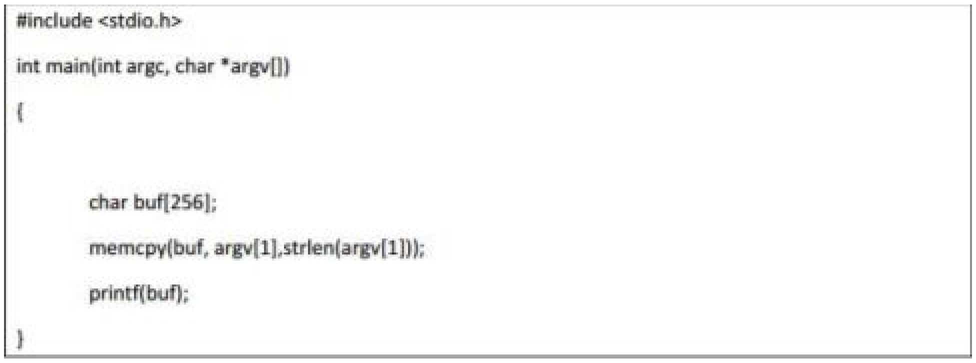
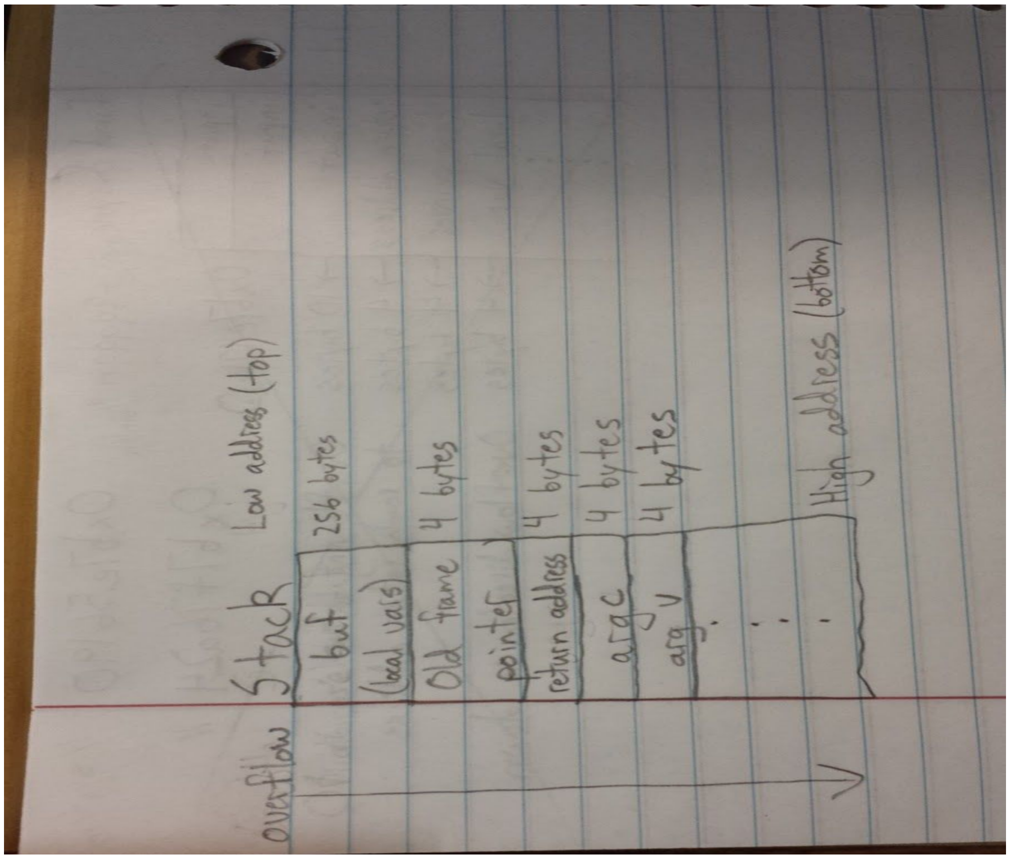
Alan Chiang

1. A. Stack



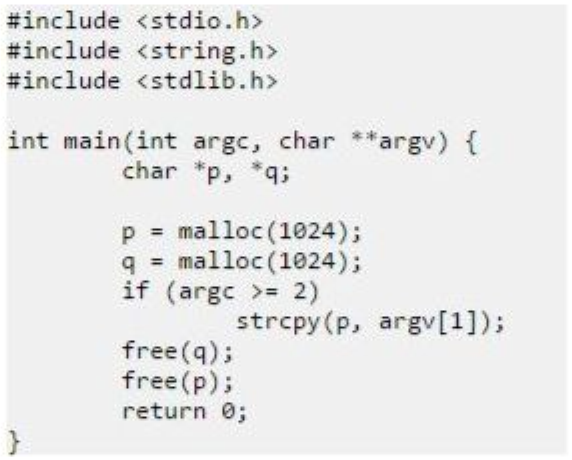
```
#include <stdio.h>

int main(int argc, char *argv[])

{


          char buf[256];

          memcpy(buf, argv[1],strlen(argv[1]));

          printf(buf);

}
```

This program is vulnerable to a return-by-libc attack. An example of this would be to overwrite the return address with the address of the system function, which would then be called after main completes. In this case, argc becomes the new return address and the first argument becomes argv. If this return-by-libc attack wanted to spawn a shell, it would then set argv to point at the string "sh" in libc.



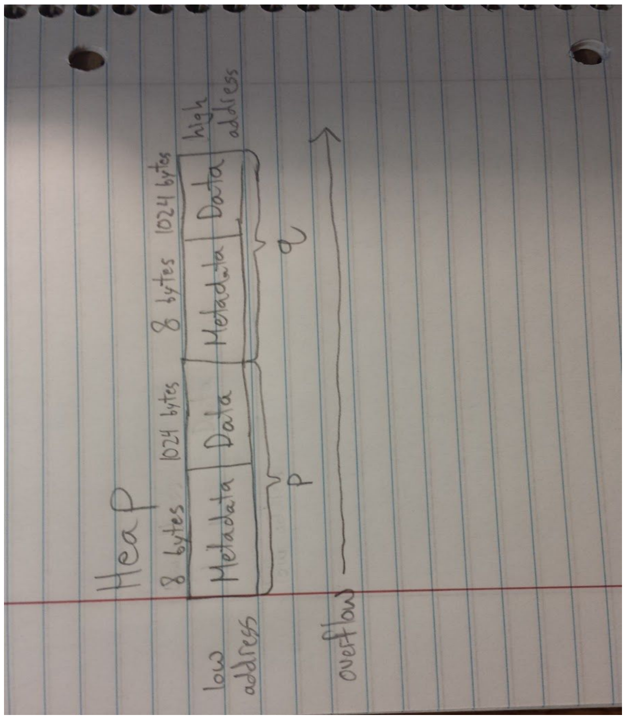
Source: https://www.exploit-db.com/docs/28553.pdf

1. B. Heap



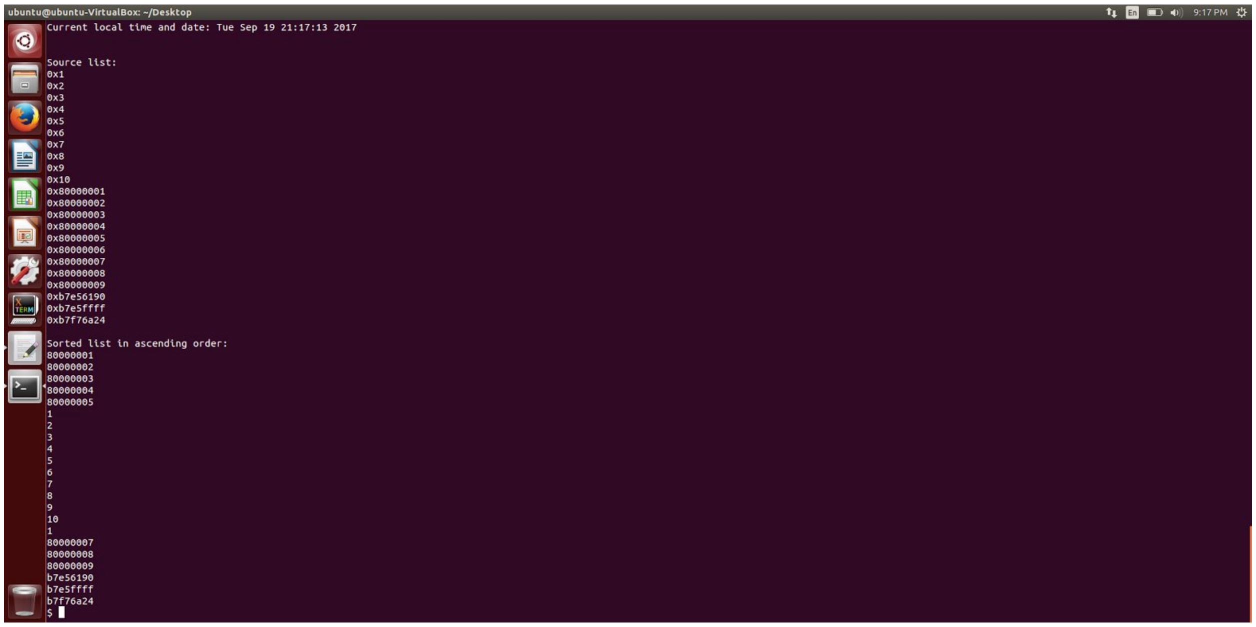
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        char *p, *q;

        p = malloc(1024);
        q = malloc(1024);
        if (argc >= 2)
                strcpy(p, argv[1]);
        free(q);
        free(p);
        return 0;
}
```

We can tell chunk q that chunk p is not in use by unsetting the least significant bit of the q chunk's prev_size value. Then we free chunk q, and because it thinks chunk p is free it will attempt to combine itself with chunk p and then remove the combined chunk from the linked list. This can be done by free navigating from q to p by subtracting prev_size and then setting p's previous chunk's forward pointer to point to p's next chunk while simultaneously setting p's next chunk's back pointer to point to p's previous chunk (thus removing p). And since we can use overflow to write into prev_size, we can trick the program into think at the forward and back pointers are values in the buffer, which are actually set by us. Creating fake chunks in the buffer allows us to write arbitrary values into arbitrary locations in memory via the assignment statements in unlink.



Source: http://www.win.tue.nl/~aeb/linux/hh/hh-11.html

2. (higher-quality screenshot attached seperately)



```
Current local time and date: Tue Sep 19 21:17:13 2017

Source list:
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0x10
0x80000001
0x80000002
0x80000003
0x80000004
0x80000005
0x80000006
0x80000007
0x80000008
0x80000009
0xb7e56190
0xb7e5ffff
0xb7f76a24

Sorted list in ascending order:
80000001
80000002
80000003
80000004
80000005
1
2
3
4
5
6
7
8
9
10
1
80000007
80000008
80000009
b7e56190
b7e5ffff
b7f76a24
$ 
```

3. The biggest difference between jump-oriented programming and return-oriented programming is evident in their names. Whereas return-oriented programming overwrites return addresses/arguments, and therefore uses the stack to point the flow of execution towards gadgets (and chain those gadgets together), jump-oriented programming uses indirect jump instructions in sequence to move between gadgets (and chain them together). Both of them are code-reuse attacks that make use of snippets of malicious code, called gadgets, which are connected through the control flow. Both of them also end each gadget with a ret or jmp instruction, respectively. But because jump-oriented programming does not use the stack this allows jump-oriented programming to circumvent a major weakness of return-oriented programming, namely its reliance on the stack, from which almost all known countermeasures defend against. However, this comes at a cost, since return instructions are by their nature bidirectional, and can return to the address stored on the stack. Jump instructions on the other hand are unidirectional, and until recently this obstacle prevented jump-oriented programming from taking off. With the recent development of a dispatcher gadget, which controls the flow of lesser functional gadgets, jump-oriented programming has moved from theory to reality.

While jump-oriented programming does introduce a new attack paradigm, it is not without its own weaknesses. Manually constructing attack code is more complex than in return-oriented programming, because specific registers must be reserved for saving the state, and specific gadgets may have to introduced solely as infrastructure for other functional gadgets. Also, whereas full Turing-complete operation is possible with return-oriented programming, jump-oriented has not yet been proven to be capable of such.