Flare-On 5: Challenge 6 Solution – magic

Challenge Author: Sebastian Vogl

Overview

In this challenge, we receive a 64-bit Linux binary called magic. The binary is partly encrypted and uses self-modifying code to permutate the binary on disk and in memory. To solve the challenge, we have to enter 666 keys. Each of the keys is a permutation of the same key that is generated while the binary modifies itself. The functions that magic uses to validate each key are encrypted and use various well-known algorithms for validation. There are in total seven of these comparison functions in the binary.

We will provide two solutions to the challenge. The first solution uses a combination of static and dynamic analysis. The second solution uses a brute-force approach to retrieve the flag.

Analysis

Initial Analysis

We start our initial analysis by executing the binary that was given to us in virtual machine. We see the following output:

```
magic$ ./magic
Welcome to the ever changing magic mushroom!
666 trials lie ahead of you!
Challenge 1/666. Enter key:
```

Figure 1: Initial execution of the binary.

This provides us with some basic information about the binary: We must solve 666 challenges. The first challenge (and probably the remaining 665) requires us to enter a key. In addition, the program refers to itself as the “ever changing magic mushroom” which hints at the fact the binary will modify itself in some way.

If we enter a random key, we receive an error message as can be seen in Figure 2.

```
magic$ ./magic
Welcome to the ever changing magic mushroom!
666 trials lie ahead of you!
Challenge 1/666. Enter key: FLARE!
No soup for you!
magic$
```

Figure 2: Error message that appears if an incorrect key is entered.

With this in mind, we take a closer look at the strings in the program using the ***strings*** Linux utility:

```
No soup for you!
Failed to read file
Could not write file!
Run, Forrest, run!!
Generated first permutation!
Welcome to the ever changing magic mushroom!
%d trials lie ahead of you!
Challenge %d/%d. Enter key:
Congrats! Here is your price:
```

Figure 3: The interesting strings in the binary.

Besides the strings that were printed when we executed the program, there are a few other strings that seem interesting. First, there are some errors about reading and writing a file, which indicates that the binary performs some file operations.

Second, there is the string “Run, Forrest, run!!”, which seems out of place. In contrast to the other strings, this string does not seem to be a message that is intended for the user. We should take a closer look in IDA where and how this string is used.

Third, there is a message that states: “Generated first permutation!”. This is a further hint at the fact that the binary may modify itself or at least will permutate something. Given that we have 666 challenges to solve, a possible target for the permutation might be the key.

Finally, there is the string “Congrats! Here is your price:” which probably is related to printing the flag. We should take a closer look at the references to this string in IDA as well.

Static Analysis in IDA

Having conducted our initial analysis, we open the binary in IDA. We first take a look at the main function of the

program (0x4038D4). In the main function we find both of the strings that we identified during our initial analysis. The first string, “Run, Forrest, run!!” is used to seed ***srand***:

```
.text:0000000000403A08
.text:0000000000403A08 loc_403A08:                             ; CODE XREF: main+16B↓j
.text:0000000000403A08                 mov     edx, [rbp+dwRunForrestCounter]
.text:0000000000403A0E                 mov     rax, [rbp+szRunForrest] ; "Run, Forrest, run!!"
.text:0000000000403A15                 add     rax, rdx
.text:0000000000403A18                 mov     eax, [rax]
.text:0000000000403A1A                 xor     [rbp+seed], eax ; [rbp+seed] is initially set to zero.
.text:0000000000403A20                 add     [rbp+dwRunForrestCounter], 4
.text:0000000000403A27
.text:0000000000403A27 loc_403A27:                             ; CODE XREF: main+132↑j
.text:0000000000403A27                 mov     ebx, [rbp+dwRunForrestCounter]
.text:0000000000403A2D                 mov     rax, [rbp+szRunForrest]
.text:0000000000403A34                 mov     rdi, rax        ; s
.text:0000000000403A37                 call    _strlen
.text:0000000000403A3C                 cmp     rbx, rax
.text:0000000000403A3F                 jb      short loc_403A08
.text:0000000000403A41                 mov     eax, [rbp+seed]
.text:0000000000403A47                 mov     edi, eax        ; seed
.text:0000000000403A49                 call    _srand
```

Figure 4: Seeding srand.

At the beginning of the main function, a pointer to the string “Run, Forrest, run!!” is moved into the variable ***szRunForrest*** (0x403911). In the loop show in Figure 4, the string is processed in blocks of four bytes. Each block is XORed into the variable **seed** (0x403A1A), which is initially set to zero (0x40391C). Later on, **seed** is passed to ***srand*** (0x403A49) and thus used to seed the pseudo-random number generator.

This information will allow us to predict the output of the pseudo-random number generator and therefore the result of calls to ***rand***. While we do not need this information right now, it will become relevant at a later point during our analysis.

Moving on, we find the main loop of the program (0x403BD3):

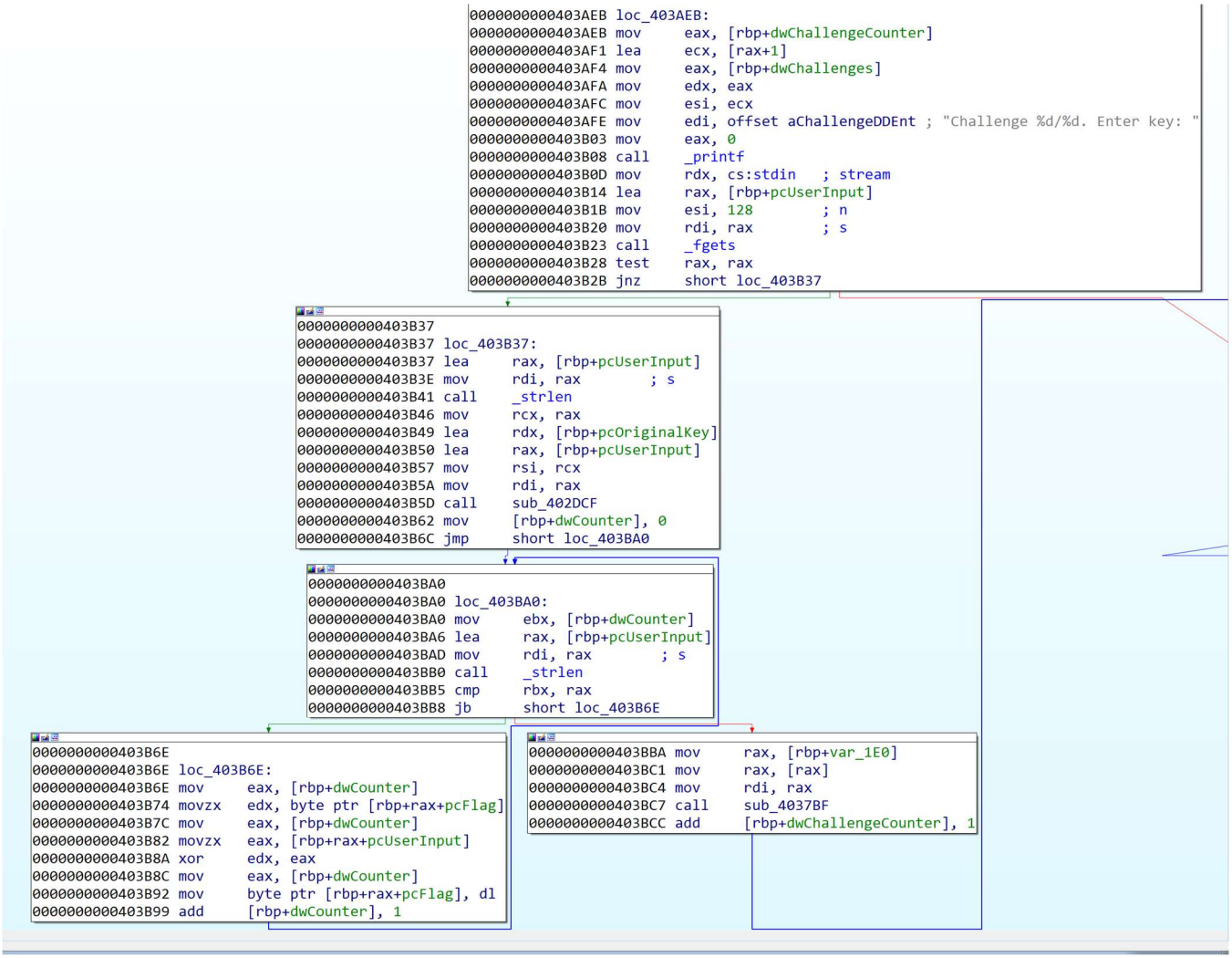

Figure 5: The main loop of the binary.

The main loop first prints the number of the current challenge and asks us for the key (0x403AEB-0x403B08). The key is then read from stdin using ***fgets*** and is at most 128 bytes long (0x403B23).

Next, function ***sub_402DCF*** is invoked. It receives the user input, a stack buffer, and the length of the user input as argument (0x403B5D).

After the function call follows another loop @ 0x403BA0. This loop XORs each byte of the user input with the

corresponding byte of the stack string ***pcFlag*** (pcFlag[0] ^= user_input[0], pcFlag[1] ^= user_input[1], etc.). This stack string is actually the encrypted flag as we can see by taking a look at the ***printf*** call that directly follows the main loop and is show in Figure 6.

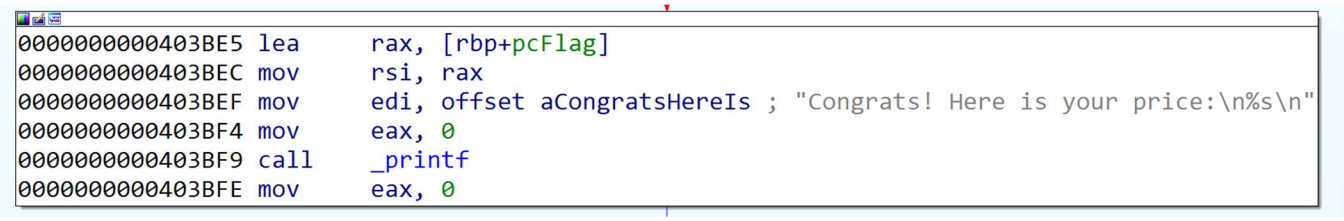

Figure 6: The flag is printed.

Finally, the main loop invokes the function ***sub_4037BF*** (0x403BC7), increases the challenge counter, and jumps to the top.

This part of the binary gives some more information about the challenge: The flag is stored on the stack. It is XORed with the 666 keys that we have to enter. Once we enter all keys, the flag will be printed in plaintext.

However, the questions that remain are where the user input is verified and how the current challenge key is generated. During our analysis of the main loop we encountered two functions calls that we have not consider yet: ***sub_402DCF*** and ***sub_4037BF***. We continue by analyzing each of these functions.

sub_402DCF is the function that validates the user input. The quickest way to see that is to look at the function @ 0x402CC7 that is invoked at multiple locations during the execution of the validation function. As can be seen in Figure 7, the function will print “No soup for you!” and call ***exit***. This is the error message that we saw when we executed the program and entered the incorrect key.

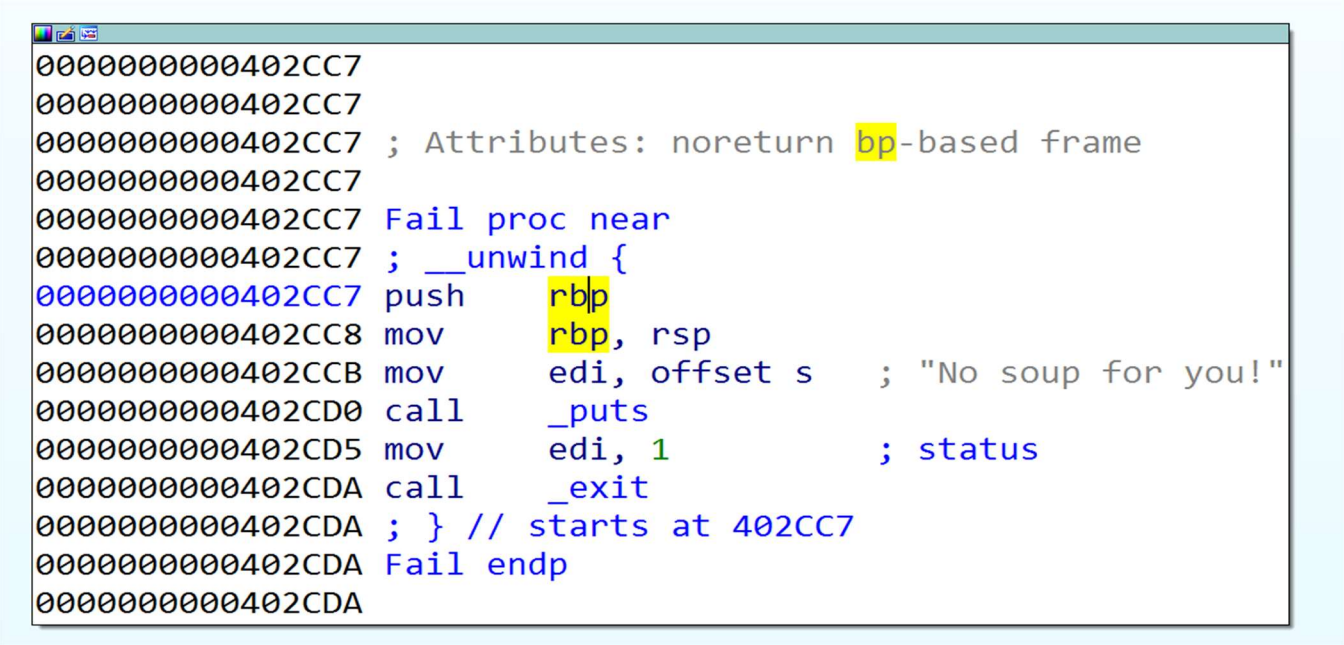

Figure 7: The incorrect key was entered.

In the validation function, we see a lot of references to the memory location 0x605100. Understanding what is stored at this location is one of the most important aspects of the challenge. By reverse engineering the function, we can deduce that the memory location contains an array of structures that have the following layout:

```
typedef int (*comparison_func)(char *input, unsigned int len, char *target);

struct magic_entry {

    comparison_func func;             // Pointer to the encrypted comparison
function

    unsigned int func_sz;               // The size of the comparison function
in bytes

    unsigned int input_offset;         // The part of the input that the
comparison function will process

    unsigned int input_len;             // The number of characters that the
comparison function will

                                                          // consider starting
from input[input_offset]

    unsigned int output_offset;       // The offset in the output that the input
bytes correspond to
```

```
     char *key;                                    // The encryption key for the
comparison function
     char target[256];                           // The values that the comparison
function will compare the
                                                                  // transformed input
with
}
magic_entry magic_table[21];
```

This leads to the following pseudo code for the validation function:

```
void sub_402DCF(char *szUserInput, unsigned long dwUserInputLen, char *pcOut)
{
  unsigned int i;

  for ( i = 0; i < 0x21; ++i )
  {
    if (magic_table[i].input_offset + magic_table[input_len] > dwUserInputLen )
      Fail();
    Encrypt_Decrypt(magic_table[i].func, magic_table[i].func_sz,
magic_table[i].key);
    if (!magic_table[i].func(magic_table[i].input_offset + szUserInput,
magic_table[i].input_len, magic_table[i].target)
    {
      Encrypt_Decrypt(magic_table[i].func, magic_table[i].func_sz,
magic_table[i].key);
```

```
      Fail();
    }
    Encrypt_Decrypt(magic_table[i].func, magic_table[i].func_sz,
magic_table[i].key);
    memcpy(magic_table[i].output_offset + pcOut), (magic_table[i].input_offset +
szUserInput), magic_table[i].input_len);
  }
}
```

The validation function validates the user input in a loop. In each iteration of the validation loop, a different part of the entered key is validated. For this purpose, the validation function decrypts a comparison function and invokes it. Each comparison function receives the part of the input that it should verify, the number of characters to verify, and a pointer to the solution for this function. The solution will be used to verify the input and depends on the comparison function.

The comparison functions are XOR encrypted. The same function is used for encryption and decryption. It is located @ 0x402CDF and shown in Figure 8. The encryption function receives a target pointer, the length of the target, and a pointer to a key as parameters. The function then simply XORs each byte of the target with the corresponding byte in the key (target[0] ^= key[0], target[1] ^= key[1], etc.).

Each comparison function either returns ***true*** or ***false***. If the comparison function returns ***false***, the error message is printed, and the program exits. Otherwise validation continues. We will take a closer look at each of the validation functions at a later point in the analysis.

In the last step of the validation loop, the function invokes ***memcpy*** to copy the validated part of the input to ***pcOut+magic_table[i].output_offset***. This will effectively restore the original key from the permutated key. The original key will be written to ***pcOut***. This means we are able to see the original key once we solve a single of the 666 challenges.

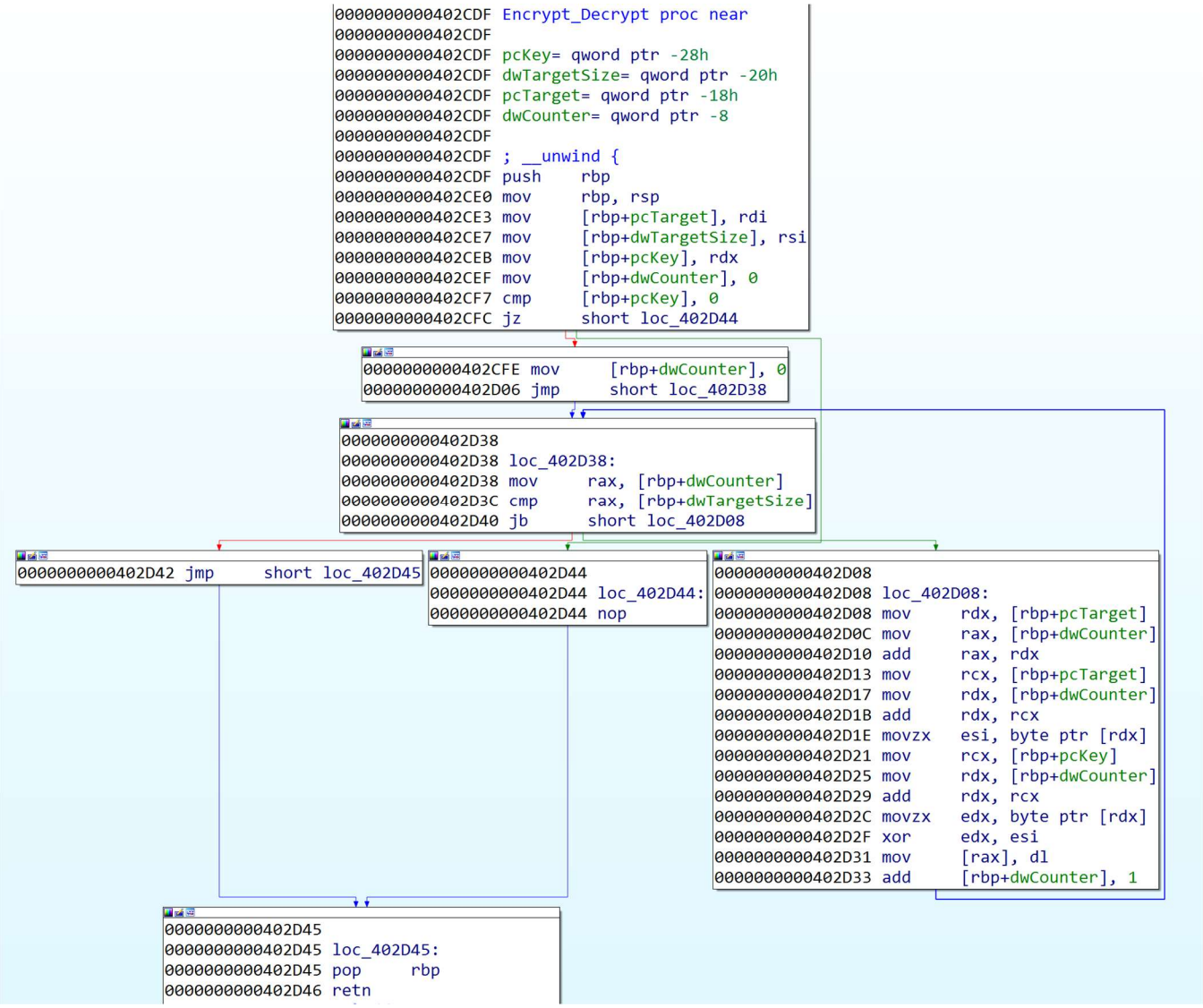

Figure 8: Encryption/Decryption function.

This leaves the question how the permutations of the key are generated. For this purpose, we will take a look at function ***sub_4037BF***. The pseudo code for the function is show below:

```
void sub_4037BF(char *path)
```

```
{
    char *input_file;
    size_t input_file_len;
    char *cmp_functions;
    void *first_func = NULL;
    size_t all_func_sz = 0;
    magic_entry *_magic_table;

    // Get File
    read_file(path, &input_file, &input_file_len);

    // Find comparsion functions
    first_func = magic_table[find_first_function(magic_table)].func;
    cmp_functions = find_cmp_functions(input_file);
    _magic_table = (magic_entry *)find_magic_table(input_file);

    // Replace magic table in the input file
    memcpy(_magic_table, magic_table, sizeof(magic_table));

    // Permutate
    all_func_sz = permutate(cmp_functions, _magic_table, first_func);

    // Replace in memory
```

```
    memcpy(first_func, cmp_functions, all_func_sz);
    memcpy(magic_table, _magic_table, sizeof(magic_table));


    // Replace file
    replace_file(path, input_file, input_file_len);
}
```

The function sub_4037BF – from here on referred to as update function - will first read in the magic binary. Next, it searches for the ***magic_table*** and the comparisons function within the binary as the virtual address of the table and the functions is different from their address within the file. The function then replaces the magic table in the file, with the magic table that is currently in memory. Notice that this change happens to the file in memory not to the file on disk.

In the next step, the update function permutates the magic table and the comparison functions by invoking the permutate function. We will take a closer look at the permutate function after we completed the analysis of the update function.

Finally, the update function replaces the comparison functions and the magic table in memory and writes the modified file to disk. At this point, the binary modified itself on disk and in memory. Both versions use the same magic table and the same comparison functions.

The permutation function is located @ 0x40332D. The pseudo code of the function is shown below:

```
size_t permutate(char *functions, magic_entry *table, void *first_func)
{
    magic_entry *_magic_table = table;
    unsigned int index;
    unsigned int i;
    size_t total = 0;
```

```
    size_t tmp = 0;
    unsigned int new_key;

    // Permutate functions
    for (i = 0; i < magic_table_entries; i++) {
        // Get random   index
        index = rand();
        index %= (magic_table_entries-i);
        index += i;

        // Update encryption key
        Encrypt_Decrypt(_magic_table[index].func, _magic_table[index].func_sz,
_magic_table[index].key);
        new_key = rand();
        new_key %= (sizeof(rbytes) - _magic_table[index].func_sz + 1);
        _magic_table[index].key = &rbytes[new_key];
        Encrypt_Decrypt(_magic_table[index].func, _magic_table[index].func_sz,
_magic_table[index].key);

        // Move current function to a different location
        memcpy(functions, _magic_table[index].func, _magic_table[index].func_sz);

        // Update pointer
```

```
        _magic_table[index].func = (check_func)first_func;
        functions += _magic_table[index].func_sz;
        first_func += _magic_table[index].func_sz;
        total += _magic_table[index].func_sz;



        // Swap input offset. This effectively permutates the key.
        _magic_table[index].input_offset = tmp;
        tmp += _magic_table[index].input_len;
        swap_entry(i, index, _magic_table);
    }

    // Permutate the magic table.
    for (i = 0; i < magic_table_entries; i++) {
        // Get random
        index = rand();
        index %= (magic_table_entries-i);
        index += i;

        // Swap
        swap_entry(i, index, _magic_table);
    }
```

```
    return total;

}
```

The permutate function operates in two steps. First, it will permutate the order of the comparison functions, updating the encryption key for each function in the process. Next, it changes the order of the entries in the magic table.

To permutate the comparison functions, the permutation function begins by selecting a random comparison function in the magic table. The index of the function is computed with the help of the ***rand*** function. It then decrypts the function and selects a new encryption key. This is achieved by calculating another random index into an array of random bytes (***rbytes***). This index is once more calculated based on the output of the ***rand*** function.

To change the encryption key of the function, the key pointer of the function within the magic table is updated to point to the new key. Afterwards the function is encrypted using the new key.

Having updated the encryption of the comparison function, the permutation function copies the function into the next free spot in the array of comparison functions that is pointed to by ***functions***. Next, the permutation function updates the necessary pointers. For instance, it updates the ***functions*** pointer to point to the next free spot within the comparison functions array. Finally, the permutation function changes the input offset of the current function to its new position within the comparison function array, swaps the entry of the function in the magic table with the current index ***i***, and continues to process the remaining functions.

Once all comparison functions have been reordered, the permutation function follows the same process to reorganize the magic table. In particular, it selects a random array index based on a call to rand. Next, it swaps the selected entry with entry ***i*** in the table (this is essentially the topmost entry in the magic table that has not been swapped yet). Finally, it updates ***i*** and continues with the remaining entries.

The most important part about the permutation process is the permutation of the key. Once we have the first key, understanding the permutation of the key will allow us to predict the next permutation and thus to solve the remaining 665 challenges. The permutation of the key occurs based on the new index of a comparison function within the magic table. We can predict the index since we know the seed to ***srand*** and we know how many ***rand*** calls occur within each call to the permutation function. This will allow us to predict the next permutations of the key as we will see later on.

The comparison functions

To get the first key of the challenge, we have to take a closer look at the comparison functions. There are various approaches that we can use to do so. For instance, we could write a python script that will decrypt the functions (the encryption key is given in the magic table) and analyze them in IDA. Or we can use dynamic analysis and let the binary decrypt the functions for us.

We follow the latter approach in this analysis. For this purpose, we will run the binary in GDB and place a breakpoint at the ***call rcx*** instruction @ 0x402F06. This instruction invokes all the comparison functions. When the comparison functions are invoked, they will be unencrypted in memory and we can analyze them. Figure 9 shows the process in GDB for the first comparison function.

```
magic$ gdb ./magic
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./magic...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) b *0x402F06
Breakpoint 1 at 0x402f06
(gdb) run
Starting program: /tmp/magic
Welcome to the ever changing magic mushroom!
666 trials lie ahead of you!
Challenge 1/666. Enter key: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x0000000000402f06 in ?? ()
(gdb) x/2i $rip
=> 0x402f06:    call   rcx
   0x402f08:    test   eax,eax
(gdb) x/10i $rcx
   0x400c55:    push   rbp
   0x400c56:    mov    rbp,rsp
   0x400c59:    mov    QWORD PTR [rbp-0x48],rdi
   0x400c5d:    mov    DWORD PTR [rbp-0x4c],esi
   0x400c60:    mov    QWORD PTR [rbp-0x58],rdx
   0x400c64:    mov    DWORD PTR [rbp-0x4],0x0
   0x400c6b:    jmp    0x400d89
   0x400c70:    mov    eax,DWORD PTR [rbp-0x4]
   0x400c73:    lea    rdx,[rax*8+0x0]
   0x400c7b:    mov    rax,QWORD PTR [rbp-0x58]
(gdb)
```

Figure 9: Placing a breakpoint at the invocation of the comparison functions.

To analyze the remaining functions, we have to make sure that we enter a long enough key. In particular, the key that we enter must be at least as long as the key that the binary expects. Otherwise the validation function will exit

without invoking all comparison functions.

In addition, we need to make sure that the comparison functions return true. To achieve this, we place a second breakpoint @ 0x402F08 directly after the ***call rcx*** instruction. When we continue execution using the “c” command, this breakpoint will be hit. Now, we can simply change the value of the rax register to pretend that we entered the correct input, which will lead us to the next comparison function. This process is shown in Figure 10.

```
   0x400c99:    add    rdx,rax
   0x400c9c:    mov    ecx,DWORD PTR [rbp-0x4]
   0x400c9f:    mov    rax,QWORD PTR [rbp-0x48]
(gdb) b *0x402f08
Breakpoint 2 at 0x402f08
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000402f08 in ?? ()
(gdb) i r
rax            0x0      0
rbx            0x14     20
rcx            0x0      0
rdx            0x19438b44a658   27777890035288
rsi            0x3      3
rdi            0x7fffffffdd42   140737488346434
rbp            0x7fffffffdcf0   0x7fffffffdcf0
rsp            0x7fffffffdcc0   0x7fffffffdcc0
r8             0x6186ab 6391467
r9             0x7ffff7fd5500   140737353962752
r10            0x618010 6389776
r11            0x246    582
r12            0x400ad0 4197072
r13            0x7fffffffdfc0   140737488347072
r14            0x0      0
r15            0x0      0
rip            0x402f08 0x402f08
eflags         0x287    [ CF PF SF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) set $rax = 1
(gdb) print $rax
$1 = 1
(gdb) c
Continuing.

Breakpoint 1, 0x0000000000402f06 in ?? ()
(gdb)
```

Figure 10: Modifying the return value of the comparison functions.

This approach will allow us to analyze all comparison functions and to determine the part of the input that the

function validates. As a reminder, comparison functions receive three parameters: The user input to validate, the number of bytes of the input to validate, and the solution to validate against. By reverse engineering a comparison function and using the solution that is passed to the function, we can thus predict which input the function expects. This will give us a part of the key. By repeating this process for all functions, we will be able to obtain the first key of the challenge.

There are seven different comparison functions. In the following, we will analyze each of the comparison functions as they appear in the binary.

Fibonacci

The first comparison function that we encounter calculates the Fibonacci number for the decimal representation of each character. For example, if the input character would be 'A', it would calculate the 65th Fibonacci number and compare it against the provided solution. If the solution matches it will perform the same operation for the following characters until all characters have been verified. The pseudo code for the function is given below:

```
static __attribute__((always_inline)) unsigned long fib(unsigned char n)
{
    unsigned long result = 0;
    unsigned long n1 = 1;
    unsigned long n2 = 0;

    while (n > 0) {
        result = n1 + n2;
        n2 = n1;
        n1 = result;
        n--;
    }
```

```
    return result;
}


int fib_comp(unsigned char *input, unsigned int len, unsigned long *target)
{
    unsigned int i;


    for (i = 0; i < len; i++) {
        if (fib(input[i]) != target[i])
            return 0;
    }


    return 1;
}
```

One of the important things that we need to know about the function is that it uses the type ***unsigned long*** for the solution. This means the solutions are 8-byte numbers on a 64-bit system. However, Fibonacci numbers may get larger than 8-bytes. To find the correct characters, we thus have to compare the last 8-byte of the calculated Fibonacci number against the solution to get the correct character. We can use the following python code to get the character for a solution number:

```
def fib():
       cur = [0, 1]
       i = 0
       while True:
```

```
            if i == 0:
                 yield cur[0]
            elif i == 1:
                yield cur[1]
            else:
                tmp = cur[0] + cur[1]
                cur[0] = cur[1]
                cur[1] = tmp
                yield tmp
            i += 1


def find_fib(solution):
        for i, x in enumerate(fib()):
            if (x & solution) == solution:
                print(chr(i-1))
                return
```

As an example, let us get the first character that the function verifies. When the function is invoked, the pointer to the input characters is stored in rdi (1st parameter). For the first execution of the function the first character that is verified is the third character of the input. The function will verify three characters (2nd parameter stored in rsi). The solution that is expected for the first character is 0x12062f76909038c5 (pointed to by rdx, which contains the 3rd parameter to the function). Using the script above we find that the third character of the input should be the character 'd'.

We can use the same approach to get the remaining characters and in every other case where the same function is used. The Fibonacci function is used as comparison function in the rounds 1, 2, 6, 7, 12, 29, and 31 of the initial version of magic.

CRC32

The CRC32 comparison function is first used in round 3 of the initial magic binary. The function calculates the CRC32 of all input characters and compares it against the solution. The following pseudo code shows the operations of the function:

```
int crc32_comp(unsigned char *input, unsigned int len, unsigned char *target)
{
    int i, j;
    unsigned int byte, crc, mask;
    char first = 0;

    i = 0;
    crc = 0xFFFFFFFF;
    while (i < len) {
        byte = input[i];
        crc = crc ^ byte;
        for (j = 7; j >= 0; j--) {
            mask = -(crc & 1);
            crc = (crc >> 1) ^ (0xEDB88320 & mask);
        }
        i = i + 1;
    }
```

```
    if (*(unsigned int*)target != ~crc)
        return 0;

    return 1;
}
```

We can use the following python script to brute force the characters that the function expects:

```
import itertools
import string
def find_crc32(solution, chars):
         for e in itertools.product(string.printable, repeat=chars):
             if binascii.crc32(("".join(e)).encode("ascii")) == solution:
                 print(e)
                 return
```

In round 3 the function only calculates the CRC32 for the input character 17 (rdi = pointer to input, rsi = number of characters). The solution for the 17th character is given in rdx. Using the script above we receive '.' for this character.

We can use the same approach for all other occurrences of the function. The function is used in round 3 and 16 of the initial magic binary.

RC4

The RC4 comparison function is first used in round 4. The function encrypts the input characters using RC4 and compares the result against the solution. The encryption key is a contained in the function and is "Tis but a scratch.". The pseudo code for the function is show below:

```
#define swap_bytes(a, b) \
    { \
```

```
        *(a) ^= *(b); \
        *(b) ^= *(a); \
        *(a) ^= *(b); \
    }

#define N 256

int rc4_comp(unsigned char *input, unsigned int len, unsigned char *target)
{
    unsigned int i;
    unsigned char j;
    unsigned char perm[N];
    unsigned char index1;
    unsigned char index2;
    unsigned char key[] = "Tis but a scratch.";
    unsigned int keylen = 18;
    char first = 0;

    for (i = 0; i < N; i++)
        perm[i] = i;

    index1 = 0;
    index2 = 0;
```

```
    for (i = 0; i < N; i++) {
        j = (j + perm[i] + key[i % keylen]) % N;

        swap_bytes(&perm[i], &perm[j]);
    }

    for (i = 0; i < len; i++) {
        index1 = (index1 + 1) % N;
        index2 = (index2 + perm[index1]) % N;

        swap_bytes(&perm[index1], &perm[index2]);

        j = (perm[index1] + perm[index2]) % N;

       if (target[i] != (input[i] ^ perm[j])) {
            return 0;
        }
    }

    return 1;
}
```

An important observation that we can make about this function is that it contains a bug. The variable "***j***" should be

initialized to zero before the second for loop. In the code above the variable j will be used uninitialized. As a result, the RC4 algorithm used here will behave differently than the original RC4 algorithm and the result of the encryption will depend on the uninitialized value of ***j***. However, since ***j*** is used in a modulo calculation, the encryption will produce at most 256 different results for the same key. This allows us to brute force the possible inputs for ***j*** and to select the most likely decryption value. We can use the following python script for this purpose:

```
import string

def swap(l, a, b):

        tmp = l[a]

        l[a] = l[b]

        l[b] = tmp


def rc4_mod(solution, j=0):

        key = "Tis but a scratch."

        perm = []

        for i in range(256):

            perm.append(i)

        for i in range(256):

             j = (j + perm[i] + ord(key[i % len(key)])) % 256

            swap(perm, i, j)

        index1 = 0

        index2 = 0

        result = ""

        for i in range(len(solution)):

            index1 = (index1 + 1) % 256
```

```
            index2 = (index2 + perm[index1]) % 256
            swap(perm, index1, index2)
            tmp = (perm[index1] + perm[index2]) % 256
            result += chr(ord(solution[i]) ^ perm[tmp])
        return result


def possible_chars(solution):
         for i in range(256):
         x = rc4_mod(solution, j=i)
         if all(c in string.printable for c in x):
             print("{0}: {1}".format(i, x))
```

For round 4, the solution that the function receives is “\xad\x2d\x84” (pointed to by rdx in little endian byte order). The function processes the 8th input character (rdi) and the length of the input are three characters (rsi). Running the script above we receive the following outputs:

```
0: ng
2: <{@
18: *~i
48: A;7
54: Wf0
56: <1:
61: '2]
73: >[F
87: kf9
```

```
96: }rc

148: jEr

159: X4Y

176: S(/

199: syQ

200: 2Ka

238: %)!

253: !d|
```

The most likely output seems to be the first (“ng “). We can verify our choice by passing the recovered key to magic and checking whether it accepts it, once we determined all characters. We can try one of the other solutions if magic should not accept the key.

Note that we are probably able to predict the value of ***j***. The value will depend on what the previous function stored in that particular memory location. The previous comparison function (CRC32) stored the value zero at this location. Thus, j would be zero in this case. In fact, for both of the times that the RC4 function is used in the initial version of magic ***j*** will be zero.

In addition, if you are using a brute force approach to solve the challenge, you will probably also have to address the bug in this function. Otherwise, it is very like the approach will fail. We discuss a possible brute force approach to the challenge at the very end of this document.

The RC4 function is used in round 4 and 20 of the initial magic binary.

B64

The comparison function used in round 5 is using base64 with a custom alphabet to validate the input. The function will convert the input characters to base64 and compare them against the provided solution. The pseudo code for the function is given below:

```
int b64_comp(unsigned char *input, unsigned int len, unsigned char *target)

{
```

```
    unsigned char alpha[] = "*9_d\xc7\xa2F#SktG(MpBI%Rjb8@JiEDY-
1$PgyT!Lvqf+chmQWO0eNZ4un3l7H&2wazKV";
    unsigned int i, j;
    unsigned char tmp;
    unsigned char cur;
    unsigned char in;

    for (i = 0, j = 0; i <= len; i++) {
        in = i < len ? input[i] : '\0';

        if (i % 3 == 0 && i < len) {
            cur = in >> 2;
            tmp = in & 0x3;

            if(target[j++] != alpha[cur])
                return 0;
        }
        else if (i % 3 == 1) {
            cur = (tmp << 4) | in >> 4;
            tmp = in & 0xf;

            if(target[j++] != alpha[cur])
                return 0;
```

```
        }
        else if (i % 3 == 2) {
            cur = tmp << 2 | in >> 6;

            if(target[j++] != alpha[cur])
                return 0;

            if (i < len) {
                cur = (in & 0x3f);
                if(target[j++] != alpha[cur])
                   return 0;
            }
            tmp = 0;
        }
    }

    return 1;
}
```

We can use the following python function to get the input characters:

```
def b64_dec(solution):
         a = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
         b = "*9_d\xc2\xa7F#SktG(MpBI%Rjb8@JiEDY-
1$PgyT!Lvqf+chmQWO0eNZ4un3l7H&2wazK"
```

```
        s = solution.translate(str.maketrans(b, a))


        while True:
            try:
                x = base64.b64decode(s)
                return x.decode("ascii")
            except:
                s += "="
```

For round 5, we receive the character ‘ ‘ (space) for the solution “S*” pointed to by rdx. This is the 64th input character (rdi) and we verify a single character (rsi).

The same function is also used in round 23 of the initial magic binary.

ROT13

The comparison function ROT13 is first used in round 8 of the initial magic binary. The function adds 13 to every input character and compares the result to the solution. The pseudo code of the function is shown below:

```
int rot13_comp(unsigned char *input, unsigned int len, unsigned char *target)
{
    unsigned int i = 0;

    for (i = 0; i < len; i++) {
        if ((input[i] + 13) != target[i]) {
            return 0;
        }
    }
```

```
    return 1;
}
```

We can use the following python function to obtain the original characters:

```
def rot13(solution):
        return "".join([chr(ord(x) - 13) for x in solution])
```

For round 8, we are looking for three characters (rsi). The function validates input offset 31 (rdi). Using the script above we receive “the” for the solution “\x81ur” pointed to by rdi.

ROT13 is also used in the rounds 11, 13, 19, 24, and 32 of the original magic binary.

CMP

This is the simplest function of all comparison functions. The CMP function simply compares the input with the solution. Thus, the solution characters are the input characters we are looking for. The function appears first in round 9 of the original magic binary and its pseudo code is given below:

```
int cmp_comp(unsigned char *input, unsigned int len, unsigned char *target)
{
    unsigned int i;

    for (i = 0; i < len; i++) {
        if (input[i] != target[i])
            return 0;
    }

    return 1;
```

```
}
```

For round 9 the solution character pointed to by rdx is “ “ (space). This is the value for the 11th input character (rdi) and the function only validates a single character (rsi).

The CMP function is used in round 9, 10, 15, 17, 25, and 26 of the original magic binary.

XOR

The last comparison function is XOR. It XORs every input character with the number 42 and compares it against the solution character. The function first appears in round 14 of the initial magic binary. The pseudo code of the function is shown below:

```
int xor_comp(unsigned char *input, unsigned int len, unsigned char *target)
{
    unsigned int i;

    for (i = 0; i < len; i++) {
        if ((input[i] ^ 42) != target[i])
            return 0;
    }

    return 1;
```

We can use the following python function to decode the solution characters:

```
def xor42(solution):
        return "".join([chr(ord(x) ^ 42) for x in solution])
```

In round 14 the function uses the solution “O\n” (rdx) to validate two characters (rsi) at input offset 19 (rdi). Using the script above we receive the characters “e “.

The XOR comparison function is used in round 14, 18, 21, 22, 27, 28, 30, and 33 of the initial magic binary.

The key for the first challenge

Using the approaches described in the last section, we finally receive the key for the first challenge:

inds isng llg w. e HthitheoftheAh,urnolik inefe yo blrhot in owace

A good way to identify functions during this process is to look at the size of the functions in the magic table. Functions with the same size are the same functions with different solutions and different encryption keys. Based on the information in the last section, we could of course also automate the process of obtaining the first key.

If we enter the key and place a breakpoint directly after the validation function (0x403B62), we can print the variable ***pcOriginalKey*** ($rbp – 0x120) to see the original key as shown in Figure 11.

```
magic$ gdb ./magic
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./magic...(no debugging symbols found)...done.
(gdb) b *0x403B62
Breakpoint 1 at 0x403b62
(gdb) run
Starting program: /tmp/magic
Welcome to the ever changing magic mushroom!
666 trials lie ahead of you!
Challenge 1/666. Enter key: inds isng  llg w. e  HthitheoftheAh,urnolik inefe  yo blrhot in
 owace

Breakpoint 1, 0x0000000000403b62 in ?? ()
(gdb) x/s $rbp - 0x120
0x7fffffffddc0: "Ah, there is nothing like the hot winds of Hell blowing in your face."
(gdb)
```

Figure 11: The original key of the challenge.

Getting the flag

The last task that remains is to solve the remaining 665 challenges. To achieve this, we will predict the remaining 665 keys based on the first key that we already know and the permutation function that we reverse engineered.

To predict the keys, we essentially have to rebuild the permutation function and use the same seed for ***srand*** as the magic binary uses. This leads to the following C code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>
```

```
#include <linux/random.h>
#include <errno.h>

typedef struct {
    void *func;
    unsigned int func_sz;
    unsigned int input_offset;
    unsigned int input_len;
    unsigned int output_offset;
    char *key;
    char target[256];
} magic_entry;

// Magic table
magic_entry magic_table[0x21];

void err(char *msg)
{
      perror(msg);
      exit(EXIT_FAILURE);
}

// Read the magic table from the magic binary
```

```
void get_magic_table(char *path)
{
	FILE *fp;
	size_t read;

	if ((fp = fopen(path, "r")) == NULL)
		err(NULL);

	fseek(fp, 0x5100, SEEK_SET);
	read = fread(magic_table, sizeof(magic_entry), 0x21, fp);

	if (read != 0x21)
		err("Could not read magic table");
}

// Seed srand the same way that magic seeds it.
void seed()
{
	const char *seed_key = "Run, Forrest, run!!";
	unsigned int seed, i;

	seed = 0;
	for (i = 0; i < strlen(seed_key); i += 4) {
```

```
        seed ^= *(unsigned int *)(seed_key + i);
    }

    srand(seed);
}

// Predict the remaining keys
void predict(char *key)
{
      unsigned int i, j, index, offset, len, swap;
      char last[128];
      char cur[128];
      char *tmp;
      magic_entry entry;

      memset(last, 0, sizeof(last));
      memset(cur, 0, sizeof(cur));
      strcpy(last, key);

      printf("%s\n", key);

      for (i = 0; i < 665; i++) {
            tmp = cur;
```

```
            for (j = 0; j < 0x21; j++) {
                  index = rand();
                  index %= 0x21 - j;
                  index += j;

                  // Update the input offset of the entry in the magic table just
as the original binary
                  offset = magic_table[index].input_offset;
                  len = magic_table[index].input_len;
                  magic_table[index].input_offset = tmp - cur;

                  // Update our current copy of the key to get the next key
                  memmove(tmp, &last[offset], len);
                  tmp += len;

                  // Update the magic table in the same way the magic binary
operates.
                                          // This requires us to swap the current
entry (i) with the calculated entry (index)
                  memcpy(&entry, &magic_table[j], sizeof(entry));
                  memcpy(&magic_table[j], &magic_table[index], sizeof(entry));
                  memcpy(&magic_table[index], &entry, sizeof(entry));
```

```
                // The function calls rand once to update the encryption key. We
do not care about
                                        // the encryption, but need to keep the
pseudo-random number generator in sync.
                rand();
        }

                        // We also need to perform the permutation of the magic
table.
        for (j = 0; j < 0x21; j++) {
                index = rand();
                index %= 0x21 - j;
                index += j;

                // Swap
                memcpy(&entry, &magic_table[j], sizeof(entry));
                memcpy(&magic_table[j], &magic_table[index], sizeof(entry));
                memcpy(&magic_table[index], &entry, sizeof(entry));
        }

                        // Print the current key
        printf("%s\n", cur);
```

```
                        // Update the last key
            memcpy(last, cur, sizeof(cur));
      }
}

int main(int argc, char *argv[])
{
      char *first_key = "inds isng  llg w. e  HthitheoftheAh,urnolik inefe  yo
blrhot in owace";

      get_magic_table("magic");
      seed();
      predict(first_key);
}
```

The program will print all 666 keys. We can save this output to a file and feed it to the magic binary to get the flag:

```
magic$ ./predictor > solutions.txt
magic$ ./magic < solutions.txt | tail -1
mag!iC_mUshr00ms_maY_h4ve_g!ven_uS_Santa_ClaUs@flare-on.com
magic$
```

Figure 12:Getting the flag.

Alternative Solution: Brute-force

Each of the comparison functions that magic uses only validates 1-3 characters. This makes the binary

vulnerable to brute-force attacks. To show this, we will write a GDB script that will brute-force the individual keys and obtain the flag. The script will operate as follows:

1. It will generate 666 random inputs that it will pass to magic. This ensures that the binary will run through all challenges.
2. It will place a breakpoint on the call rcx instruction @ 0x402F06 directly before the comparison functions are invoked. When the breakpoint is invoked during execution, the script will:
   a) Generate all possible values for the number of characters that the comparison function expects. The number of character is stored in rsi.
   b) Store the current register values such that we can restore the CPU state in case the current input is not the expected input.
   c) Overwrite the current input to the function pointed to by rdi with the current character sequence.
3. It will place another breakpoint directly after the call rcx instruction @ 0x402F08. When the breakpoint is invoked the script will:
   a) Check the value of the rax register. If it is zero, we passed an incorrect character sequence. In this case, we will continue with step b). Otherwise we found the solution for the current comparison functions and continue execution.
   b) If we provided an incorrect input, we will restore the CPU state using the registers that we saved in step 2). In addition, we will overwrite the stack area below the current function (the stack area that is used by the comparison functions) with zeros to account for the bug in the RC4 function. Finally, we continue execution which will invoke breakpoint 2) and we will try the next character sequence.

The following python GDB script can be used to brute-force magic. It contains a slight optimization and will store the solution and input characters for each comparison functions that it encounters. This

allows it to directly determine the input characters if we encounter a known solution. The optimization is based on the fact that each comparison function will always validate the same input characters. The script will solve the challenge about 1-2 hours.

```
#!/usr/bin/gdb -P

import gdb
import sys
import itertools

TRIALS = 666
BRUTE = None
DEBUG = True

def debug(s, level):
    global DEBUG

    if DEBUG:
        if level == 0:
            print("\n[!] {0}".format(s))
        elif level == 1:
            print("[+] {0}".format(s))
        elif level == 2:
            print("\t -> {0}".format(s))
        elif level == 3:
            sys.stdout.write("\r\t -> {0}".format(s))
        elif level == 4:
            sys.stdout.write("{0}".format(s))

def gen_input():
    """Generate some input."""
```

```
    result = []

    for i in range(0, TRIALS):
        result.append("A" * 100)

    return result

class Snapshot:
    """A poor mans snapshot of the CPU state."""
    def __init__(self):
        self._regs = ["rax", "rbx", "rcx", "rdx", "rsi", "rdi", "rbp",
                      "rsp", "r8", "r9", "r10", "r11", "r12", "r13",
                      "r14", "r15", "rip", "eflags"]

        self._snapshot = {}

    def save(self):
        self._snapshot.clear()

        for reg in self._regs:
            self._snapshot[reg] = int(gdb.parse_and_eval("${0}".format(reg)))

    def restore(self):
        for reg, value in self._snapshot.items():
            gdb.execute("set ${0} = 0x{1:x}".format(reg, value))

        # Reset stack
        # This is needed to work around the bug in the RC4 function
        for i in range(1, 4):
            gdb.execute("set *(unsigned long *)($rsp - {0}) = 0".format(i*8))
```

```
class BruteForcer:
    """Our reliable friend. Brute-forcing always works. At some point."""
    def __init__(self):
        self._chars = ("abcdefghijklmnopqrstuvwxyz"
                       " ,.!?:;"
                       "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
        self._reset = True
        self._log = {}
        self._key = None

    @property
    def last(self):
        return self._last

    def save(self):
        self._log[(self._key, self._length)] = self._last

    def start(self, length, key):
        self._length = int(length)
        self._reset = True
        self._key = key
        debug("Starting BruteForcer with length {0} ({1})".format(length,
                                                                  key), 1)

    def get(self):
        if self._reset:
            self._cur = itertools.product(self._chars, repeat=self._length)
            self._reset = False

        if (self._key, self._length) in self._log:
            rv = self._log[(self._key, self._length)]
```

```
            if self._last != rv:
                debug("Using saved value {0}...".format(rv), 2)
                self._last = rv
                return rv

        self._last = next(self._cur)
        return self._last

SUCCESS = True

class CallBreakpoint(gdb.Breakpoint):
    def stop(self):
        global SUCCESS
        global BRUTE
        global SNAPSHOT

        if SUCCESS:
            target = int(gdb.parse_and_eval("$rdx"))
            key = gdb.inferiors()[0].read_memory(target, 8).tobytes()
            BRUTE.start(gdb.parse_and_eval("$rsi"), key)
            SNAPSHOT.save()
            SUCCESS = False

        attempt = BRUTE.get()

        debug("Trying {0}...".format(attempt), 3)

        # Overwrite input
        for i in range(0, len(attempt)):
            gdb.execute("set *(char *)($rdi + {0}) = '{1}'".format(i,
```

```
                                                                attempt[i]))

        # Continue
        return False

class CheckBreakpoint(gdb.Breakpoint):
    def stop(self):
        global SUCCESS
        global SNAPSHOT
        global BRUTE

        eax = int(gdb.parse_and_eval("$rax"))
        debug(" -> ({0})".format(eax), 4)

        if eax == 1:
            debug("SUCCESS! ({0})".format(BRUTE.last), 0)
            SUCCESS = True
            BRUTE.save()
        else:
            #debug("FAIL", 0)
            SNAPSHOT.restore()
            gdb.execute("set $rip = 0x402f03")

        return False

def stop_handler(event):
    print("FAILED")

# Register stop Handler
gdb.events.stop.connect(stop_handler)
```

```
# Generate default input
inputs = gen_input()

# Write inputs to a file
with open("inputs", "w") as f:
    f.write("\n".join(inputs))

# Create helper objects
BRUTE = BruteForcer()
SNAPSHOT = Snapshot()

# GDB settings
gdb.execute("set disassembly-flavor intel")
gdb.execute("set python print-stack full")
gdb.execute("set pagination off")

# Place breakpoint on indirect call
CallBreakpoint("*0x402f06")

# Place a breakpoint after the call
CheckBreakpoint("*0x402f08")

# Run GDB
gdb.execute('run < inputs')
```

To start the script, we can use the following command line:

```
gdb -ex "source solver.py" ./magic
```