GreyCTF writeup

Chai Yichen, Ho Jie Feng, Leonard Ong

June 2022

1 Permutation

The challenge presents an element $g \in S_{5000}$ along with g^a and g^b where $a, b \in \mathbb{Z}$. Presented here, we have some facts:

1. g^a and $g^b \in \langle g \rangle$, the cyclic group generated by g.
2. $\langle g \rangle$ is an abelian group, and by the structure theorem of finite abelian groups, it is a direct product of cyclic groups.
3. The cyclic groups are precisely the cycles when the elements of any the ith element is fixed after applying g^x where x is the size of the group.

For each element, record the x_i such that $g^{x_i} \equiv g^a$ and the size of the cyclic group. A simple application of Chinese Remainder Theorem completes the solution.

2 Equation 2

Rearranging, we have $7m_2 = g - m_1^2$. Substituting this into the first equation, we get a 4th degree polynomial equation modulo p. Solving this gives m_1 directly, and m_2 can be calculated easily.

3 Hypersphere

Let x, y, z be three quarternions, we define $f_x : \mathbb{H} \to (\mathbb{H} \to \mathbb{H})$ as $f_x : x \mapsto x \cdot -$, i.e. the multiplication map.
Note that $f_x(y + z) = x \cdot (y + z) = x \cdot y + x \cdot z = f_x(y) + f_x(z)$ and $f_x(ky) = x \cdot ky = k(x \cdot y) = kf_x(y)$. Since $\mathbb{H}$ is a vector space, so f_x represents a linear transformation and can be represented by a matrix. Solving the matrix by multiplying a quarternion by the basis vectors 1,**i**,**j**,**k**, we get:

$$A = \begin{pmatrix} a & -b & -c & -d \\ b & a & d & -c \\ c & -d & a & b \\ d & c & -b & a \end{pmatrix}$$
$$\det(A) = (a^2 + b^2 + c^2 + d^2)^2 = 1^2 = 1$$

The characteristic polynomial has roots $a \pm \sqrt{-b^2 - c^2 - d^2}$. Since we are acting in Z_p, We could choose $-b^2 - c^2 - d^2$ such that it is a quadratic residue of p and so the square root exists.

Taking the Jordan canonical form, we have $A = PJP^{-1}$. Taking J_b as $P^{-1}BP$, and we find that the Jordan blocks are single elements, so we can solve the discrete logarithm element on Z_p instead. Choose a p where $p - 1$ is easily factorized will allow us to solve it easily.

4 Coopy

This challenge presents a custom written vector class which works by storing items in an internal array and expanding the array as needed.

From lines 37 to 40, the code implements indexing into the vector class. While it checks if the index may exceed the length of the internal array, it does not check if the index may be negative, which is possible as index is of the signed int type. As such, we can make use of negative indices to access out of bounds memory below the ptr array.

In C++, a string is a 32 byte structure. When a the length of the string's data exceeds 32 bytes, heap memory will be allocated to store the data, and the heap pointer will be placed at offset 8 in the aforementioned string structure. The pair of 8 bytes following the pointer represent the string length as well as the actual length of the heap chunk used to store the string.

By adding two strings of sizes `0x700` and `0x200`, we induce a unsorted bin chunk of size `0xea0`. We then add three more random strings, which causes the realloc function of the vector class to be called. Now, the internal array ptr of the vector class resides above our string of size `0x200`.

Now, by accessing a negative index on our vector class, the data in the 0x200 string will be treated as a string structure. By faking a string structure with an arbitrary pointer and sizes, we can achieve an arbitrary read and write primitive by reading and editing the string respectively.

By first using print_stats on a legitimate string object, we obtain a pointer to heap. By some static offset calculation, we can find the top of the heap, where pointers to the LIBC reside. With our arb read primitive, we can leak these pointers.

From here, there are a variety of ways to solve the challenge. Due to some mistakes made during the challenge, I took a longer way. I used arbitrary read to leak a stack address via the LIBC symbol environ and subsequently wrote a ROP chain on stack to call `system('/bin/sh')`.

5 Runtime Environment 2

The binary is compiled Haskell, which makes it difficult to conduct static analysis due to the high level of indirection present in the binary. Dynamic analysis cannot be done easily as well, as Haskell programs do not rely on the standard call and return calling convention in other languages, meaning that backtraces are practically nonexistent.

The binary itself works in a simple fashion, taking an input and providing an output, with no additional messages or menu. I decided to start my dynamic analysis from the point where output is printed and work backwards.

As the Haskell heap addresses remain constant across runs in gdb, I was able to set a write watchpoint on the output buffer, which leads me to a call to memcpy, suggesting that the output buffer was copied from another memory location. I set a breakpoint on the caller to memcpy and indeed found the other buffer storing the same output data.

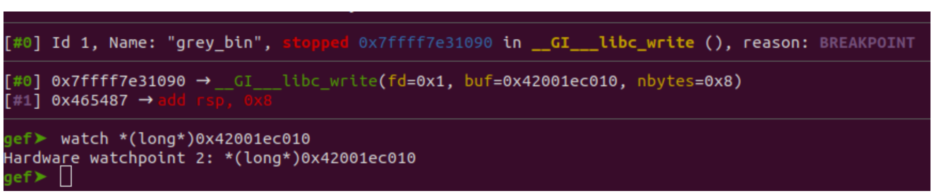

Figure 1: Write watchpoint on output buffer

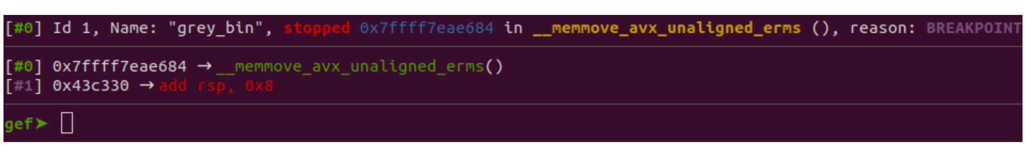

Figure 2: Watchpoint brings us to memcpy

Figure 3: Breaking on caller of memcpy

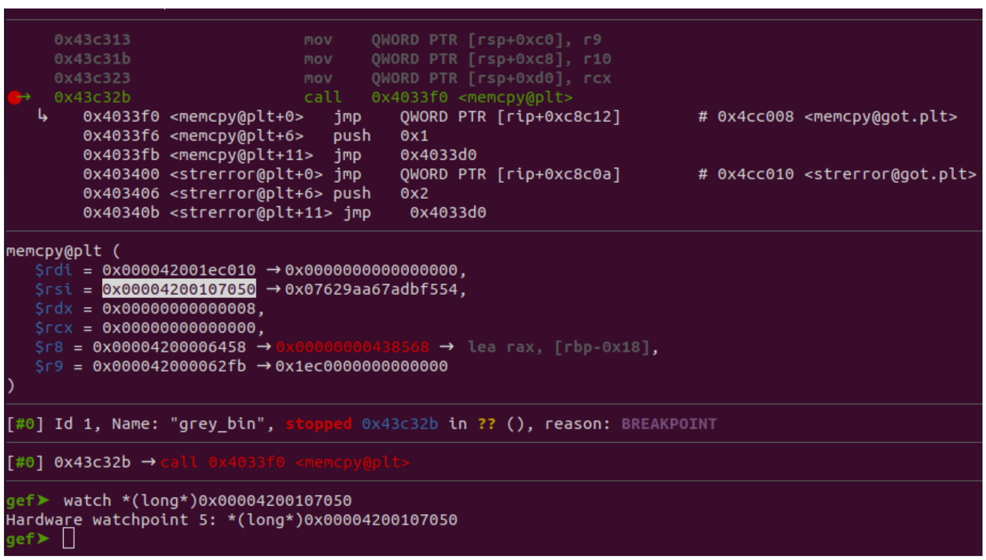

Figure 4: Write watchpoint on second buffer

Setting a write watchpoint on this second buffer, I was lead to the function `bytestringzm0zi10zi12zi1_DataziByteStringziInternal_zdwgo1_info`, which is the encoded form of `bytestring-0.10.12.1_Data.ByteString.Internal_$wgo1_info`. By breaking on the mov instruction at `0x40b5b0`, I figured that the individual bytes in the second buffer appear to be stored as quadwords in this bytestring's internal structure.

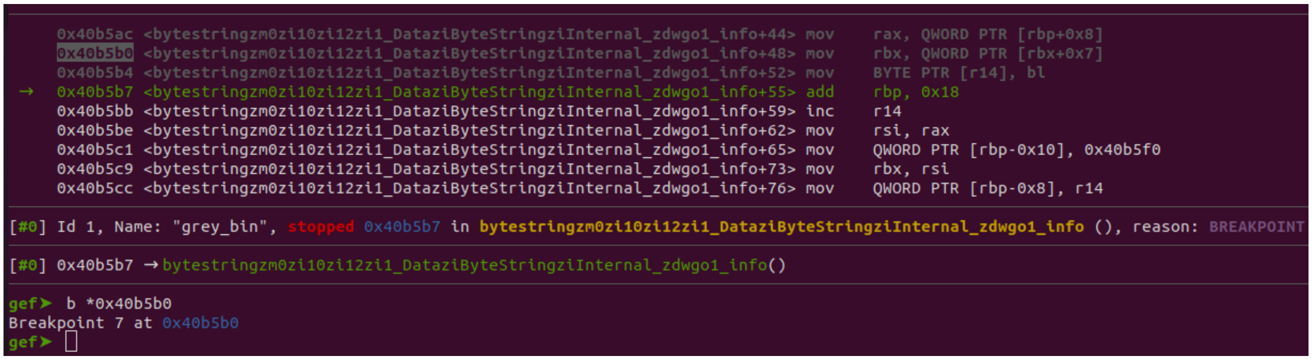

Figure 5: Write watchpoint on second buffer leads us here

I realised after scanning the whole process space that the quad word in fact only appears 4-5 times in the whole Haskell heap. As usual, I added write watchpoints on each of them. One of the watchpoints triggered in `base_GHC.Int_$fBitsInt32_$cxor_info`, the xor operation, which indicated that I was on the right track.

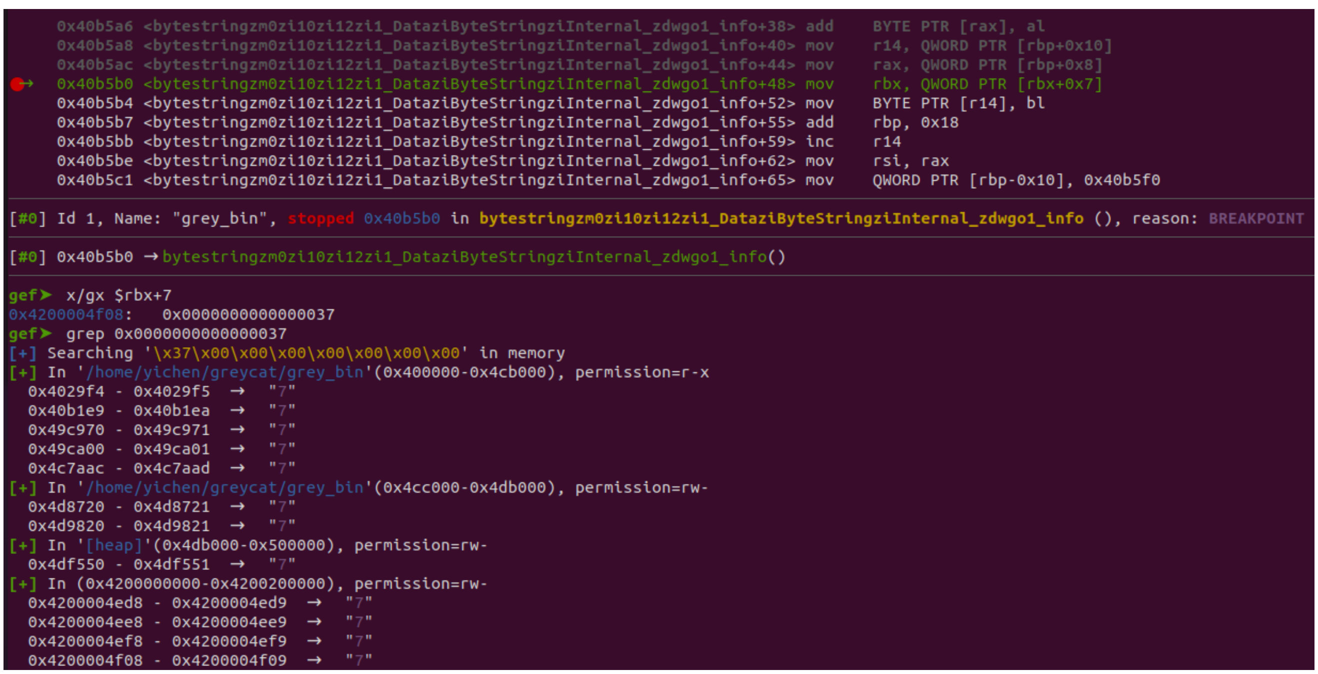

Figure 6: The quadword shows up 4 times in the Haskell heap

Figure 7: Ending in the XOR function

By breaking on `0x43e45d`, I observed something interesting. The epoch in seconds (knowledge obtained from other testing) stored in rbx appears to be repeatedly XOR'd with three unknown values before it is XOR'd with one byte of our input. The process was repeated for each character in our input to produce the output.

By repeating the same trick of setting write watchpoints on the memory location of the unknown values, I arrived at `base_GHC.Int_$fBitsInt32_$cshiftR_info` and `base_GHC.Int_$fBitsInt32_$cshiftL_info`. From here, it was easy to see what the program was doing. With the internal state initially set to epoch, the internal state was bit shifted and XOR'd with itself three times. This is a LFSR, seed by the epoch.

Guessing that challenge.bin provided the input to the program, I implemented the LFSR in python and produced the output for challenge.bin based on the epoch timestamps for the past few days. At epoch 1654353866, I got the flag.