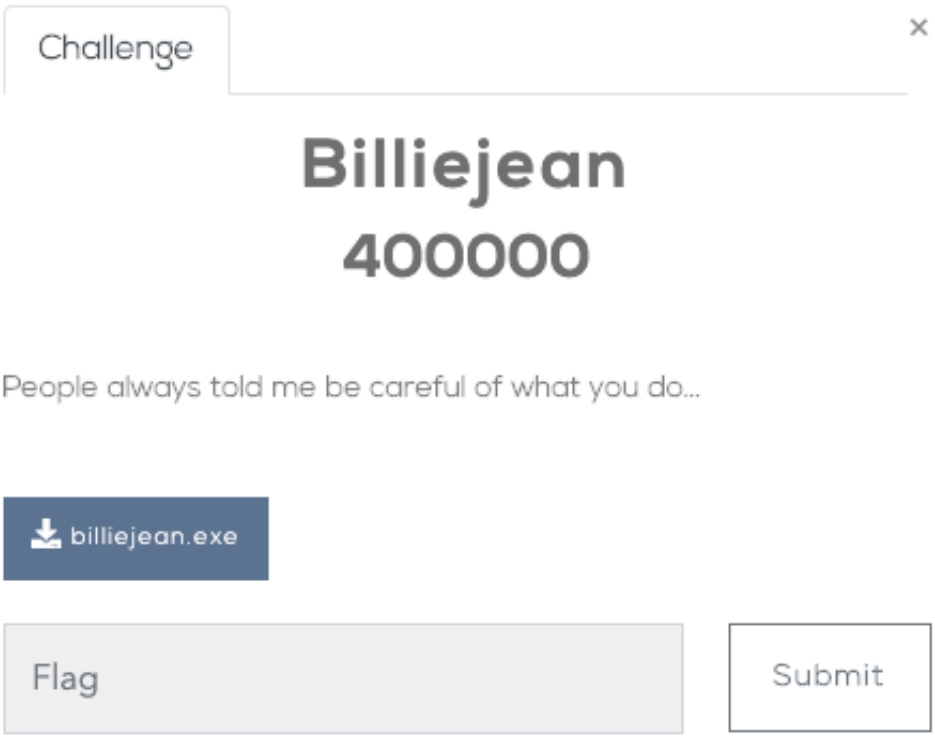
billiejean

Author: Shaked Reiner



`billiejean.exe` is a 32bit Windows binary.

```
$ file billiejean.exe
  billiejean.exe: PE32 executable (console) Intel 80386, for MS Windows
```

Let's execute it in a VM and see how it behaves.

```
PS C:\> .\billiejean.exe
Nope...
```

Disassembling the file, we'll try to understand why we get this output.

```
.text:004014D0                 cmp     [ebp+argc], 1
.text:004014D4                 jz      bad_input_exit
.
.
.
```

```
.text:00401654 bad_input_exit:                         ; CODE XREF:
_main+14↑j
.text:00401654                                         ; _main+3C↑j ...
.text:00401654                 push    offset aNope    ; "Nope...\n"
.text:00401659                 call    printf
.text:0040165E                 mov     ecx, [ebp+var_4]
.text:00401661                 add     esp, 4
.text:00401664                 xor     ecx, ebp
.text:00401666                 mov     eax, 1
.text:0040166B                 call    @__security_check_cookie@4 ;
.text:00401670                 mov     esp, ebp
.text:00401672                 pop     ebp
.text:00401673                 retn
.text:00401673 _main           endp
```

Right at the start of `main`, we can see the binary expects a command-line argument and exists otherwise.

Next, the code checks the length of the argument supplied to the program and exits if it's not `32` in length.

```
.text:004014DA                 mov     eax, [ebp+argv]
.text:004014DD                 mov     ecx, [eax+4]
.text:004014E0                 mov     cmdline_argument, ecx
.text:004014E6                 lea     edx, [ecx+1]
.text:004014E9                 nop     dword ptr [eax+00000000h]
.text:004014F0
.text:004014F0 bad_input_exit:                         ; CODE XREF:
_main+35↓j
.text:004014F0                 mov     al, [ecx]
.text:004014F2                 inc     ecx
.text:004014F3                 test    al, al
.text:004014F5                 jnz     short loc_4014F0
.text:004014F7                 sub     ecx, edx
.text:004014F9                 cmp     ecx, 32
.text:004014FC                 jnz     bad_input_exit
```

So, if we give the executable one argument of 32 characters, we'll get to the interesting part. First we have a loop from `00401512` to `004015DA`. `ebx` is our counter for this loop and the following code tells us it'll be executed `4` time.

```
.text:00401512 loc_401512:                             ; CODE XREF:
_main+11A↓j
.text:00401512                 push    0Ah             ; lpType
.text:00401514                 lea     eax, [ebx+123]
.text:00401517                 movzx   eax, ax
.text:0040151A                 push    eax             ; lpName
.text:0040151B                 push    0               ; hModule
```

```
.text:0040151D                  call    ds:FindResourceW
.text:00401523                  mov     esi, eax
.text:00401525                  push    esi             ; hResInfo
.text:00401526                  push    0               ; hModule
.text:00401528                  call    ds:LoadResource
.text:0040152E                  push    esi             ; hResInfo
.text:0040152F                  push    0               ; hModule
.text:00401531                  mov     edi, eax
.text:00401533                  call    ds:SizeofResource
.text:00401539                  push    edi             ; hResData
.text:0040153A                  mov     esi, eax
.text:0040153C                  call    ds:LockResource
.text:00401542                  mov     edi, eax
.text:00401544                  mov     [ebp+flOldProtect], 0
.text:0040154B                  lea     eax, [ebp+flOldProtect]
.text:0040154E                  push    eax             ; lpflOldProtect
.text:0040154F                  push    40h             ; flNewProtect
.text:00401551                  push    esi             ; dwSize
.text:00401552                  push    edi             ; lpAddress
.text:00401553                  call    ds:GetCurrentProcess
.text:00401559                  push    eax             ; hProcess
.text:0040155A                  call    ds:VirtualProtectEx
.text:00401560                  mov     eax, cmdline_argument
.text:00401565                  mov     dl, [eax+ebx]
.text:00401568                  movsx   eax, dl
.text:0040156B                  mov     [ebp+var_15], dl
.text:0040156E                  movd    xmm0, eax
.text:00401572                  xor     eax, eax
.text:00401574                  punpcklbw xmm0, xmm0
.text:00401578                  punpcklwd xmm0, xmm0
.text:0040157C                  pshufd  xmm2, xmm0, 0
.text:00401581                  test    esi, esi
.text:00401583                  jz      short loc_4015CB
.text:00401585                  cmp     esi, 20h
.text:00401588                  jb      short loc_4015BA
.text:0040158A                  mov     ecx, esi
.text:0040158C                  mov     edx, esi
.text:0040158E                  and     ecx, 1Fh
.text:00401591                  sub     edx, ecx
.text:00401593
.text:00401593 loc_401593:                              ; CODE XREF: _main+F5↓j
.text:00401593                  movups  xmm0, xmmword ptr [edi+eax]
.text:00401597                  movaps  xmm1, xmm2
.text:0040159A                  pxor    xmm1, xmm0
.text:0040159E                  movups  xmmword ptr [edi+eax], xmm1
.text:004015A2                  movups  xmm0, xmmword ptr [edi+eax+10h]
.text:004015A7                  pxor    xmm0, xmm2
.text:004015AB                  movups  xmmword ptr [edi+eax+10h], xmm0
.text:004015B0                  add     eax, 20h
```

```
.text:004015B3                 cmp     eax, edx
.text:004015B5                 jb      short loc_401593
.text:004015B7                 mov     dl, [ebp+var_15]
.text:004015BA
.text:004015BA loc_4015BA:                             ; CODE XREF: _main+C8↑j
.text:004015BA                 cmp     eax, esi
.text:004015BC                 jnb     short loc_4015CB
.text:004015BE                 xchg    ax, ax
.text:004015C0
.text:004015C0 loc_4015C0:                             ; CODE XREF:
_main+109↓j
.text:004015C0                 xor     [eax+edi], dl
.text:004015C3                 lea     ecx, [eax+edi]
.text:004015C6                 inc     eax
.text:004015C7                 cmp     eax, esi
.text:004015C9                 jb      short loc_4015C0
.text:004015CB
.text:004015CB loc_4015CB:                             ; CODE XREF: _main+C3↑j
.text:004015CB                                         ; _main+FC↑j
.text:004015CB                 push    ebx
.text:004015CC                 mov     edx, edi
.text:004015CE                 call    sub_401000
.text:004015D3                 inc     ebx
.text:004015D4                 add     esp, 4
.text:004015D7                 cmp     ebx, 4
.text:004015DA                 jl      loc_401512
```

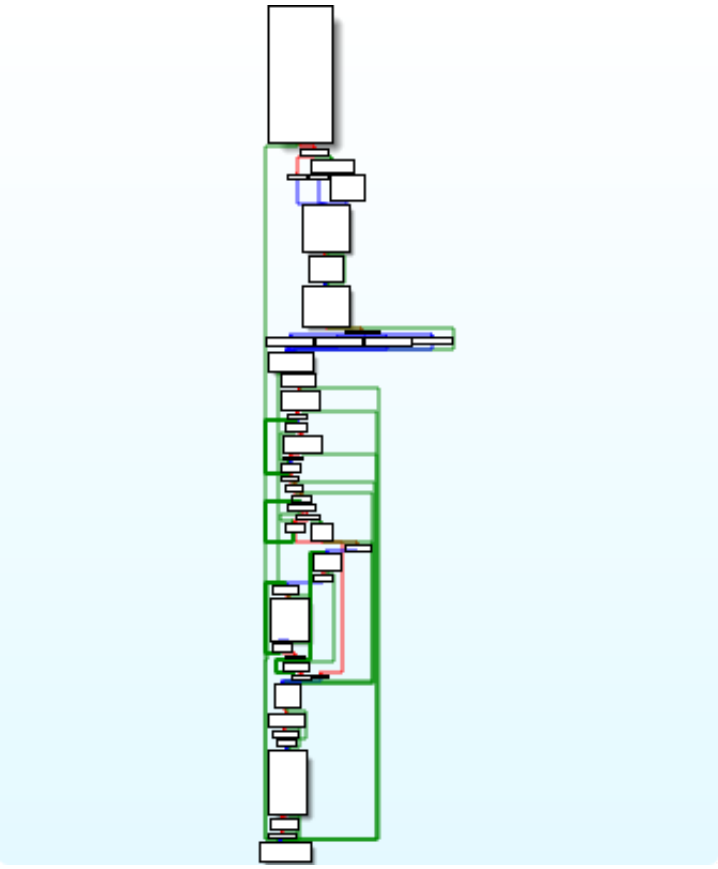
The code does a few things every iteration of the loop (total of `4`):

1. Finds, loads and gets a pointer to a resource using the appropriate WinAPI calls; `FindResourceW()`, `LoadResource()`, `SizeOfResource()` and `LockResourceW()` (staring at `0040151D`).
2. Changes resource's memory permissions to `PAGE_EXECUTE_READWRITE` using `VirtualProtectEx` (`0040155A`).
3. Some calculations and memory modification using the `xmm` registers.
4. Calls `sub_401000` with the loop counter as an argument.

Stages `1` and `2` are pretty self explenatory. Debugging stage `3` will show that this short loop simply XORs the loaded resource with a single byte from our input (first resouce will be xored with our input at offset 0, second with the input at offset 1 and so on). This is an optimized implementation of a simple xor encryption that matches the following C code:

```
void decrypt(BYTE * pStart, DWORD dwSize, CHAR cKey) {
  for (int i = 0; i < dwSize; i++) {
    pStart[i] ^= cKey;
  }
}
```

Stage `4` is a bit more complex, taking a look at `sub_401000` will show this overview:



We'll try to get the general idea of this quite long function by using 1 of two methods: either using the `Proximity View` in IDA, or by printing the function summary in `r2`:

```
$ r2 -A billiejean.exe
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
 .--.     .------------------.
 | _|_    |                  |
 | O O   <  Here be dragons. |
 |  ||    |                  |
 | _:|    `------------------'
 |   |
 `---'
[0x00401969]> s 0x401000
[0x00401000]> pdsf~sym.imp
0x00401059 call dword [sym.imp.KERNEL32.dll_CreateProcessA]
0x00401071 call dword [sym.imp.KERNEL32.dll_LoadLibraryA]
0x00401085 call dword [sym.imp.KERNEL32.dll_GetProcAddress]
0x004010ec call dword [sym.imp.KERNEL32.dll_ReadProcessMemory]
0x0040111a call dword [sym.imp.KERNEL32.dll_ReadProcessMemory]
```

```
0x004011d6 call dword [sym.imp.KERNEL32.dll_GetModuleHandleA]
0x004011e2 call dword [sym.imp.KERNEL32.dll_GetProcAddress]
0x00401209 call dword [sym.imp.KERNEL32.dll_VirtualAllocEx]
0x00401232 call dword [sym.imp.KERNEL32.dll_WriteProcessMemory]
0x00401266 call dword [sym.imp.KERNEL32.dll_WriteProcessMemory]
0x0040133e call dword [sym.imp.KERNEL32.dll_ReadProcessMemory]
0x00401360 call dword [sym.imp.KERNEL32.dll_WriteProcessMemory]
0x004013a7 call dword [sym.imp.api_ms_win_crt_heap_l1_1_0.dll_malloc]
0x004013c9 call dword [sym.imp.KERNEL32.dll_ReadProcessMemory]
0x004013d4 call dword [sym.imp.api_ms_win_crt_heap_l1_1_0.dll_free]
0x004013e6 call dword [sym.imp.api_ms_win_crt_heap_l1_1_0.dll_malloc]
0x00401408 call dword [sym.imp.api_ms_win_crt_string_l1_1_0.dll_strncpy]
0x00401434 call dword [sym.imp.KERNEL32.dll_WriteProcessMemory]
0x0040146b call dword [sym.imp.KERNEL32.dll_GetThreadContext]
0x00401482 call dword [sym.imp.KERNEL32.dll_SetThreadContext]
0x0040148f call dword [sym.imp.KERNEL32.dll_ResumeThread]
```

The command `pdsf` will **P**rint the **D**isassembly **S**ummary of the **F**unction and `~` will filter for only results containing `sym.imp` in order to get WinAPI imported funtcions.

If you are farmiliar with the *Process Hollowing* or *RunPE* technique, you may recognize this sequence of API calls. Simply put, the binary will create a new benign process, then inject the code of the resouce to its memory and execute. In this case, the binary will create the process `charmap`.

```
|   0x00401041      53              push ebx        ; LPPROCESS_INFORMATION
pProcessInformation
|   0x00401042      56              push esi        ; LPSTARTUPINFOA
lpStartupInfo
|   0x00401043      6a00            push 0          ; LPCSTR lpCurrentDirectory
|   0x00401045      6a00            push 0          ; LPVOID lpEnvironment
|   0x00401047      6a04            push 4          ; 4 ; DWORD dwCreationFlags
|   0x00401049      6a00            push 0          ; BOOL bInheritHandles
|   0x0040104b      6a00            push 0          ; LPSECURITY_ATTRIBUTES
lpThreadAttributes
|   0x0040104d      6a00            push 0          ; LPSECURITY_ATTRIBUTES
lpProcessAttributes
|   0x0040104f      6830324000      push str.charmap; 0x403230 ; "charmap" ;
LPSTR pCommandLine
|   0x00401054      6a00            push 0          ; LPCSTR lpApplicationName
|   0x00401056      0f1103          movups xmmword [ebx], xmm0
|   0x00401059      ff150c304000    call dword
[sym.imp.KERNEL32.dll_CreateProcessA]
```

Let's start by examining the resources of the binray. The following `r2` command will print a short hexdump of every resource of the binary:

```
[0x00401016]> px @@= `iR~vaddr[1]`
```

```
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x00406130  2532 f868 6b68 6868 6c68 6868 9797  %2.hkhhhlhhh..
0x0040613e  6868 d068 6868 6868 6868 2868 6868  hh.hhhhhhh(hhh
0x0040614c  6868 6868 6868 6868 6868 6868 6868  hhhhhhhhhhhhhh
0x0040615a  6868 6868 6868 6868 6868 6868 6868  hhhhhhhhhhhhhh
0x00406168  6868 6868 9068 6868 6677 d266 68dc  hhhh.hhhfw.fh.

- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x00417d30  7d6a a030 3330 3030 3430 3030 cfcf  }j.030004000..
0x00417d3e  3030 8830 3030 3030 3030 7030 3030  00.0000000p000
0x00417d4c  3030 3030 3030 3030 3030 3030 3030  00000000000000
0x00417d5a  3030 3030 3030 3030 3030 3030 3030  00000000000000
0x00417d68  3030 3030 c830 3030 3e2f 8a3e 3084  0000.000>/.>0.

- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x00429930  2136 fc6c 6f6c 6c6c 686c 6c6c 9393  !6.lollllhlll..
0x0042993e  6c6c d46c 6c6c 6c6c 6c6c 2c6c 6c6c  ll.lllllll,lll
0x0042994c  6c6c 6c6c 6c6c 6c6c 6c6c 6c6c 6c6c  llllllllllllll
0x0042995a  6c6c 6c6c 6c6c 6c6c 6c6c 6c6c 6c6c  llllllllllllll
0x00429968  6c6c 6c6c 946c 6c6c 6273 d662 6cd8  llll.lllbs.bl.

- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x0043b530  2136 fc6c 6f6c 6c6c 686c 6c6c 9393  !6.lollllhlll..
0x0043b53e  6c6c d46c 6c6c 6c6c 6c6c 2c6c 6c6c  ll.lllllll,lll
0x0043b54c  6c6c 6c6c 6c6c 6c6c 6c6c 6c6c 6c6c  llllllllllllll
0x0043b55a  6c6c 6c6c 6c6c 6c6c 6c6c 6c6c 6c6c  llllllllllllll
0x0043b568  6c6c 6c6c 946c 6c6c 6273 d662 6cd8  llll.lllbs.bl.

- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x0044d130  3c3f 786d 6c20 7665 7273 696f 6e3d  <?xml version=
0x0044d13e  2731 2e30 2720 656e 636f 6469 6e67  '1.0' encoding
0x0044d14c  3d27 5554 462d 3827 2073 7461 6e64  ='UTF-8' stand
0x0044d15a  616c 6f6e 653d 2779 6573 273f 3e0d  alone='yes'?>.
0x0044d168  0a3c 6173 7365 6d62 6c79 2078 6d6c  .<assembly xml
```

We know based on the aforementioned loop that we should have 4 resources. the last on we got is just the file's manifest so it's not relevant. Knowing what we know on the main loop of the code that performs *process hollowing* and writes the executable from every resource into a benign process, we can assume that the resources should be valid *PE* files (even tough this is not mandatory). So we need the first two bytes of every resource to be `MZ` (since we only have a one byte key, the `M` will suffice). To get the first 4 characters, we'll first get the four addresses of the resources.

```
[0x00406130]> iR~vaddr[1]
0x00406130
0x00417d30
0x00429930
0x0043b530
```

For each address we'll xor the first byte with `M`.

```
[0x00401969]> ? 0x`p8 1 @ 0x00406130` ^ 'M' ~string
string  "h"
[0x00406130]> ? 0x`p8 1 @ 0x00417d30` ^ 'M' ~string
string  "O"
[0x00406130]> ? 0x`p8 1 @ 0x00429930` ^ 'M' ~string
string  "l"
[0x00406130]> ? 0x`p8 1 @ 0x0043b530` ^ 'M' ~string
string  "l"]
```

We got the first four characters of the flag `hOll`.

Now let's decode and dump the resources using the following steps in r2 for each resource.

```
[0x00401969]> iR                                        # Get the size of
resource
Resource 0
  name: 123
  timestamp: Mon Dec 31 22:00:00 1979
  vaddr: 0x00406130
  size: 71K
  type: RCDATA
  language: LANG_ENGLIS
.
.
```

XOR the resource with the correct XOR key using the **W**rite **O**peration **X**or command. Our key will be `68` (hex representation of ASCII `h`).

```
[0x00401969]> wox 68 @ 0x00406130!71K                   # xor <hex-key> @
<addr>!<size>
```

Dump the result to a file using the **W**rite **T**o **F**ile command.

```
[0x00401969]> wtf res1.exe 71K @ 0x00406130
```

Let's continue to scroll through `sub_401000`'s code. It looks like a classic *process hollwing* up until the jumptable at `0x001711AA`.

```
.text:004011A5                 cmp     ecx, 3          ; switch 4 cases
.text:004011A8                 ja      short loc_4011D1 ; jumptable 004011AA
default case
.text:004011AA                 jmp     ds:off_4014A8[ecx*4] ; switch jump
.text:004011B1 ; ---------------------------------------------------------------
--------------
.text:004011B1
```

```
.text:004011B1 loc_4011B1:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011B1                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011B1                 movsx   eax, al         ; jumptable 004011AA
case 0
.text:004011B4                 xor     [esi+80h], eax
.text:004011BA                 jmp     short loc_4011D1 ; jumptable 004011AA
default case
.text:004011BC ; ---------------------------------------------------------------
--------------
.text:004011BC
.text:004011BC loc_4011BC:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011BC                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011BC                 cbw                     ; jumptable 004011AA
case 1
.text:004011BE                 xor     [esi+14h], ax
.text:004011C2                 jmp     short loc_4011D1 ; jumptable 004011AA
default case
.text:004011C4 ; ---------------------------------------------------------------
--------------
.text:004011C4
.text:004011C4 loc_4011C4:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011C4                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011C4                 cbw                     ; jumptable 004011AA
case 2
.text:004011C6                 xor     [esi+6], ax
.text:004011CA                 jmp     short loc_4011D1 ; jumptable 004011AA
default case
.text:004011CC ; ---------------------------------------------------------------
--------------
.text:004011CC
.text:004011CC loc_4011CC:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011CC                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011CC                 movsx   eax, al         ; jumptable 004011AA
case 3
.text:004011CF                 xor     [esi], eax
```

Backtracking to find the what `ecx` contains, we see that it's actually the argument to the function, which we know to be the loop counter.

```
.text:00401010                mov     eax, [ebp+arg_0]
.
.
.text:0040101B                mov     [ebp+var_48], eax
.
.
.text:00401181                mov     ecx, [ebp+var_48]
.
.
.text:004011A5                cmp     ecx, 3          ; switch 4 cases
.text:004011A8                ja      short loc_4011D1 ; jumptable 004011AA
default case
.text:004011AA                jmp     ds:off_4014A8[ecx*4] ; switch jump
```

For each of the 4 processes created, the binary will behave differently. We'll analyze it one by one starting with `ecx == 0` (case 0).

```
.text:004011B1 ; ---------------------------------------------------------------
--------------
.text:004011B1
.text:004011B1 loc_4011B1:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011B1                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011B1                movsx   eax, al         ; jumptable 004011AA
case 0
.text:004011B4                xor     [esi+80h], eax
.text:004011BA                jmp     short loc_4011D1 ; jumptable 004011AA
default case
.text:004011BC ; ---------------------------------------------------------------
--------------
```

We need to know what do `eax` and `esi` contain, so let's backtrack again.

```
.text:00401194                mov     eax, cmdline_argument
.
.
.text:004011A1                mov     al, [ecx+eax+4]
```

`eax` will containg the next character in every iteration of the loop (4th char, 5th char etc).

```
.text:0040153C                call    ds:LockResource             ; main
.text:00401542                mov     edi, eax
.
.
.text:004015CC                mov     edx, edi                    ; main
.text:004015CE                call    sub_401000
```

```
.
.
.text:00401018                 mov     [ebp+lpBuffer], edx          ;
sub_401000
.
.
.text:0040116F                 mov     edx, [ebp+lpBuffer]          ;
sub_401000
.text:00401172                 add     esp, 10h
.text:00401175                 mov     ecx, [edx+3Ch]
.text:00401178                 add     ecx, edx
.text:0040117A                 mov     [esi+0Ch], ecx
.text:0040117D                 movzx   eax, word ptr [ecx+6]
.text:00401181                 mov     ecx, [ebp+var_48]
.text:00401184                 mov     [esi+14h], eax
.text:00401187                 mov     eax, [edx+3Ch]
.text:0040118A                 add     eax, 0F8h
.text:0040118F                 add     eax, edx
.text:00401191                 mov     [esi+18h], eax
.text:00401194                 mov     eax, cmdline_argument
.text:00401199                 mov     esi, [edx+3Ch]
.text:0040119C                 add     esi, edx
```

Going from `main` `edx` will contain a pointer to our decoded resouce. In `sub_401000`, this pointer will be stored in `ebp+lpBuffer` and treated as a PE header. Finally, `esi` will contain a pointer to the *PE header* (taken from `edx+0x3C`). Now we can continue to analyze the switch statement.

```
.text:004011B1 ; ---------------------------------------------------------------
--------------
.text:004011B1
.text:004011B1 loc_4011B1:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011B1                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011B1                 movsx   eax, al         ; jumptable 004011AA
case 0
.text:004011B4                 xor     [esi+80h], eax
.text:004011BA                 jmp     short loc_4011D1 ; jumptable 004011AA
default case
.text:004011BC ; ---------------------------------------------------------------
--------------
```

During the **first** loop iteration, the binary will use the 1st decoded resource header (`esi`) and the 5th character (`eax`). At offset `0x80` of the *pe header* (a convenient reference here) we can find the `ImportDirectory Virtual Address`. That means we need the right value in the flag that will fix the `ImportDirectory VA` for the file to run properly. First, check the current value.

```
$ r2 res1.exe
 -- Execute commands on a temporary offset by appending '@ offset' to your
command.
[0x00401479]> ih
...
IMAGE_DIRECTORY_ENTRY_IMPORT
  VirtualAddress : 0x110b4
```

We'll look in the binary the find the correct virtual address for the import table which will be `0x11084`. XORing the two values, we'll get the correct character value - `0`.

```
[0x00401479]> ? 0x110b4 ^ 0x11084~string
string  "0"
```

```
.text:004011BC ; ---------------------------------------------------------------
--------------
.text:004011BC
.text:004011BC loc_4011BC:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011BC                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011BC                 cbw                     ; jumptable 004011AA
case 1
.text:004011BE                 xor     [esi+14h], ax
.text:004011C2                 jmp     short loc_4011D1 ; jumptable 004011AA
default case
.text:004011C4 ; ---------------------------------------------------------------
--------------
```

In the **second** case, `eax` will be pointing to the 6th character of the flag and `[esi+14h]` will point to the *PE header* at the `SizeOfOptionalHeader` field.

```
$ r2 res2.exe
 -- In soviet russia, radare2 debugs you!
[0x00401469]> ih~SizeOfOptionalHeader
  SizeOfOptionalHeader : 0x97
```

Checking the correct size manually or using *010 Editor* we'll get the following:

Name	Value	Start	Size
▶ struct IMAGE_OPTIONAL_HEADER32 OptionalHeader		110h	E0h

So to get the correct value:

```
[0x00401469]> ? 0xE0 ^ 0x97 ~string
string  "w"
```

Moving forward to the **third** case:

```
.text:004011C4 ; ---------------------------------------------------------------------------
.text:004011C4
.text:004011C4 loc_4011C4:                             ; CODE XREF: sub_401000+1AA↑j
.text:004011C4                                         ; DATA XREF: .text:off_4014A8↓o
.text:004011C4                 cbw                     ; jumptable 004011AA case 2
.text:004011C6                 xor     [esi+6], ax
.text:004011CA                 jmp     short loc_4011D1 ; jumptable 004011AA default case
.text:004011CC ; ---------------------------------------------------------------------------
```

`eax` will contain the 7th character and `[esi+6]` will point to the `NumberOfSections` field of the *PE header*.

```
$ r2 res3.exe
 -- Find wide-char strings with the '/w <string>' command
[0x00401469]> ih~NumberOfSections
  NumberOfSections : 0x37
```

In order to get the real number of sections by using the **i**nfo **S**ections command:

```
[0x00401469]> iS
[Sections]
Nm Paddr       Size Vaddr      Memsz Perms Name
00 0x00000400 42496 0x00401000 45056 -r-x .text
01 0x0000aa00 22528 0x0040c000 24576 -r-- .rdata
02 0x00010200  2048 0x00412000  8192 -rw- .data
03 0x00010a00   512 0x00414000  4096 -r-- .gfids
04 0x00010c00   512 0x00415000  4096 -r-- .rsrc
05 0x00010e00  3584 0x00416000  4096 -r-- .reloc
06 0x08458b00   252 0x460720ec 2213318656 srwx sect_6
07 0x07eb0000  4096 0xc749741f 3263369216 ---- 3343479839
08 0x0001f045  4096 0xf9c326f1 2200240128 sr-- sect_8
09 0x00000001  4096 0x7578f883 62951424 -rwx 1970862211
10 0x00000001  4096 0x75400003 3694997504 -rwx 1967128579
.
.
.
```

`r2` tries to list all `0x37` sections as stated in the header, but it is obvious that there are only `0x6` sections that look correct. This is our next character.

```
[0x00401469]> ? 0x37 ^ 6 ~string
string  "1"
```

The **fourth** and final case:

```
.text:004011CC ; ---------------------------------------------------------------
--------------
.text:004011CC
.text:004011CC loc_4011CC:                             ; CODE XREF:
sub_401000+1AA↑j
.text:004011CC                                         ; DATA XREF:
.text:off_4014A8↓o
.text:004011CC                 movsx   eax, al         ; jumptable 004011AA
case 3
.text:004011CF                 xor     [esi], eax
```

Here, the 8 character is XORed with `[esi]` that points to the start of the *PE header* that should contain its signature: `0x4550`.

```
$ r2 -nn res4.exe                                       # load only binary
structs
Struct or fields name can not contain dot symbol (.)
 -- Bindings are mostly powered by tears.
[0x00000000]> pfo pe32
[0x00000000]>  pf.pe_nt_image_headers32 @ `pfq.dos_header.e_lfanew`
Index out of bounds
      signature : 0x000000f8 = ">E"
      .
      .
[0x00000000]> ?x \>
3e
```

```
[0x00000000]> ? 0x3E ^ 0x50 ~string
string  "n"
```

The rest of `sub_401000` seem to be standard *process hollowing* code, except for one block.

```
.text:004013E4 loc_4013E4:                             ; CODE XREF:
sub_401000+3D1↑j
.text:004013E4                 push    6               ; Size
.text:004013E6                 call    ds:__imp_malloc
.text:004013EC                 mov     ecx, cmdline_argument
.text:004013F2                 mov     esi, eax
.text:004013F4                 mov     eax, [ebp+var_48]
.text:004013F7                 push    6               ; Count
.text:004013F9                 mov     byte ptr [esi+6], 0
```

```
.text:004013FD                  lea     edx, [eax+eax*2]
.text:00401400                  lea     ecx, [ecx+edx*2]
.text:00401403                  add     ecx, 8
.text:00401406                  push    ecx             ; Source
.text:00401407                  push    esi             ; Dest
.text:00401408                  call    ds:strncpy
.text:0040140E                  push    esi
.text:0040140F                  push    offset aCharmapHs ; "charmap %hs"
.text:00401414                  lea     eax, [ebp+Buffer]
.text:00401417                  push    0Fh
.text:00401419                  push    eax
.text:0040141A                  call    w_swprintf
.text:0040141F                  add     esp, 20h
.text:00401422                  lea     eax, [ebp+NumberOfBytesWritten]
.text:00401425                  push    eax             ;
lpNumberOfBytesWritten
.text:00401426                  push    1Eh             ; nSize
.text:00401428                  lea     eax, [ebp+Buffer]
.text:0040142B                  push    eax             ; lpBuffer
.text:0040142C                  mov     eax, [ebp+hProcess]
.text:0040142F                  push    dword ptr [eax+44h] ; lpBaseAddress
.text:00401432                  push    dword ptr [ebx] ; hProcess
.text:00401434                  call    ds:WriteProcessMemory
.text:0040143A                  push    2CCh            ; Size
.text:0040143F                  mov     byte ptr [edi+2], 1
.text:00401443                  mov     dword ptr [edi+68h], 70h
.text:0040144A                  call    ???2@YAPAXI@Z   ; operator new(uint)
.text:0040144F                  push    2CCh            ; Size
.text:00401454                  mov     esi, eax
.text:00401456                  push    0               ; Val
.text:00401458                  push    esi             ; Dst
.text:00401459                  call    memset
.text:0040145E                  add     esp, 10h
.text:00401461                  mov     dword ptr [esi], 10002h
.text:00401467                  push    esi             ; lpContext
.text:00401468                  push    dword ptr [ebx+4] ; hThread
.text:0040146B                  call    ds:GetThreadContext
.text:00401471                  test    eax, eax
.text:00401473                  jz      short loc_401495
```

Here the binary allocates memory for 6 characters (for each process hollowed), formats those characters to `"charmap %hs"` and then writes it to the newly created process into `pPEB->ProcessParameters->CommandLine.Buffer`. This is done before the new process is resumed and thus changes the commandline arguments supplied to it.

So until now we have 8 characters of the flag: 4 that were used to decode the resources and 4 that were used to fix the header - `hOllOw1n`. 24 more characters to go.

```
.text:004015E2                  call    ds:Sleep
```

```
.text:004015E8                 mov     ebx, ds:GlobalDeleteAtom
.text:004015EE                 mov     edi, 1
.text:004015F3                 mov     [ebp+lpString], offset atom_names_array
; "V_I"
.text:004015FA                 xor     esi, esi
.text:004015FC                 mov     [ebp+var_10], (offset
atom_names_array+4) ; "V_II"
.text:00401603                 mov     [ebp+var_C], (offset
atom_names_array+0Ch) ; "V_III"
.text:0040160A                 mov     [ebp+var_8], (offset
atom_names_array+14h) ; "V_IV"
.text:00401611
.text:00401611 loc_401611:                             ; CODE XREF:
_main+16E↓j
.text:00401611                 push    [ebp+esi*4+lpString] ; lpString
.text:00401615                 call    ds:GlobalFindAtomA
.text:0040161B                 movzx   eax, ax
.text:0040161E                 test    ax, ax
.text:00401621                 jz      short loc_401628
.text:00401623                 push    eax             ; nAtom
.text:00401624                 call    ebx ; GlobalDeleteAtom
.text:00401626                 jmp     short loc_40162A
```

Here, the binary treis to delete the following global atoms: `V_I`, `V_II`, `V_III` and `V_IV`. If it failed to find one of them, it will fail and exit, but if the 4 were present, the binary will print the success message.

```
.text:00401626                 jmp     short loc_40162A
.text:00401628 ; ---------------------------------------------------------------
--------------
.text:00401628
.text:00401628 loc_401628:                             ; CODE XREF:
_main+161↑j
.text:00401628                 xor     edi, edi
.text:0040162A
.text:0040162A loc_40162A:                             ; CODE XREF:
_main+166↑j
.text:0040162A                 inc     esi
.text:0040162B                 cmp     esi, 4
.text:0040162E                 jb      short loc_401611
.text:00401630                 test    edi, edi
.text:00401632                 pop     edi
.text:00401633                 pop     esi
.text:00401634                 pop     ebx
.text:00401635                 jz      short bad_input_exit
.text:00401637                 push    offset aYouAreTheOneGo ; "You are the
one!\nGood job."
.text:0040163C                 call    sub_401680
.text:00401641                 add     esp, 4
```

```
.text:00401644                 xor     eax, eax
.text:00401646                 mov     ecx, [ebp+var_4]
.text:00401649                 xor     ecx, ebp
.text:0040164B                 call    @__security_check_cookie@4 ;
.text:00401650                 mov     esp, ebp
.text:00401652                 pop     ebp
.text:00401653                 retn
```

Look like there's nothing we missed in this binary, so let's start analyzing the 4 new process it creates.

Analyzing `res1.exe`, it simply gets 1 command line argument, verifies every character in a loop and if all characters are correct it then adds a global atom with a name `V_I`. Let's have a look on what the verification function does (decompiled code from *HexRays Decompiler* for convenience).

```
BOOL __cdecl sub_401000(int a1, int current_char)
{
  BOOL v3; // [esp+1Ch] [ebp-4h]

  v3 = 0;
  switch ( a1 )
  {
    case 0:
      v3 = current_char != -44;
      break;
    case 1:
      v3 = (current_char ^ 0x65) == 58;
      break;
    case 2:
      v3 = current_char == 121;
      break;
    case 3:
      v3 = 49 * (current_char ^ 0x54) == 4900;
      break;
    case 4:
      v3 = current_char == 117;
      break;
    case 5:
      v3 = 49 * (current_char ^ 0x54) == 2842;
      break;
    default:
      return v3;
  }
  return v3;
}
```

For every character of the 6 characters supplied to the binary by `billiejean.exe` there's one condition in the swtich statement. Based on those conditions, here are our 6 characters verified by the first decoded resource (`res1.exe`) - `?_yOun` where `?` can be any character that isn't equal to `-44`.

Opening the second binary in a disassembler we can see that only the constants in the verification function are different, so in a similar fashion, we will calculate the rest of our characters and get: `hOllOw1n?_yOun?_prOc?5535_?34r75`.

```
$ .\billiejean.exe hOllOw1n?_yOun?_prOc?5535_?34r75
You are the one!
Good job.
```