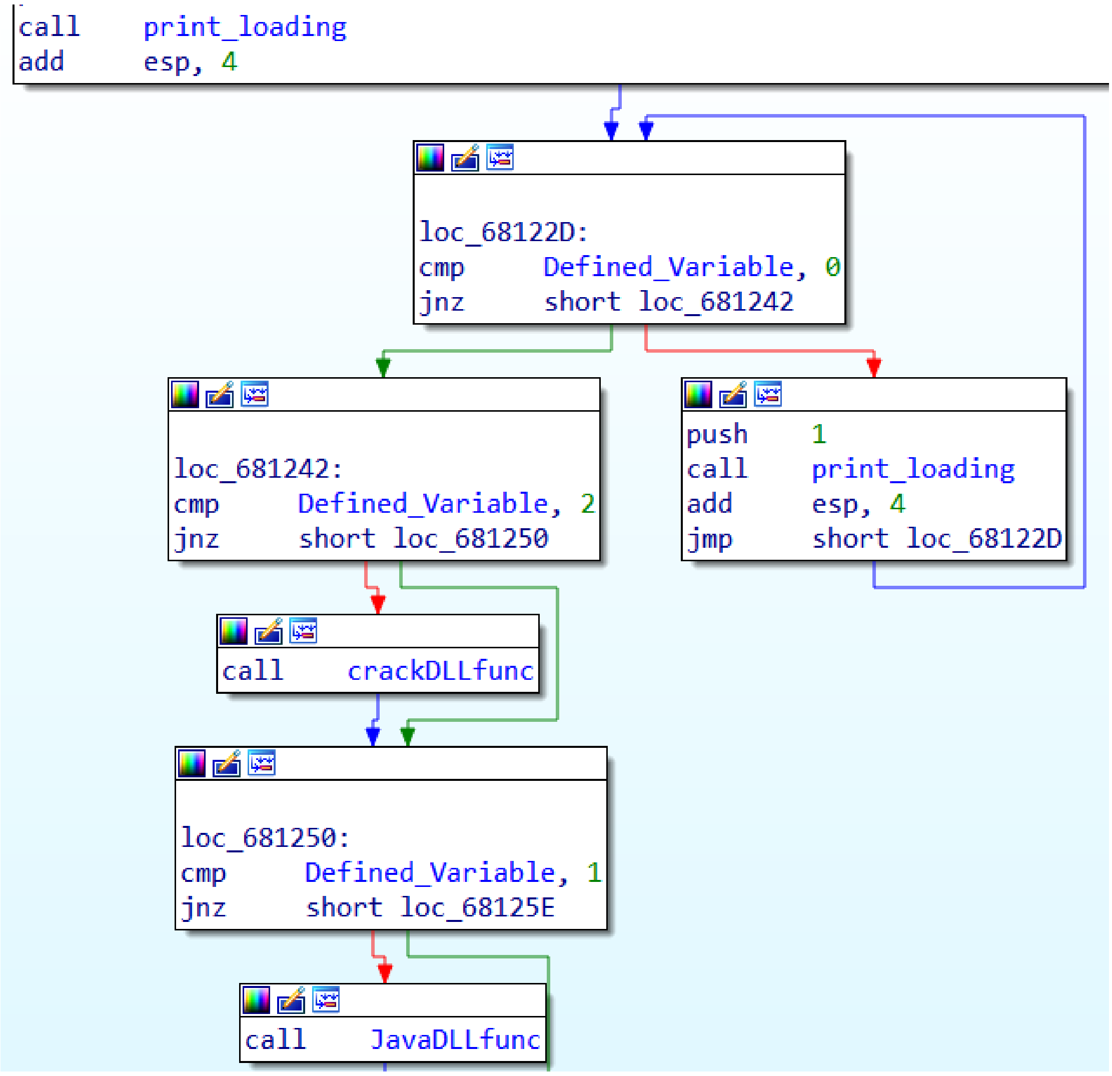
Final Assignment – Reversing 2024 – Roey Gross's Writeup

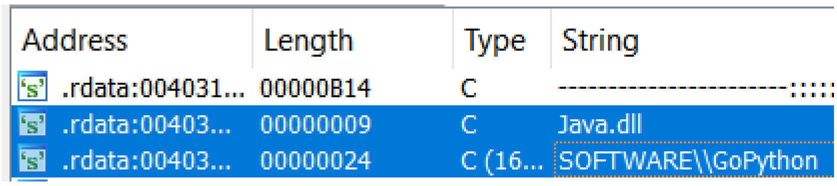
Python

I opened the exe in IDA, started debugging main and noticed that the flow depends on a variable that has been already miraculously defined, while before the debugging, it was undefined. The running of the program defines it, but where?



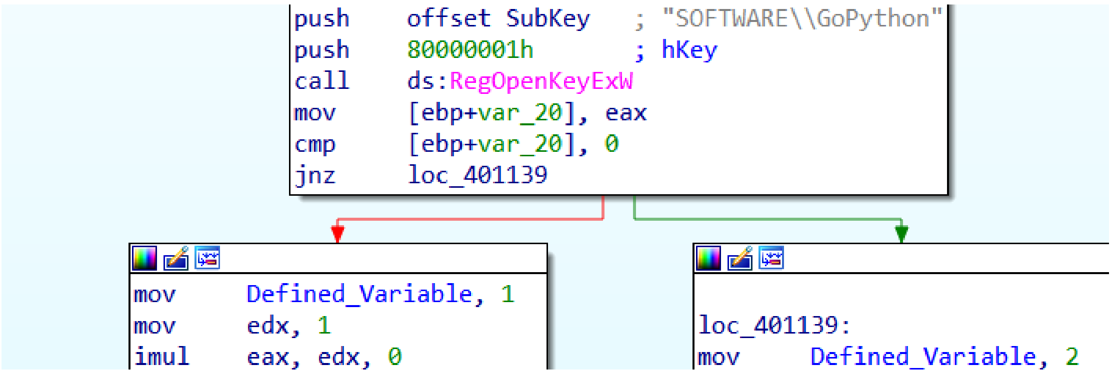
To continue with the main flow and not get stuck at the printing function, we need to find the mysterious part of the program that defines the variable and make sure it defines it to something different from 0.

I have realized that something must be activated before the entry point of main. I looked for some clues and went to the strings window. I found unique strings from a function called TlsCallback_0.



Address	Length	Type	String
.rdata:004031...	00000B14	C	----------------------:::::
.rdata:00403...	00000009	C	Java.dll
.rdata:00403...	00000024	C (16...	SOFTWARE\\GoPython

This function, creates a new thread, and runs a code that defines with the mysterious variable from the main function:



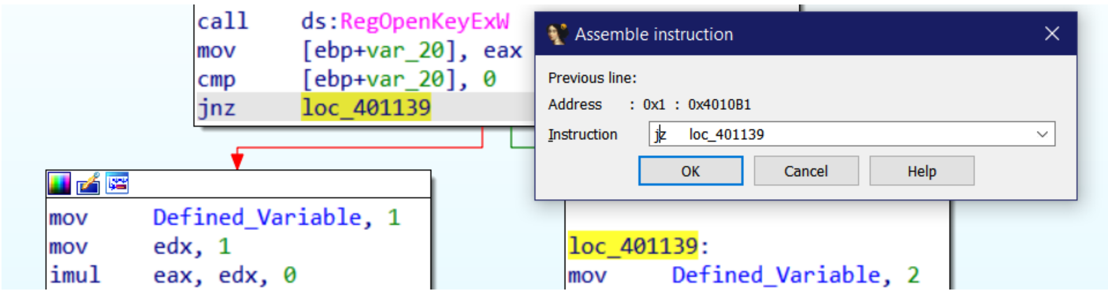
```
push    offset SubKey   ; "SOFTWARE\\GoPython"
push    80000001h       ; hKey
call    ds:RegOpenKeyExW
mov     [ebp+var_20], eax
cmp     [ebp+var_20], 0
jnz     loc_401139

mov     Defined_Variable, 1
mov     edx, 1
imul    eax, edx, 0

loc_401139:
mov     Defined_Variable, 2
```

BINGO! – Here's where the "predefined" variable is defined! At the Tls_Callback function! The variable is changed based on the existence of a key named "GoPython" inside the registry at the "SOFTWARE" file.

I noticed that there is no use of this registry key at the rest of the program, so I decided to patch the condition.



Adding keys can be dangerous in some situations, while patching the condition when possible - is safer. Now the defined variable is equal to 1.

Parallelly, I wondered how the Tls_Callback function was running before the main, and why my debugger didn't notice it.

I found this recommended article from the digital whisper magazine about TLS callbacks: https://www.digitalwhisper.co.il/files/Zines/0x62/DW98-2-TLSCallBacks.pdf

TLS callbacks are created for thread-specific initialization and cleanup tasks. A program can use a TLS callback to execute code before the main entry point of the program. It enables the program to operate covertly without the debugger notice, as the debugger will typically start at the main entry point and may not be aware of the TLS callback!

Back to the function. The flow inside the TLS callback continues with a decryption algorithm that turns a ciphertext into a plaintext called "Java.dll".

```
.data:00E77018 LibFileName     db 'Mfqf)ckk',0
.data:007A7018 LibFileName     db 'Java.dll',0
```

At the end of this decryption, we go into a function that creates and opens for writing a dll file called Java.dll. (If the dll already exists and has content, the function overrides it all. If the dll doesn't exist it creates a new one.)

```
push    offset Mode     ; "wb"
push    offset FileName ; "Java.dll"
lea     eax, [ebp+Stream]
push    eax             ; Stream
call    ds:fopen_s
```

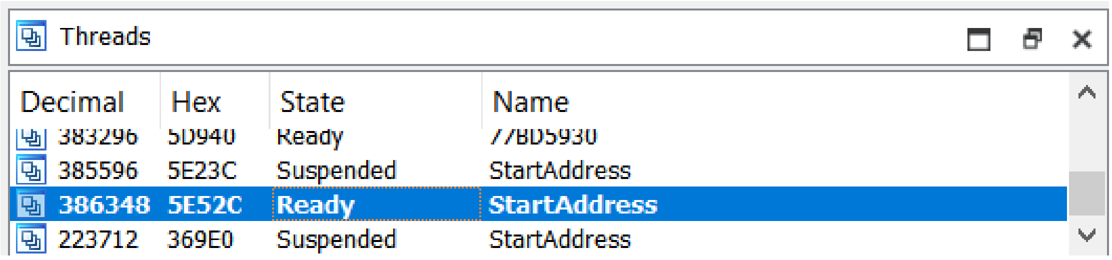
Then it writes to the Java.dll a very big number of bytes and closes the stream.

```
mov     ecx, [ebp+Stream]
push    ecx             ; Stream
push    2400h           ; ElementCount
push    1               ; ElementSize
push    offset unk_683D68 ; Buffer
call    ds:fwrite
add     esp, 10h
mov     [ebp+var_10], eax
mov     edx, [ebp+Stream]
push    edx             ; Stream
call    ds:fclose
```

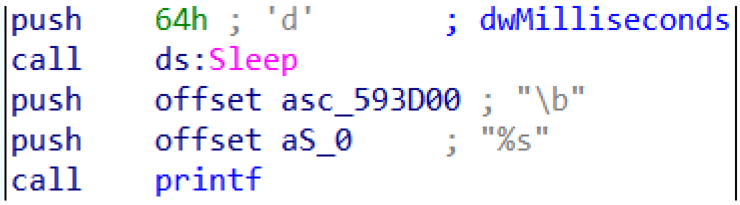
That's THE END of the TlsCallback function.

To sum up, I found that TLS Callbacks can run before the main function, then to continue without touching the registry I patched a condition, so now the defined variable equals 1. Then it decrypts a ciphertext to "Java.dll" string. Lastly, it creates a file named Java.dll and writes inside huge number of bytes.

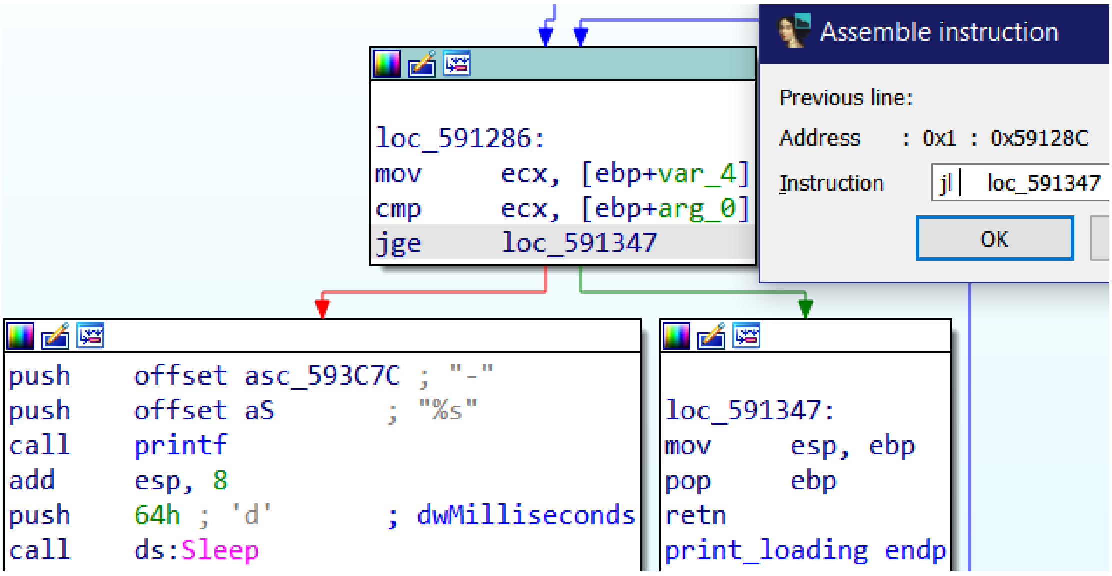
While debugging I noticed that the program is multithreaded - Randomly, IDA decided to jump to the beginning and run from there, then it switched back and forth between parts of the program. I noticed that every time it returns to the beginning - it creates a new thread. To disable this mess, I had to suspend every new thread that returned me to the start:



Another thing that slowed down my debugging process was the graphic printing. It has many Sleep functions that caused the thread to wait a lot:



I patched the condition before the printing, to disable the prints and sleeps.



Java

Back to main. As I mentioned before, the variable changed from 0 to 1. So, the main leads us to a new function I called the "JavaDLL" function.

```
loc_681250:
cmp     Defined_Variable, 1
jnz     short loc_68125E

call    JavaDLLfunc
```

The function uses the "Java.dll" string that was decrypted before at the TLS callback, and tries to load the Java.dll that the TLS callback has created before.

```
push    offset LibFileName ; lpLibFileName
call    ds:LoadLibraryA
mov     [ebp+hModule], eax
cmp     [ebp+hModule], 0
```

Before loading the unknown Java.dll let's check and open it in HxD.

```
Java.dll
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  CC 09 11 53 82 53 81 53 85 53 81 53 7E AC 81 53  Ì..S‚S.S…S.S~¬.S
00000010  39 53 81 53 81 53 81 53 C1 53 81 53 81 53 81 53  9S.S.S.SÁS.S.S.S
```

That doesn’t look like a DLL! There is no "MZ" at the beginning as should be! Also, usually in DLLs, there are a lot of repeating zero sequences. But here, there are only repeating sequences of "8153"!

The repeating sequence led me to the thought - maybe there is some kind of encryption on the DLL?

Back to main. If the Java.dll is loaded successfully, the program opens and runs a function from Java.dll called "XorForever":

```
push    offset ProcName ; "XorForever"
mov     eax, [ebp+hModule]
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+var_8], eax
```

I considered that as a clue and tried to figure out some XOR decryption on the DLL. Usually there are repeating sequences of zeros, but from the repeating sequence of 83-51, I understood that the sequences of zero were XORed by a key of 83-51. (because - A XOR 0 = A, so 83-51 XOR 00-00 = 83-51).

I tried to decrypt the first two bytes that are usually MZ with the 83-51 key:

```
CC 09
XOR
81 53
=
4D 5A <=> M Z
```

BINGO! The first two bytes are decrypted to MZ if the key is 83-51! I found the key for decrypting the DLL!

I wrote a C program that opens Java.dll, decrypts it with the key, and writes back the decryption inside a new DLL called DecJava.dll .

```
int main() {
    xor_decrypt("Java.dll", "DecJava.dll");
    return 0; }
```

"xor_decrypt" calls for the opening of the DLLs, and allocates a buffer for reading from Java.dll. It reads from Java.dll to the buffer and then decrypts the buffer. This is the important part of the decryption:

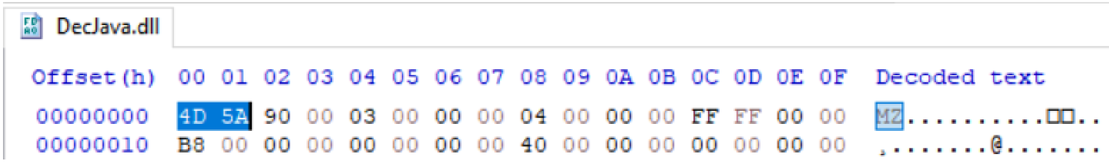
```
// Decryption
// XOR each first byte with 0x81 and each second byte with 0x53
char first, second;
for (int i = 0; i < bytes_read; i += 2) {
    first=buffer[i];
    first ^= 0x81;
    buffer[i] = first;
    second = buffer[i+1];
    second ^= 0x53;
    buffer[i + 1] = second;
}
```

Explanation: The Encryption is repeatedly XOR with key of 0x8153. Thus, the decryption is - for every two bytes from the Java.dll, the first one is XORed with 0x81, and the second one is XORed with 0x53.

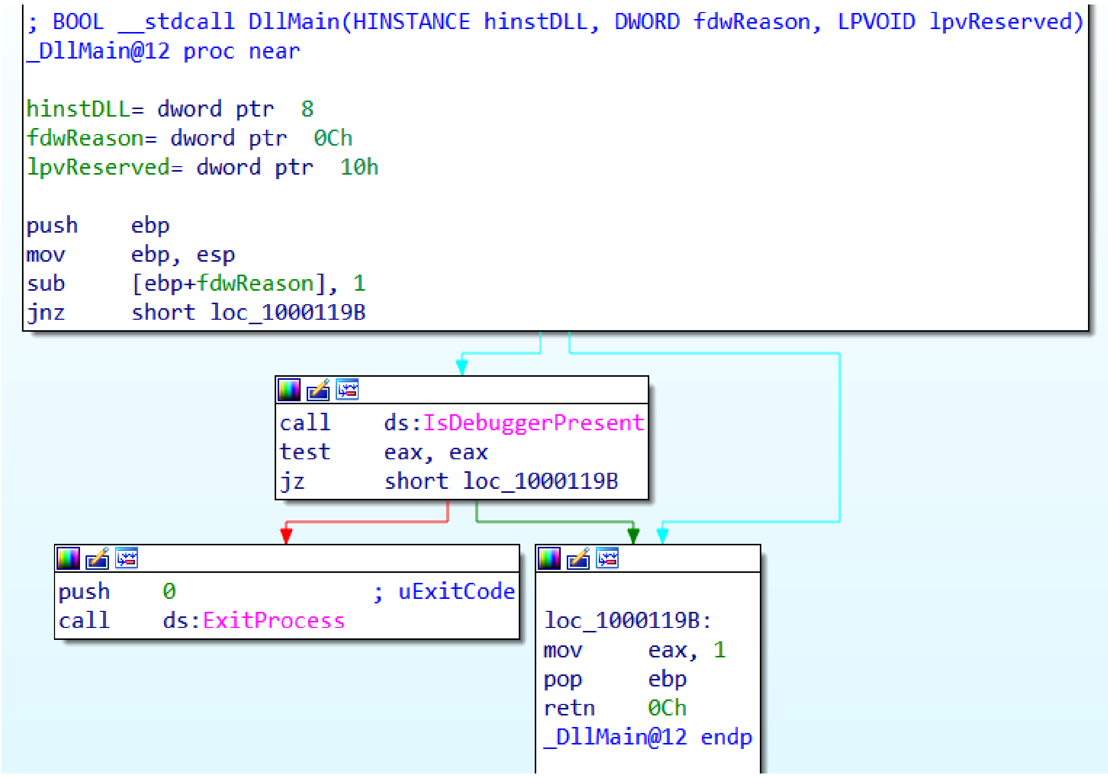
Then, it writes to the DecJava.dll the decrypted buffer.

```
// Write decrypted buffer to DecJava.dll
fwrite(buffer, 1, bytes_read, output_file);
```

After the decryption DecJava.dll looks normal, with the MZ at the beginning, and repeating zeros.

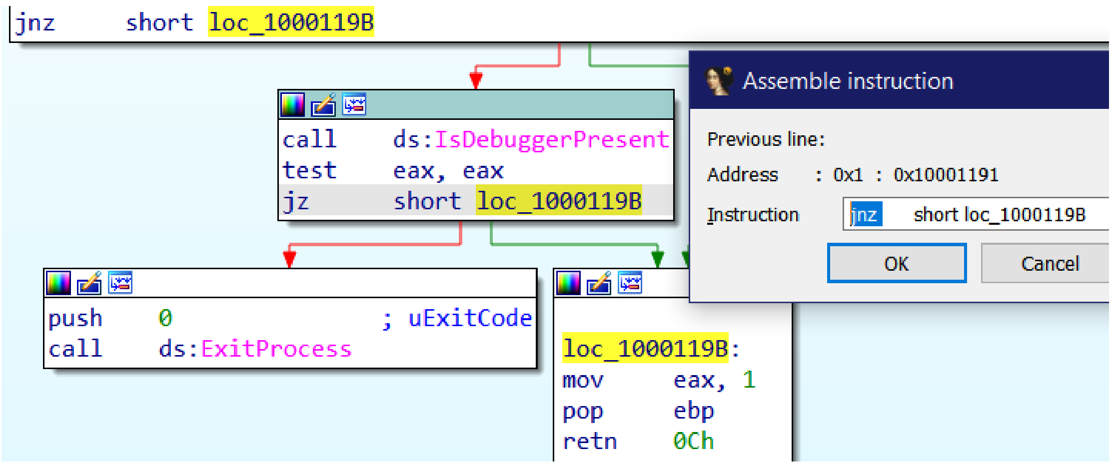


Before loading the unknown DecJava.dll in main, let's check it in IDA.



Surprise! there is a debugger checking at the DLL main.

Let's patch it and update it, so we will be able to debug it.



After patching the anti-debugging in the DLL main, lets continue to the XorForever function. It looks like we need to find a password:

```
push    offset Format   ; "Password:\n"
movsw
movsb
call    fprintf
add     esp, 4
lea     eax, [esp+70h+Arglist]
push    15h
push    eax             ; Arglist
push    offset aS       ; "%s"
call    fscanf
```

Afterwards, there is a validation checking of the password that has two conditions:

1. The password's length is not 0.
2. The ASCII sum of the characters is equal to 1337.

After the validation there is a building of an URL!!!!

```
movzx   eax, [esp+70h+var_2D]
mov     [esp+70h+ArgList], al
movzx   eax, [esp+70h+var_37]
mov     [esp+70h+ArgList+1], al
movzx   eax, [esp+70h+var_4E]
mov     [esp+70h+ArgList+2], al
movzx   eax, [esp+70h+var_49]
mov     [esp+70h+ArgList+3], al
movzx   eax, [esp+70h+var_5F]
mov     byte ptr [esp+70h+var_20+2], al
lea     eax, [esp+70h+ArgList]
push    eax             ; ArgList
push    offset aBitLyS  ; "bit.ly/%s\n"
call    fprintf
```

Back to Whoof!.exe main. As mentioned before, it loads the Java.dll, and runs the DLL's XorForever function.

Now, the main loads the old Java.dll. I want it to load the new DLL so I patched the file name to the decrypted one:

```
push    offset NewJava  ; "DecJava.dll"
call    ds:LoadLibraryA
mov     [ebp+hModule], eax
cmp     [ebp+hModule], 0
```

Now we can run main. It loads DecJava.dll and runs the XorForever function that requires a password.

Let's enter a password that applies to all conditions:

```
C:\Users\roeyg\OneDrive\מסמכים\לימודים\שנה ג\סמסטר א\Reversing\תרגיל מסכם\WRITEUP\Whoof!.exe
########%######++++#%%#########%############%%%###**####*########%%%##***##%%%####%
%%%%%####%%@%%%%%#%#############%%%%%%%%%%##%%%%%#######%#%#*####******##%%%%%#######

This is Python, he is sleeping.
His dream is to run free in the field, and let Java chase him.
Will you set them free?
Password:
zzzzzzzzyM0s
bit.ly/3TwB99f
```

The password passed the checking, the link has been built and printed!

The link leads to the end! BINGO!