח יודע להוציא את מפתח ההצפנה מתוך הפלט, אך איך הוא עושה זאת? כדי להבין, יש להתבונן בפונקציה random. הפונקציה random ממושת כך:

```
int VAClcg_orig(int seed_arr[100])
{
        if (*seed_arr <= 0 || (seed[1] = seed_arr[1]) == 0)
        {
                // this is first run
                seed_arr[0] = -seed_arr[0];
                v4 = seed_arr + 41;
                if (seed_arr[0] < 1)
                        seed_arr[0] = 1;
                for (i = 39; i >= 0; --i)
                {
                        int num = 16807 * seed_arr[0] - 0x7FFFFFFF * (seed_arr[0] / 127773);
                        seed_arr[0] = num + 0x7FFFFFFF;
                        if (num >= 0)
                                seed_arr[0] = num;
                        if (i < 32)
                                *v4 = seed_arr[0];
                        --v4;
                }
                seed_arr[1] = seed_arr[2];
        }
        // https://en.wikipedia.org/wiki/Lehmer_random_number_generator#Schrage's_method
        // this is the same as (16807 * (long long)seed[0]) % 2147483647
        // this avoids overflow in 16807 * seed0
        int num = 16807 * seed_arr[0] - 0x7FFFFFFF * (seed_arr[0] / 127773);
        seed_arr[0] = num + 0x7FFFFFFF;
        if (num >= 0)
                seed_arr[0] = num;
    // this is based on last result
        int result_index = seed_arr[1] / 0x4000000 + 2;
    // this is some previous seed, we need to find out which one
        seed_arr[1] = seed_arr[result_index];
        seed_arr[result_index] = seed[0];
        return seed_arr[1];
}
```

[איור: הפונקציה random]

החלק בתוך התנאי רץ רק בהרצה הראשונה והוא פרט לא כל כך מעניין, לכן נתרכז בחלק שאחרי. האלגוריתם יוצר מספר "אקראי" עם ה-seed הנתון בעזרת LCG (Linear congruential generator) בשיטת Schrage's Method (לא כל כך חשוב, נאמר רק בשביל לזהות את הפונקציונליות).

הערה: לא אפרט מהו LCG ואיך הוא עובד, רק חשוב לדעת שניתן לעשות את הפעולה ההפוכה ממנו כלומר בהינתן פלט של LCG להשיג את ה-Seed הקודם, להבנה מדויקת איך זה נעשה ממליץ לקרוא כאן וכאן.

לאחר מכן, היא מחשבת אינדקס כלשהו result_index ומחזירה את הערך של seed_arr[result_index] (שיוצר מספר כלשהו של איטרציות קודם לכן), הפונקציה גם מציבה ב-seed_arr[result_index] את המספר אשר יוצר באיטרציה הנוכחית.

במילים אחרות, האלגוריתם מערבב את המספרים שהוא יוצר במערך בצורה כלשהי.

מכיוון שכאמור אנו יודעים להחזיר את פעולת ה-LCG, ניתן לחשוב שאנו יכולים לקחת את המספרים שיוצרו שנתונים לנו בפלט, וללכת אחורה מהם עד שנגיע ל-Seed המקורי (ואיתו כמובן ליצור את מפתח ההצפנה), אך איך נדע כמה איטרציות אחורה לחזור מתוך המספרים אשר נתונים לנו?

אם לא החלק המערבב של האלגוריתם, היינו יכולים לדעת בקלות כמה איטרציות היו אחרי ייצור המפתח, וכך לדעת כמה ללכת אחורה, אך בהינתן זה שפלט הפונקציה אינו המספר שיוצר באיטרציה הזו, אלא מספר מאינדקס כלשהו במערך שיוצר מספר מסויים של איטרציות קודם לכך, איך נוכל לדעת כמה איטרציות היו לאחר ייצור המפתח?

הפתרון לבעיית הערבוב

ראשית, נתבונן בכך שנוכל לדעת מה יהיה ה-result_index של האיטרציה הבאה של הפונקציה, בהינתן הפלט של הפונקציה (על ידי 0x4000000 * (result - 2)).

בנוסף לכך, נתונים לנו כמה מספרים אשר יוצרו על ידי האלגוריתם אחד אחרי השני ברצף.

בגלל איך שעובד הערבוב, אם שני איטרציות מקבלות אותו result_index (שאנו יודעים איך לחשב מפלט האיטרציה הקודמת), הפלט של האיטרציה השנייה מחזיר את המספר שיוצר באיטרציה הראשונה, שאנו יודעים מתי היא קרתה (כלומר יודעים כמה יש לחזור ממנה כדי להגיע ל-Seed המקורי).

במקרה כזה יהיה לנו את כל מה שאנחנו צריכים, מספר שיוצר ומתי הוא יוצר (כמה איטרציות אחרי ה-Seed המקורי), נוכל "ללכת אחורה" מהמספר הזה את כמות הפעמים הנחוצה כדי להגיע ל-Seed המקורי ולייצר את מפתח ההצפנה בעצמנו, רווח!

הערה: מבדיקה, תמיד קורה המקרה של result_index כפול. בנוסף, הפלטים שהצגתי במאמר זה הושגו בשיטה זו.

אני משוכנע שאלגוריתם זה להשגת מפתח ההצפנה מתוך המספרים שה-Module נותן לנו בפלט הוא זהה לאחד שרץ בשרת של Valve (וזה לדעתי מגניב בטירוף). קישור לקוד המלא שלי להשגת מפתח ההצפנה

סיכום

לסיכום, המאמר זה דיברנו על סוגי רמאויות שונים במשחקי מחשב, ושיטות לזיהוי רמאויות אלו. לאחר מכן התמקדנו במערכת VAC למקרה של רמאות חיצונית, תחילה דיברנו על דרכים שונות למחקר סטטי ודינמי שלה, ולאחר מכן דיברנו על פעולת המערכת והשיטות הספציפיות שהיא משתמשת בהן.

דיברנו על רעיונות שונים לעקיפת VAC בפרט אך גם מערכות למניעת רמאות בכלל (אם כי בעיקר כאלו אשר רצות ב-Usermode).

הסברנו את מתקפת ה-MiTM, הפעילות שלה והיתרונות שלה, ולמדנו "לעקוף" את ההצפנה של VAC שנועדה להקשות עלינו לבצע תקיפה זו, וכך לעקוף את זיהוי הרמאות של VAC על ידי ניקוי הפלטים שלה.

אציין שמימשתי בעצמי ובדקתי את מתקפת ה-MiTM ואני יכול לאשר שהיא אפשרית וקלה יחסית לעקיפה מלאה של VAC נכון לכתיבת מאמר זה.

על המחבר

אילן ארנגאוז, בן 18 מאשדוד. בעת כתיבת המאמר אני מלש"ב אשר עתיד להתגייס למסלול גאמא סייבר. מגיל צעיר לומד וחוקר בתחום המחשבים, בזמן האחרון מתעסק בעיקר בתחום הסייבר אך מתעניין גם בתחומים אחרים. ה-Github שלי:

https://github.com/shuruk421

לכל שאלה או דבר לא ברור תוכלו לפנות אלי במייל:

ilanarengauz1@gmail.com