* Phase 1
  + My approach to this problem was pretty straight forward. The idea is that we want to overflow the buffer and then redirect to touch1. So, my logic was that I need fill the buffer and then provide the address of the touch1 function to go there. First, I needed the size of the buffer so I could overflow it. I got this by setting a breakpoint at getbuf <b getbuf> and then running <disas>. I see that 0x28 bytes have been reserved for input. I convert the hex, and see this is the decimal 40. So, our buffer is 40 entries long. I also need the address of touch1, so I type the command <info address touch1> . I noticed initially from the objdump that touch is right after getbuf in the assembly code, so this address makes sense. From here, I create a text file with a buffer followed by the address of touch1. Bytes reversed. When I run this using the <cat p1.txt | ./hex2raw | ./ctarget>, we get the solution.

A black and silver text on a screen

Description automatically generated



* Phase2
  + So, for this phase I am told I need to write some code, and I am also told to keep in mind register %rdi. I know that after my buffer I want to be sent to touch2, and I want to pass my cookie as an argument. From here, I know I need to get the address of touch2 to put after the buffer, and I know I need to move my cookie into register %rdi, because I see this is the argument that will be used in the <cmp> statement in the disassembled code for touch2. So, I created a .s file to write my assembly code in which my cookie is moved to this register. I have attached that file below. After that, I simply tried the same approach as phase1. That is provide a buffer with a size of 0x28, followed by the address of touch2 and the address of $rdi. I realized though after inspecting touch2, that %rdi did not have the value I expected. After this I realized that the .s file I wrote needed to be compiled, or else it wouldn’t do anything. So I created a .o file with <gcc -c p2.s>. Looking at the objdump disassembly code, I see some hex of the address of my p2.s file. I get the idea that this is the address I needs to be included as well. So at this point, I am providing a buffer of zeros, followed by the address of touch2, %rdi, and my code injecting the cookie into %rdi. However, this does not work, so after moving the code around, I ended up putting the cookie code within the buffer, after I read the instruction again and realize that this injection needs to happen BEFORE we access %rdi. When I structure my injecting this way, I get a misfire. I realize that because I am putting my injection into the buffer, I need to access %rsp because this is the address holding the buffer. So, my final txt solution contains 1) injection of cookie 2) buffer 3) %rsp address 4) touch2 address.

A screenshot of a cell phone

Description automatically generated



A screenshot of a cell phone

Description automatically generated

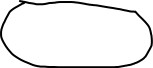
A picture containing meter, clock

Description automatically generated

* Phase 3
  + This phase is quite similar to the previous one. Only this time, the cookie needs to be in string format. I first attempted to push the same txt as phase2, except with the cookie in string format. I realized this would give me an error as I <stepi> through the disassembled code because my cookie is not being placed into the register I expected. I looked back over the instructions and realized that my cookie is being overwritten by the function <hexmatch>. So, my approach then was to place the cookie in the register, however offset by the number of bytes that come before it. Because my buffer is 0x28 bytes, I need to add my cookie to %rdi + 0x28 bytes + 8 bytes for addressing touch3 + 8 bytes for addressing %rsp, which again I need because that is where my cookie injection is happening, same as in phase 2. I get the address of $rsp from <I r> after setting a breakpoint at getbuf similar to the previous phase, and I use <I address touch3> to get the address of touch3.When I do this, and place my txt cookie as the last 8 bytes in my txt file, I get the solution. The key difference in this phase compared to phase 2 was to offset the memory address of where my cookie is being stored.

A screenshot of a cell phone

Description automatically generated



A close up of a piece of paper

Description automatically generated



A picture containing meter, people, room

Description automatically generated







* Phase 4
  + So, I knew this phase was similar in structure to phase 2. By this, I knew my answer would be a buffer, followed by my cookie and the address of touch two. The difference in this phase is that I needed to find bytes of code to insert my cookie for me instead of writing it myself. I started by looking at the objdump of rtarget to look for gadgets that would be useful to me. I knew what I had to look for thanks to phase 2. I needed to find a gadget that pushed my cookie into a register, and then have this register get assigned to %rdi. I found the two function addval\_353 and setval\_299 to do the job for me. Setval\_299 pops %rax for me and returns it. There are two bytes of code between the code and return, but because they are 90 or no operation, this is fine. I think used addval\_353 to push this value from %rax into %rdi. After knowing I had my two gadgets I tried to set up my text file, but kept getting segmentation errors. I realized that the address provided in the dump file was not correct, similar to previous questions, so I jumped into gdb to find the real address of these two functions. Once I located this, I was able to format the text file in such a way that the buffer is flooded, then the gadget one (setval\_353) is used, then the cookie (cookie must follow the pop), then gadget two or my assignment of the pop into %rdi, and finally the address of touch2 which is the same as phase 2. Once I did this, I got my solution.















