HINTS/NEWS:

1. Read about buffer overflows: [buffer overflow](https://www.owasp.org/index.php/Buffer_Overflow)
2. Read about using gdb: [using gdb](http://www.unknownroad.com/rtfm/gdbtut/)
3. Need a UNIX shell (Terminal/Cygwin)
4. USE LINUX!!!! (most binaries are compiled with ELF 64-bit binary LSB exec.: x86-64)

**SCORE: 210/710**

|  |  |
| --- | --- |
| TITLE: | Regex 1: Classy Punctuation |
| PROMPT: | Joe wants to know: He can be a bit dramatic with his punctutation, but does he ever repeat it 23 times? Help him write a regex that matches 13 characters each of which is ‘’.’’, ‘’!’’ or ‘’?’’? |
| HINT: | Learn about regular expressions at [RegexOne](http://regexone.com/). |
| HOW: | Set of “.”, “!”, and “?” in any appearance in a set of 23 items. |
| FLAG: | [.!?]{23} |

|  |  |
| --- | --- |
| TITLE: | Regex 2: Don’t Look Back |
| PROMPT: | You have been tasked with taking over FlexCorp’s web 3.0 project as the previous lead developer just went into retirement. As you shield your eyes from the unmaintable horror, you see a little regex scrawled against his desk with a little inscription reading “how to parse Web 3.0 data.” Presumably, he was talking about the large, shared CSV file in which everyone at FlexCorp had to log which tools they used on company grounds.  You know that his first name begins with an C and his last name ends with an N.  Here’s the [CSV file](https://www.pactf.com/static/ctfproblems/e392773e-3611-4acb-8f08-7bfacc95eea3/haystack.3db9ab09dabb.csv), and here’s the regex:  ^(?:.\*?,){29}(?:c\w+n),(?:.\*?,){35}(.\*?),.\*$  Can you find the name of the tool?! |
| HINT: | We weren’t kidding about the file being large. You’ll want to have a regex a bit cleverer than the veteran’s for this… Backtrack a bit to learn about character classes. |
| HOW: | \*\*This is only accessible after completing “Regex 1: Classy Punctuation”\*\*   * /^(?:.\*?,){29}(?:c\w+n),(?:.\*?,){35}(.\*?),.\*$/   + ^ assert position at start of the string   + (?:.\*?,){29} Non-capturing group     - Quantifier: {29} Exactly 29 times     - .\*? matches any character (except newline)       * Quantifier: \*? Between zero and unlimited times, as few times as possible, expanding as needed [lazy]     - , matches the character , literally   + (?:c\w+n) Non-capturing group     - c matches the character c literally (case sensitive)     - \w+ match any word character [a-zA-Z0-9\_]       * Quantifier: + Between one and unlimited times, as many times as possible, giving back as needed [greedy]     - n matches the character n literally (case sensitive)   + , matches the character , literally   + (?:.\*?,){35} Non-capturing group     - Quantifier: {35} Exactly 35 times     - .\*? matches any character (except newline)       * Quantifier: \*? Between zero and unlimited times, as few times as possible, expanding as needed [lazy]     - , matches the character , literally   + 1st Capturing group (.\*?)     - .\*? matches any character (except newline)       * Quantifier: \*? Between zero and unlimited times, as few times as possible, expanding as needed [lazy]   + , matches the character , literally   + .\* matches any character (except newline)     - Quantifier: \* Between zero and unlimited times, as many times as possible, giving back as needed [greedy]   + $ assert position at end of the string |
| FLAG: |  |

|  |  |
| --- | --- |
| TITLE: | Regex 3: Oh, The Power! |
| PROMPT: | By now you are acquainted with the powers of regex, but can you make a regex match only the powers of 5? Only a string of 5^n repeating ‘x’s should pass.  You have all of 30 characters. |
| HINT: | Take away 4/5ths of a power of 5, and you have: a power of 5. |
| HOW: | \*\*This is only accessible after completing “Regex 1: Classy Punctuation”\*\* |
| FLAG: |  |

|  |  |
| --- | --- |
| TITLE: | Guessing Game |
| PROMPT: | The evil villains have hidden their flag behind a [machine](https://www.pactf.com/static/ctfproblems/dc1f74e3-b1d3-4a4b-84b0-d7b9d4d05959/RandomGame.ca9613cea356.java) that will only spit it back out if you can guess a random number, how devious! However, their programmer wasn’t quite sure how to make *absolutely sure* that things were random, so there’s a hole in their program! Can you exploit it?  nc 104.236.216.251 64753 |
| HINT: | What happens if the same person plays multiple times? |
| HOW: | 1. Enter the same characters for “name” every single time. 2. Enter any # for the first try. 3. Since it tells you the random number, enter the number it gives you because it is calculated based on the input of the name. |
| FLAG: | PA{m1Nd\_j00r\_sE3Dz} |

|  |  |
| --- | --- |
| TITLE: | RabbitJS |
| PROMPT: | All but the 1, [, ], and = keys broke on JavaScript Mage Joey’s keyboard. That didn’t stop him from trying to tell the world what he had [learnt](https://www.pactf.com/static/ctfproblems/d44928d1-e6eb-4f71-82fc-ecd89528e990/rabbit.98d21ac0d2e3.js), though. |
| HINT: | Sometimes, it’s best to leave a viewpoint function unevaluated. |
| HOW: | Using JSFuck (jsfuck.com), the javascript presents a dialog asking: “How deep does this rabbit hole go?”.  Using Google Chrome, I set a breakpoint on line 167 of (index). Next, I examined the Global Scope in the developer tools debugger. Then, I scrolled through the variables down to window, expanded it, and read them until i found the flag. |
| FLAG: | flag{carroll\_knew\_it} |

|  |  |
| --- | --- |
| TITLE: | Magic |
| PROMPT: | Wanna see a [magic trick](https://www.pactf.com/static/ctfproblems/9bd85aa2-70c7-4f42-b080-9d6f18201a6d/magic2.97b4d643a0aa)? |
| HINT: | I wonder if you could get into memory and see the magic? |
| HOW: | \*\*Need to chmod +x the binary first\*\*  When running the binary in Terminal, it prints out “The magic is happening at 0x7ffdc22f8a80...”  Using gdb, I set a breakpoint right before the program ended to access its memory at the given address.  The flag is located at that address in memory. |
| FLAG: | pactf{eax\_more\_like\_easy} |

|  |  |
| --- | --- |
| TITLE: | Foolish Filesystem |
| PROMPT: | You found an [interesting program](https://www.pactf.com/static/ctfproblems/d64eb54d-f824-4028-9e4c-abad50508e85/adventure2.f383cdad8364) on the floor… wonder what it is? |
| HINT: | What is a directory? A miserable little pile of hidden files! |
| HOW: | \*\*Need to chmod +x the binary first\*\*  Run the binary in Terminal. Type in “ls -la” (show hidden files).  I used a hex editor to figure out the commands.  Type “cat .sekrit” to retrieve the flag. |
| FLAG: | pactf{H1d3\_uR\_s3kr1tz} |

|  |  |
| --- | --- |
| TITLE: | Sliding Letter Game |
| PROMPT: | nc 198.211.110.148 64753  It’s the letter matching game! You just need to change the characters of a string so it matches a password! The source is [here](https://www.pactf.com/static/ctfproblems/917f5c00-b29c-42a5-975b-4e0f71f548a3/passgame.c9f68aafbc1d.c). |
| HINT: | What happens if you make a number that’s *really* big? |
| HOW: | String is  Abrac4dabr4h0cuspocu5youv3done1t  But to get flag you have to overflow buffer and set secret (long int - 4 bytes) to 0x1337  Watch “Buffer Overflow Attack - Computerphile” on Youtube: <https://www.youtube.com/watch?v=1S0aBV-Waeo>  BUFFERSIZE = 256  Need to change “secret” or change the compare at line 115 but the stack is flushed when fflush(stdout) is called???  You dont need to manipulate the stack. You can change negative indexes  Change index -32 to ‘7’ because that has the character code 0x37. Now secret is 0x37  Change index -31 to 0x13… but that is not a character. I can change it to 0x10 just by pressing enter, but the console input won’t take 0x13  I haven’t been able to change the damn thingy idk what you call it so I’ve hit a wall  Use automated input: python script.py | nc ip port |
| FLAG: |  |

|  |  |
| --- | --- |
| TITLE: | Huge Numbers |
| PROMPT: | How many 10,000-digit numbers (without leading zeros) exist such that no three adjacent digits have a sum greater than 9? The flag is the first 15 digits. |
| HINT: | How many digits do you have to keep track of at once? |
| HOW: |  |
| FLAG: |  |

|  |  |
| --- | --- |
| TITLE: | Whitespace |
| PROMPT: | I’ve found this [java program](https://www.pactf.com/static/ctfproblems/f1a496d4-bae1-4042-8358-b62576ccc8ce/whitespacejava.5d3038df55e9.java) on someone’s computer that looks fishy. |
| HINT: | I wonder, is that really Java? |
| HOW: | Its stenography  ---  It’s written in the whitespace programming language, but it’s disguised as a java program. Upon copying & pasting the code at <http://www.tutorialspoint.com/execute_whitespace_online.php>, executing the program prints: “The flag is: two\_programs\_in\_one”.  Wow that is really cool good shit on that  **Ya I have no idea where to go with it. I’m looking at the ROM file structure but I feel like this requires more information from online or something**  **Someone on the chat (pactf.com/chat) told me to look for an original file online, but I couldn’t find anything useful.** |
| FLAG: | two\_programs\_in\_one |

|  |  |
| --- | --- |
| TITLE: | Sorcery |
| PROMPT: | What is this magic? My file is [here](https://www.pactf.com/static/ctfproblems/b0d1ed57-0c27-4b43-b599-e70a760d9274/sorcery.5a9777120468), but where are my hardcoded strings? |
| HINT: | And as Dorthy was told, “Follow the yello brick pointers.” (We recommend using [this online tool](https://www.onlinedisassembler.com/static/home/)! It’s pretty nice.) |
| HOW: | \*\*\*\*\*UPDATED BINARY LINK:\*\*\*\*\*[here](https://www.pactf.com/static/ctfproblems/b0d1ed57-0c27-4b43-b599-e70a760d9274/sorcery.5a9777120468)  !qzsaxwcefdvrbtngh{ymjukilop16253950}4928374af\_-  is allocated on the stack  There is a hardcoded string “personality” after you disassemble the binary  The \_z4Flagv… subroutine is fucking huge  I dont have time to go through this stuff. The assembly is made purposefully confusing with a bunch of loops that change the stack. You have to follow the registers eax, ecx, and edx because a pointer is passed around between them. What you can try to do is set a breakpoint at 401f52 and look at the stack there  Its an OS X Mach-O 64-bit executable.  So the \_z4Flag subroutine was really only important in two parts  The first part like I said above just made a character array and put the above characters on the stack  There are 48 lines just moving a byte to increasing locations on the stack:  “ mov byte [rbp-0x59B], 113 # 0DB2 \_ C6. 85, FFFFFA65, 71”  This means that the literal value ‘113’ (decimal) is being pushed to the stack base pointer (rbp) - 0x59B bytes. The stack continues to grow downward with each line.  I used a python script (maybe ill post later if i clean it up lol) to take this decimal value on each line, then get the corresponding ASCII character and print them all out, resulting in the string : ‘!qzsaxwcefdvrbtngh{ymjukilop16253950}4928374af\_-’  The second used a pointer to manipulate the stack (really in C it would just be accessing the character array using regular indexing, but in assembly this is simplified down to pointer logic)  The first thing you notice is the line  “ mov al, byte [rbp-0x5AC] # 0EFB \_ 8A. 85, FFFFFA54”  Which moves the value in the stack location (base pointer - 0x5AC) to the lowest priority section of the ax register. Effectively this puts a single byte in the ax register from the stack… characters are a single byte.  Next is  “ mov byte [rbp-0x5CA], al # 0F01 \_ 88. 85, FFFFFA36”  Where the same single byte is being moved from the ax register back onto the stack, this time at the location following the last location accessed in the first part, so it continues growing down the stack.  These two lines keep repeating, effectively copying items from the character array onto lower parts of the stack. If we track the address used in the first line (ex. 0x5AC) to see what was allocated in that location in the first section of the sub (which again, I used a shitty python script for), we find that it allocated the string “{1\_dun\_ducked\_up!}}” onto the bottom of the stack.  Obviously this doesn’t look like random garbage, so try it as the solution and it worked.  The next 2 sections of the subroutine begin by accessing a location on the stack far earlier than anything we have existing information about, so it is effectively useless. If you want to know specifics, the first one moves the value at rbp-0x790 (significantly before this function's scope, or at least the parts that have been accessed) around to various locations in the stack. This is earlier than the first hardcoded character section accesses, so we can’t pull any information from it.  Completely meaningless without knowledge of what is at that location... |
| FLAG: |  |

|  |  |
| --- | --- |
| TITLE: | Coalmine |
| PROMPT: | nc 104.236.216.251 59292  Keep digging! We’re running out of [oxygen](https://www.pactf.com/static/ctfproblems/f6d04062-fc2f-4de4-af69-d5839598a4c4/oxygen.cc66894943f7)! Check the source [here](https://www.pactf.com/static/ctfproblems/f6d04062-fc2f-4de4-af69-d5839598a4c4/oxygen.e3b0a313de9a.c) |
| HINT: | How could we figure out what the canary value is? |
| HOW: | print 'A'\*0x10c + '\x19\x7c\x04\x00' + 'A'\*8 + <addr of getFlag>  0x47c19  <http://i.imgur.com/TlTleM4.png> |
| FLAG: |  |

|  |  |
| --- | --- |
| TITLE: | Jump |
| PROMPT: | [Don’t fall off…](https://www.pactf.com/static/ctfproblems/33b3dafa-d1a4-43a0-a616-1283aa04a8d0/handout.73f14f2035f1.zip). |
| HINT: | I wonder where I’ve seen this before? |
| HOW: | It’s a Nintendo GBA ROM Image (according to “file”).  Upon opening it with an emulator, Mario and Kirby are on a cliff.  Another hint: find an original file.  ORIGINAL:  <http://www.gbadev.org/demos.php?showinfo=1275>  Diff with original  First diff is a jump which goes to code which changes data at 0x200`0000 to the next char in the flag |
| FLAG: |  |