Lab 2 Guide: be evil?:

CYBR 570 : Reverse Engineering
Spring 2025

Submitted By: Lilly Carlascio
Submission Date: 4/16/2025

1. Identify the initialization of the linked list.
   (a) I was able to identify the initialization of the linked list by finding the main function, which had a lot of function calls, so it was safe to assume that it was main, making calls to a lot of the sub functions. I looked for malloc calls, to find the initialization of the linked list, and then once I identified that variable I was able to auto create a structure for the linked list, naming it intll_t, and defining the variables in the struct as next, and data.
   (b) This binary was prepared differently than how I prepared mine (which was expected) and there was a head node with valid data and then a next node that was initialized to null. Once I identified that those were two separate nodes, I was able to see that the null node was assigned as the next node of the head node. The head node had initialized data of 44 and the next node had initialized data of 55.
2. Identify each function (including main).
   (a) **main:** As said before, I was able to identify main from looking through the functions on the left symbol window and looking for the function that had the most calls to other functions, as well as some initializations.
   (b) **print_ll:** Finding print_ll was pretty simple since I asked my partner to leave in two print statements, so I was able to just look for printf and puts calls so I was easily able to name that function.
      i. The output of the print_ll function was this, which helped me to deduce the rest of the functions:
         A. First print statement - 4455223399
         B. Second print statement - 882233
   (c) **append:** Identifying append was also pretty easy since I had already identified the linked list structure and this function had parameters that looked like the head node as well as a data value. When looking inside that function, there was a loop that went all the way to the end of the list, then there was a malloc call to allocate new space for a new node to be added at the end the new node was being assigned the data parameter being passed in, and then the linked list was returned.
      i. The append statement was called 3 times with values 22, 33, and 99 which explains the first print statement of 4455223399.

(d) **prepend:** The next function to identify was prepend which came a functions down and passed the head value and the value 88. I was able to identify this value since the next print statement started with a value 88, so I looked for the function that passed 88 and assumed that would be the prepend. Looking inside the function, the suspicions were confirmed and there was a malloc call to create the new node as well as assigning the new node as the new head, and making its next value be the previous head, making it pre-pended to the beginning of the linked list.

(e) **insert:** This was the next function identified since it passed 3 parameters, the head node, the index, and the value, so I assumed it would be the insert function. It passed the value 77 and the index 2, which didn't actually show up in the print statement, which makes sense since the delete function is called right after the insert function n the same index. Looking inside the function, a malloc call is used to allocate new space for the node to be inserted, and there is a loop that loops through the list until the desired index is reached. Then the node is inserted and the next values are adjusted for the previous and the new node.

(f) **del:** The delete function was identified next since it only passed in the head and an index value, so the indicator on this function was that it did not pass in a data value. This call passed in the head and the index 2 which makes sense now why the 77 value from the insert function wouldn't appear in the print statement. Looking inside the function, there is a loop to reach the desired index and then the node is removed and the removed node's next is assigned as the next as the node previoius to the removed node, so no links are broken. Then the removed node is free'd.

(g) **pop_head:** These last two functions were the hardest for me to identify since they both use the free function since nodes are being removed and this binary returns the removed node value, which was different than how I did it, so I wasn't as sure what to look for. The pop_head function passes in a pointer to the head node value, and once I figured that out, I was able to take a look inside that function. Inside that function, I was able to determine there was two variables, the removed node value as well as what the new head will be. The new head is assigned to the head's next node and the old head becomes the removed node, the value is taken to be returned, and then that node is free'd from memory. Once I was able to retype these variables in this function, it became clear that this was the pop_head function.

(h) **pop_tail:** Since this was the last function to diagnose, once all the others were identified, it seemed apparent that this was the pop_tail function. This function took in one parameter that was the entire head node variable. Once inside the function there is a loop to reach the end of the linked list, keeping reference of each node until the end is reached. When the end is reached, there is a reference kept to the one node previous to the end, which will become the new end of the linked list, making it's next reference to be null. The end of the linked list, known as the node to be removed, keeps reference to its data value to be returned, and then the node is free'd from memory.

3. Describe the function calling convention observed in the binary.

(a) The calling convention being used in the binary is the 'cdecl - other' since a series of MOV calls take place before the CALL to whatever function is being used.