Lab 1 – Bomb Defusing

Nathan Paternoster

**Phase 1**

The correct password for this phase is **“Crikey! I have lost my mojo!”**. In the function “strings\_not\_equal” another function “string\_length” is called twice. In the first call our guess is loaded into $r0. After string\_length exits the length of our guess is loaded into $r5 (in strings\_not\_equal+16). In the second call of string\_length the correct password is loaded into $r0.

**Phase 2**

The correct password is **0 1 1 2 3 5**. The function “read\_six\_numbers” will check to make sure that our guess is formatted as six numbers and will also push our guess onto the stack. Back in phase\_2 the first digit of our guess is loaded into $r3. It is compared with 0, so the first digit of the password must be 0 (instruction +20). The second digit is then put into $r3 and compared with 1 (instruction +32). Therefore the second digit must be 1.

A loop then occurs from +52 to +92. Every iteration of the loop checks that the current digit (in $r2) is equal to the previous digit added to the current one (in $r3, the summation happens at +60). This is essentially the Fibonacci series. The counter $r4 determines when the loop exits. It will exit after four iterations. So the next four digits of our six-digit password will be numbers of the Fibonacci series.

**Phase 3**

The correct password is **4 0,** although the first number can actually be any positive integer below 5. We chose 4. Something from the program counter is loaded into $r1. Examining $r1 we see the string “%d %d.” This suggests that the correct password must be two integers. The function scanf puts the amount of integers of our guess into $r0. $r0 must be greater than 1, further confirming the previous speculation.

At instruction +208 our first integer guess is loaded from the stack into $r3. It is compared with 5 on the condition that it must be less than. This tells us our first digit may be any integer under 5. At instruction +220 the second digit of our guess is loaded into $r3. It must be equal to whatever is in register $r2 at this point to progress safely. The value 0 is in $r2 so the second digit of our guess should be 0.

**Phase 4**

The correct password is **11 18**. Once again scanf puts the amount of integers in our guess into $r0. $r0 must be equal to two (instruction +24), so the correct password must contain two integers.

At instruction +32 the first integer of our guess is loaded into $r3. According to the two following conditions it must be greater than 0 and less than or equal to 14 (instructions +40 and +48 respectively). In func4 our goal is to exit the function with $r0 = 18 so that the check in phase\_4 at instruction +72 will be satisfied. Because we had a small number of possibilities to work with we used trial and error to determine that the correct value for our first integer is 11.

At instruction +56 the second integer of our guess is loaded into $r0. At instruction +72 it is compared to the immediate 18. Therefore the second integer of our guess must be 18.

**Phase 5**

|  |  |
| --- | --- |
| 0 | 10 |
| +4 | 2 |
| +8 | 14 |
| +12 | 7 |
| +16 | 8 |
| +20 | 12 |
| +24 | 15 |
| +28 | 11 |
| +32 | 0 |
| +36 | 4 |
| +40 | 1 |
| +44 | 13 |
| +48 | 3 |
| +52 | 9 |
| +56 | 6 |
| +60 | 5 |

The correct password for this phase is **5 115**. The same beginning instructions are used as the previous phase to check that we have two integers in our guess. $r3 is loaded with our first guess at +36 and then AND’d with the immediate 15 at +40 to insure that the resulting number would be 15 or less. At instruction +48 a check is performed to make sure this number does not equal 15. Our first guess must be in the range 0-14.

At instruction +64 $r0 is loaded with an array of 15 elements. The array is shown on the right. $r1 counts how many times we go through a loop spanning from instruction +68 to +84 (incremented at +68). The loop must go through 15 times (instruction +92). $r3 acts as the offset of the $r0 array and is also the value of our first guess. At instruction +72 $r3 is multiplied by 4 and then used to access that value in the array. That value in the array then overwrites into $r3. The next iteration of the loop will use this new value of $r3 (multiplied by 4) to be the next location accessed in the array. The loop will be broken out of when $r3 = 15 (instruction +80). In order to go through the loop the required number of 15 times we must insure that the $r3 will go through every other value of the array before becoming 15. The correct starting position of $r3 should be the value 5 at the last position in the array.

Meanwhile, register $r2 acts as the summation of all the values reached in the array (minus the first value). Our second integer guess is loaded into $r3 at instruction +100. It is compared to this summed result in the next instruction to check for equality. Therefore our second guess must be equal to the sum of the values in the array (minus the first value entered, 5). This means our second guess must be 115.

**Phase 6**

The correct password for this phase is **5 1 4 3 2 6**. The solution for this phase consisted of six numbers that each represented a link in a link list. The link list had to eventually be ordered in such a way that the values were in descending order. The values of the links couldn’t be affected but the addresses to the next link (essentially the order of the list) could be.

Instructions 0 to +12 checked to see that there were six numbers. Instructions +16 to +108 consisted of a loop within a loop. In the outside loop every digit had 1 subtracted from it and compared to 5 (instruction +36) to make sure every digit was in the range 1-6. In the inside loop each digit was compared to every other digit to make sure there were no identical digits (instruction +80).

|  |  |
| --- | --- |
| 0 | 85 |
| +4 | 1 |
| +8 | MEM |
| +12 | 922 |
| +16 | 2 |
| +20 | MEM |
| +24 | 434 |
| +28 | 3 |
| +32 | MEM |
| +36 | 194 |
| +40 | 4 |
| +44 | MEM |
| +48 | 90 |
| +52 | 5 |
| +56 | MEM |
| +60 | 850 |
| +64 | 6 |
| +68 | MEM |

The loop from instruction +112 to +132 would subtract all of our six input numbers from 7. We had to account for this switch in the final ordering of the links.

Instructions +136 to +212 consisted of another loop within a loop that wrote a set of values (instruction +176) to the stack depending on the order of our input values. The values we got were set up in an order close to this array shown on the right (the first link appeared to be in a different location in memory from the other links).

MEM means that the value there was a value of some other location in memory (specifically one of the other link’s data value). This value is equivalent to the “next” parameter of a link in a link list. Some example values we got for this parameter are: 78468, 78228, 78456, etc. Each link is given three spaces in memory. The first is its value (e.g. 85, 922, 434), the second its link number (e.g. 1, 2, 3), and the last its last is the next link it points to.

Instructions +216 to +264 loaded these values from the stack onto register $r4. The values of the links will be accessed by $r4. The loop from instruction +272 to +374 will execute five times. Each time, the value of the current link $r4 is on (specified by our input sequence and represented by $r2) is compared to the value of the next link in the list (represented by $r3). The current link’s value must be greater than the next link’s. Because of this we know that the link list must be ordered in descending order.

Working out the values of each link the correct solution would be to build the link list in the order 2 🡪 6 🡪 3 🡪 4 🡪 5 🡪 1 (values: 922 🡪 850 🡪 434 🡪 194 🡪 90 🡪 85). Conveniently, the numbers we entered match up to the order of the link list. However this is not the correct input sequence. The function earlier that subtracts our numbers from seven will end up ruining this sequence. The correct sequence is therefore 7 minus each value here. So the final correct password is 5 1 4 3 2 6.