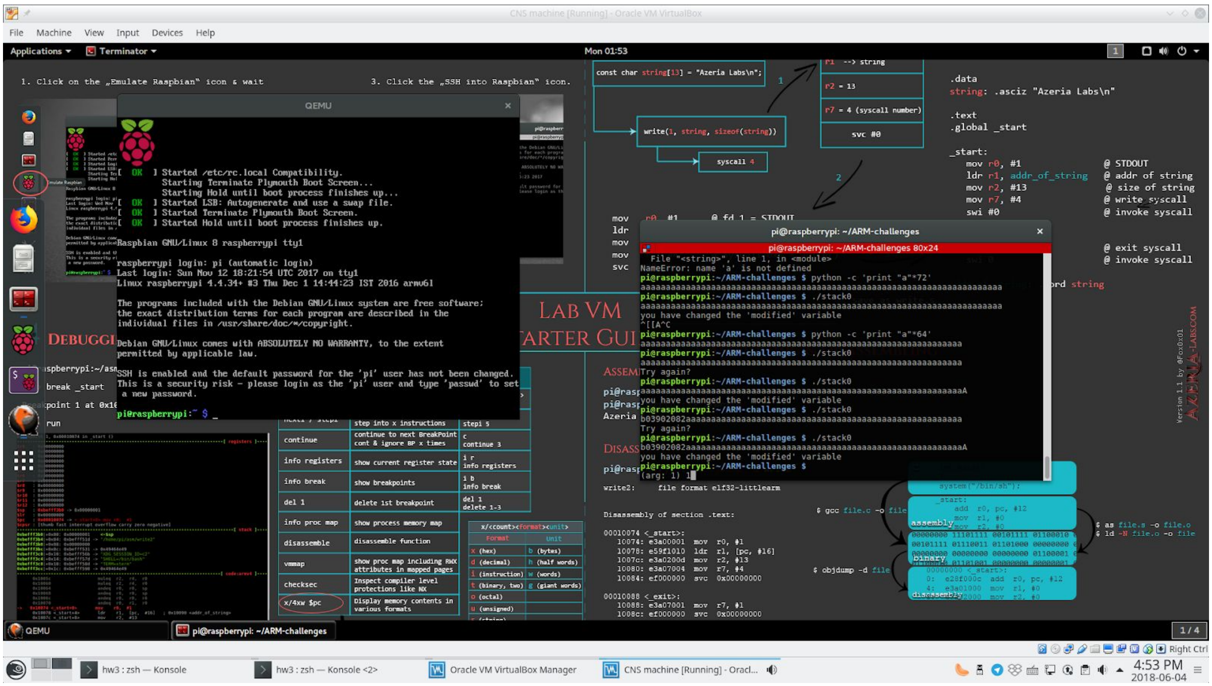
CNS HW3 Report

b03902082 江懿友

Problem 1

(a) stack0

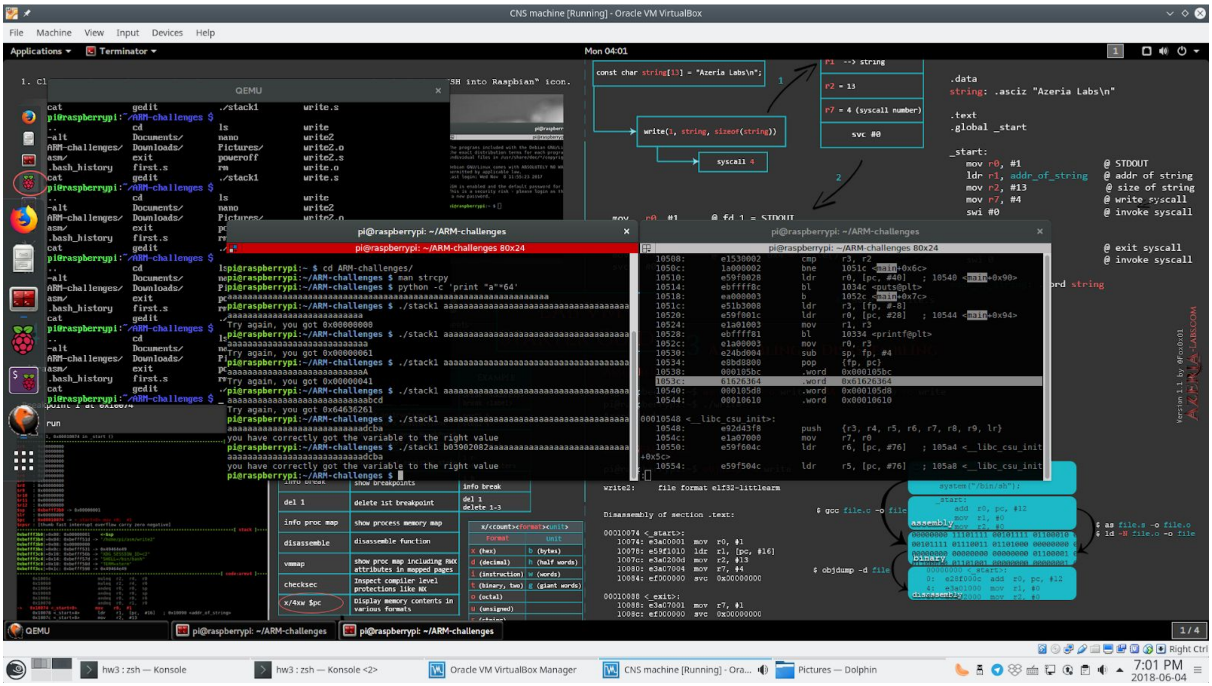
The buffer length is 64, so at least *65* characters should be input to trigger a buffer overflow.



(b) stack1

The value is 0x61626364, which is "dcba" in little-endian.

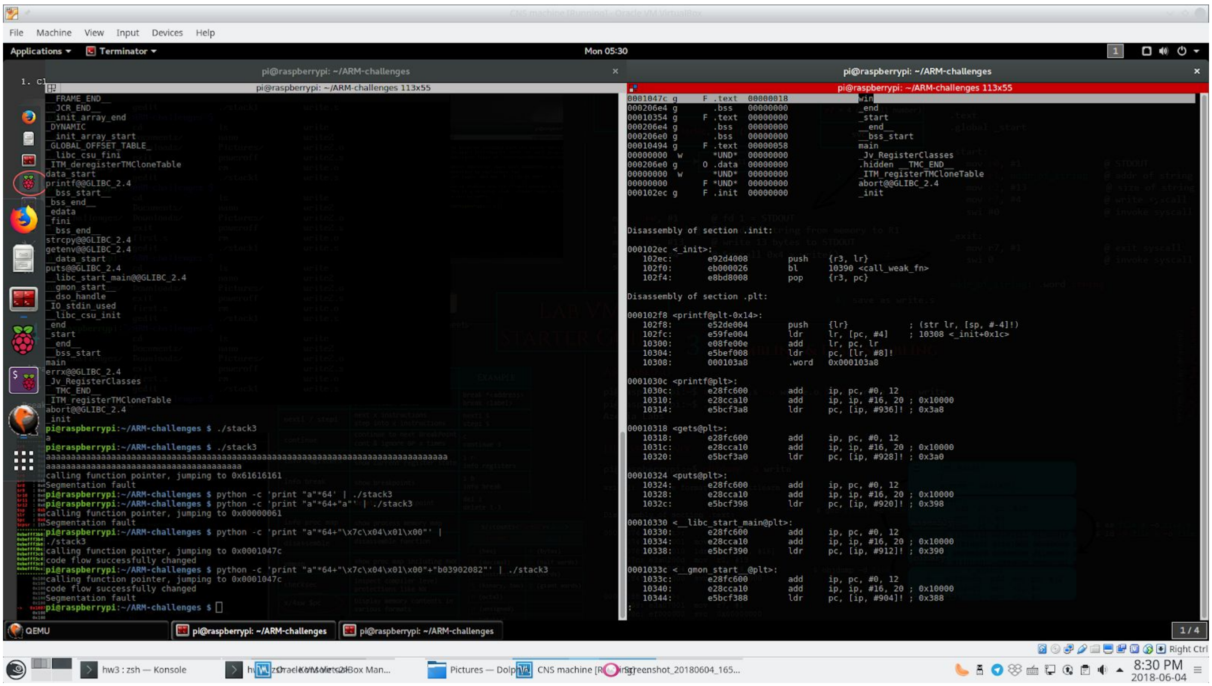
I use objdump to disassemble the program and found the value inside function <main>. The value is a pc-relative data located at 0x1053c.



(c) stack3

The address of <win> is 0x1047c.

The buffer length is 64, so I input 64 * 'a' + 0x1047c (encoded in little-endian) to overflow the buffer and overwrite the function pointer with the address of <win>.

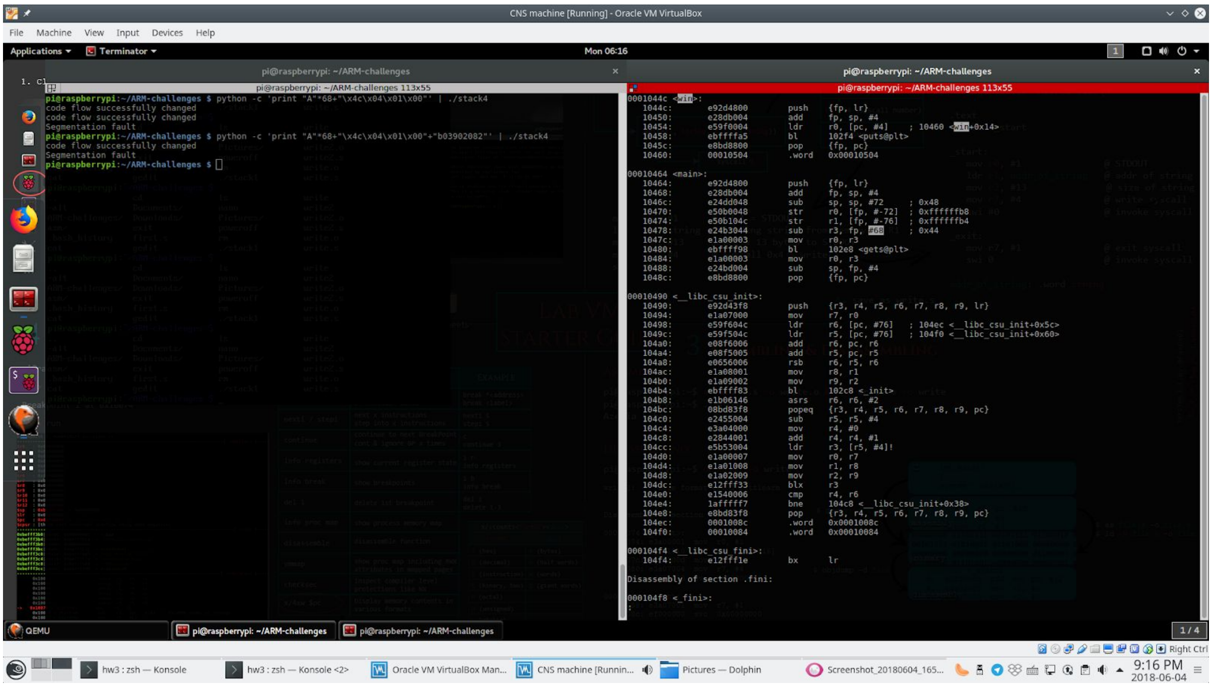


(d) stack4

The address of <win> is 0x1044c.

The buffer length is 64, so I input 68 * 'a' + 0x1044c (encoded in little-endian) to overflow the buffer, overwrite old_fp and overwrite return address with the address of

<win>. This way when <main> returns the control flow would be redirected to <win>.



Problem 2

(a) aMAZEing

Solution of the maze: ddddssaaaassssdddddwddwwaawwwddddsss

Flag: BALSN{4M4z31nG_bUg_F0uNd_bY_KLEE!!}

"Bugs" of check_wall(): (y, x) = {(7, 4), (6, 8), (4, 8), (1, 10), (2, 11)}

(b) Flag Verifier

Flag: BALSN{5ymb0l1c_Ex3cut10n_Hurr4y}

I changed this:

```c
int cal(int a, int b, int c)
{
    int i, j, k;
    int total = 0;

    for(i = 0; i < a; i++)
    {
        for(j = 0; j < b; j++)
        {
            for(k = 0; k < c; k++)
            {
                if(i > k && j < i)
                    return cal(a-1, b-1, c-1);
                return total + cal(a-1, b-1, c-1);
            }
        }
    }

    return 1;
}
```

to this:

```c
#define cal(a, b, c) (1)
```

Problem 3

(a) Mirror Force

Use a DNS query amplifier. This DNS query works pretty well:

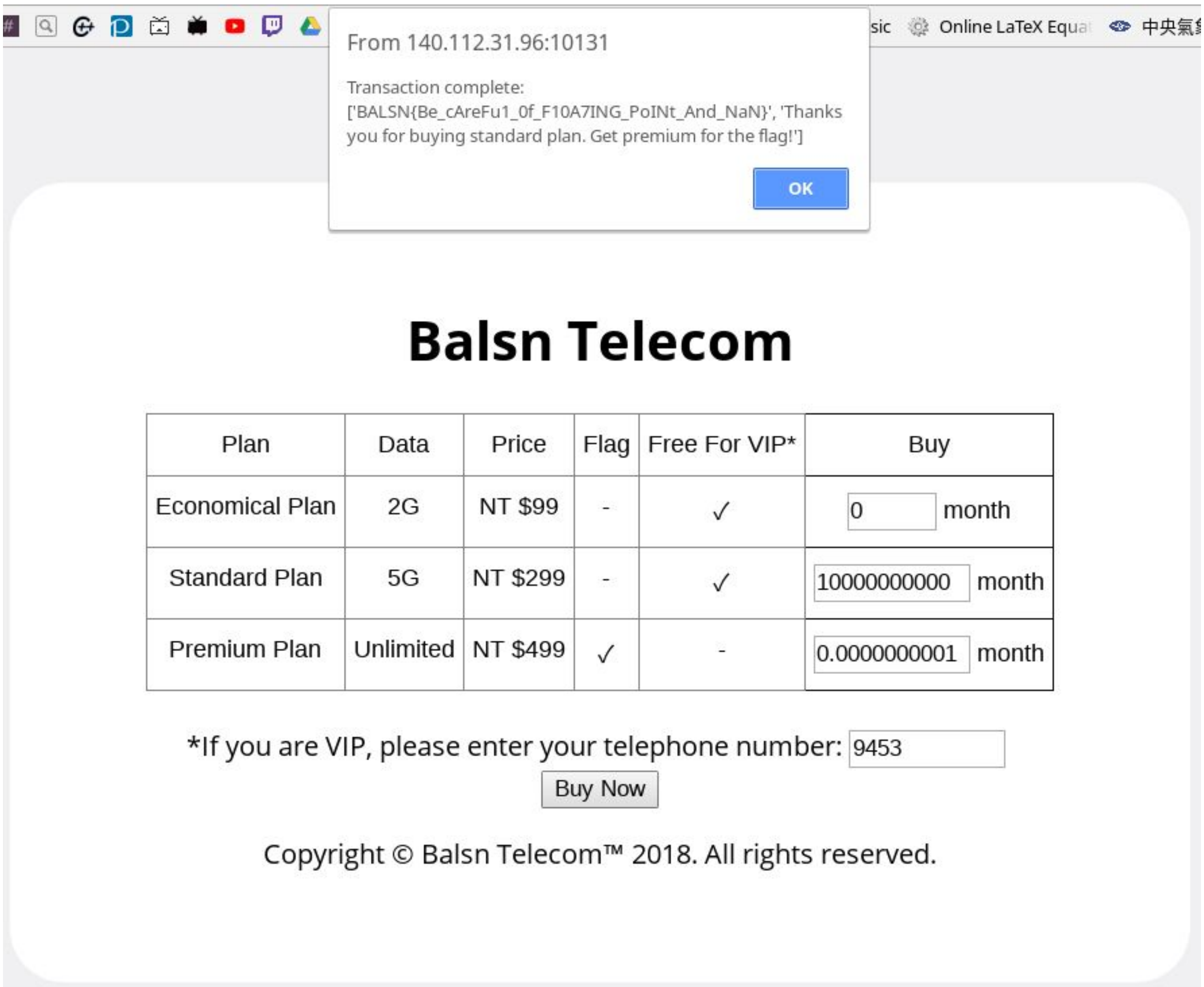
```
dig @8.8.8.8 RRSIG isc.org +notcp
```

I use wireshark to analyze the packets:

No.	Time	Source	Destination	Protoco	Length	Info
6	2.046075759	192.168.1.103	8.8.8.8	DNS	90	Standard query 0xa308 RRSIG isc.org OPT
8	2.191665126	8.8.8.8	192.168.1.103	IPv4	1482	Fragmented IP protocol (proto=UDP 17, off=0, ID=ea38) [Reassembled in #9]
9	2.191867135	8.8.8.8	192.168.1.103	DNS	762	Standard query response 0xa308 RRSIG isc.org RRSIG RRSIG RRSIG RRSIG RRSIG

I send an UDP packet of size 90 and google's public DNS server replies with an UDP packet of size 2244, so the amplification factor is 2244/90 = 24.93.
Because this DNS query uses UDP to transfer query/response, there is no TCP handshake and I can spoof the query packet's source IP to victim's IP, and thus the amplified response packet would be sent to the victim.

(b) The Revenge of Hash Table
(c) 499 Chaos
Flag: BALSN{Be_cAreFu1_0f_F10A7ING_PoINt_And_NaN}



The vulnerability is caused by a floating point rounding error. I guess that the python server uses IEEE-754 formatted double-precision floating point. IEEE-754 formatted double-precision floating point stores up to roughly 15 digits of precision (in base 10), so buying 1e10 months of "standard plan" and 1e-10 months of "premium plan" costs 1e10 * 299 + 1e-10 * 499 which rounds to 2.99e12. The cost of "premium plan" is loss and "standard plan" costs no money for VIP. Thus the total cost equals zero.

Problem 4

Challenge 0
Pass in my student ID and it is solved.

Challenge 1
The solution is 777. The answer can be found in the smart contract source.

Challenge 2

Send 1 eth and get score.

Challenge 3
Brute force search the range of solution (0~255) and the answer can be found.
keccak256(146) =
0x313b2ea16b36f2e78c1275bfcca4e31f1e51c3a5d60beeefe6f4ec441e6f1dfc

Challenge 4
The solution of challenge 4 equals uint16(keccak256(blockhash(block.number-1), block.timestamp)).
The block number of the contract is 3245076.
The blockhash of block 3245075 is
0xb388f89fb847890e2d96c9a37c9d25beadef55fb84c2b1b2cf41dc7703bd08f5.
The timestamp of block 3245076 is 1526464861.
Thus the answer is
uint16(keccak256(0xb388f89fb847890e2d96c9a37c9d25beadef55fb84c2b1b2cf41dc7703bd08f5, 1526464861)) = 47194.

Challenge 5

Bonus
A hard fork is that a cryptocurrency undergoes some changes in its code and the blockchain splits into two. One fork is the new chain that runs the new code and the other fork is the old chain that runs the unchanged, "old", code.
In 2016, the Ethereum community decided to implement a hard fork in an attempt to undo the harm caused by DAO, this resulted in two separate chains, Ethereum, the new chain and Ethereum Classic, the old chain.

The DAO vulnerability is kind of double spend attack. The DAO contract's withdrawBalance() function calls the fallback function before setting the user's balance to zero. So in the case that the fallback function (which can be provided by the user/client/attacker) calls withdrawBalance() again, the attacker can withdraw double the amount of his balance.