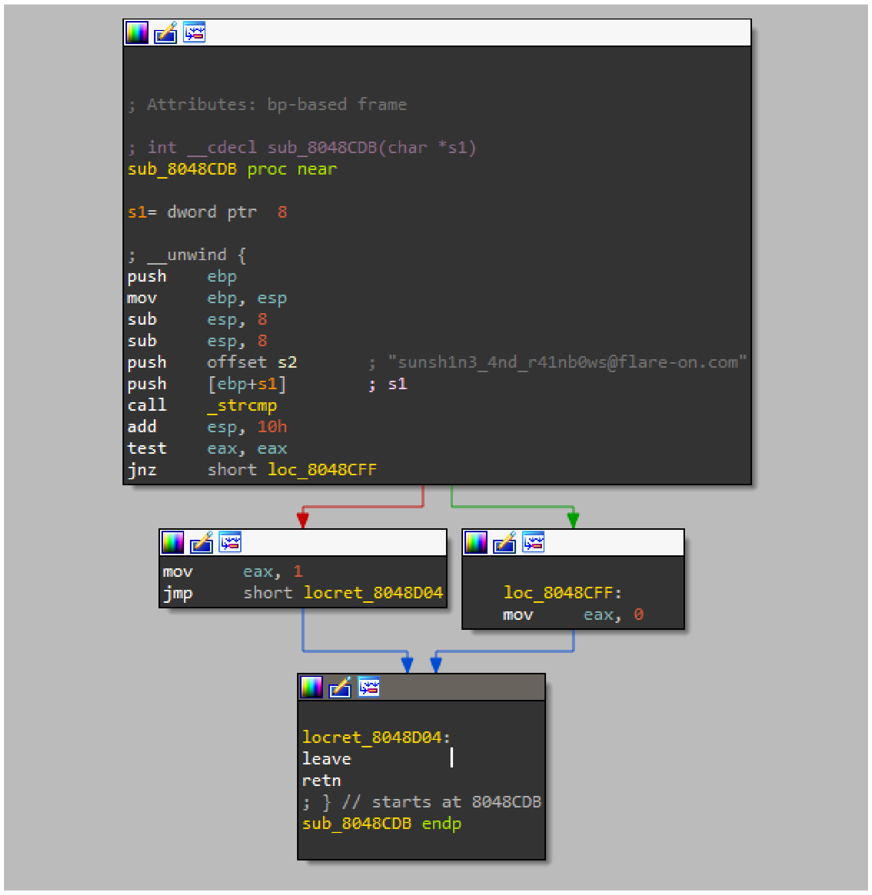
Flare-On 7: Challenge 10 – break

Challenge Author: Chris Gardner

Introduction

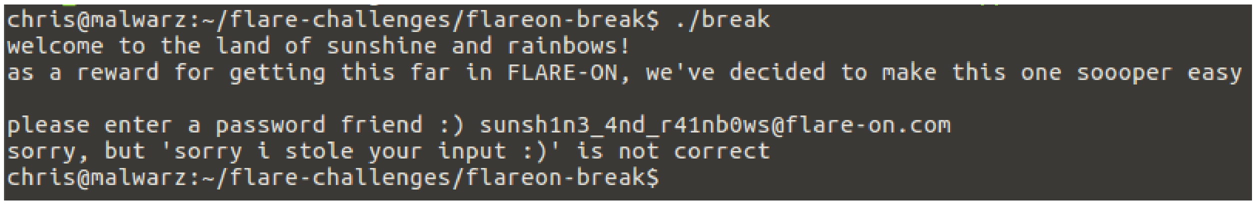
Break is 32 bit ELF that on the outside looks very simple, but has much more complexity than meets the eye. By disassembling the binary in IDA Pro we can see that the main function looks very simple – it passes the user input to a function that compares it with a hardcoded flag, as seen in Figure 1.



```
; Attributes: bp-based frame

; int __cdecl sub_8048CDB(char *s1)
sub_8048CDB proc near

s1= dword ptr  8

; __unwind {
push    ebp
mov     ebp, esp
sub     esp, 8
sub     esp, 8
push    offset s2       ; "sunsh1n3_4nd_r41nb0ws@flare-on.com"
push    [ebp+s1]        ; s1
call    _strcmp
add     esp, 10h
test    eax, eax
jnz     short loc_8048CFF

mov     eax, 1
jmp     short locret_8048D04

loc_8048CFF:
mov     eax, 0

locret_8048D04:
leave
retn
; } // starts at 8048CDB
sub_8048CDB endp
```

Figure 1: The fake flag checker function

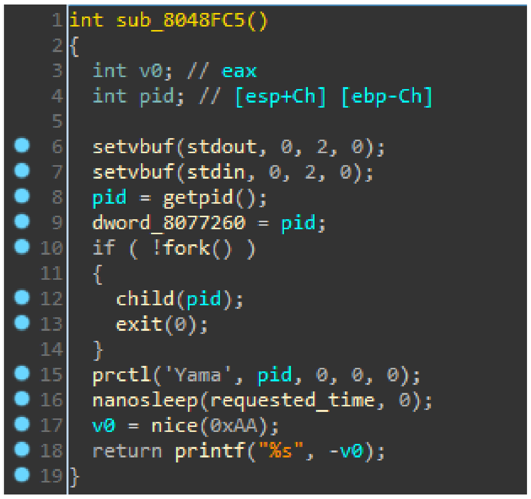
However, when running the binary not is all as it seems. Instead of printing out the success message, it prints out the fail message and our input is replaced with ‘sorry i stole your input ☺’. (Figure 2)



```
chris@malwarz:~/flare-challenges/flareon-break$ ./break
welcome to the land of sunshine and rainbows!
as a reward for getting this far in FLARE-ON, we've decided to make this one soooper easy

please enter a password friend :) sunsh1n3_4nd_r41nb0ws@flare-on.com
sorry, but 'sorry i stole your input :)' is not correct
chris@malwarz:~/flare-challenges/flareon-break$
```

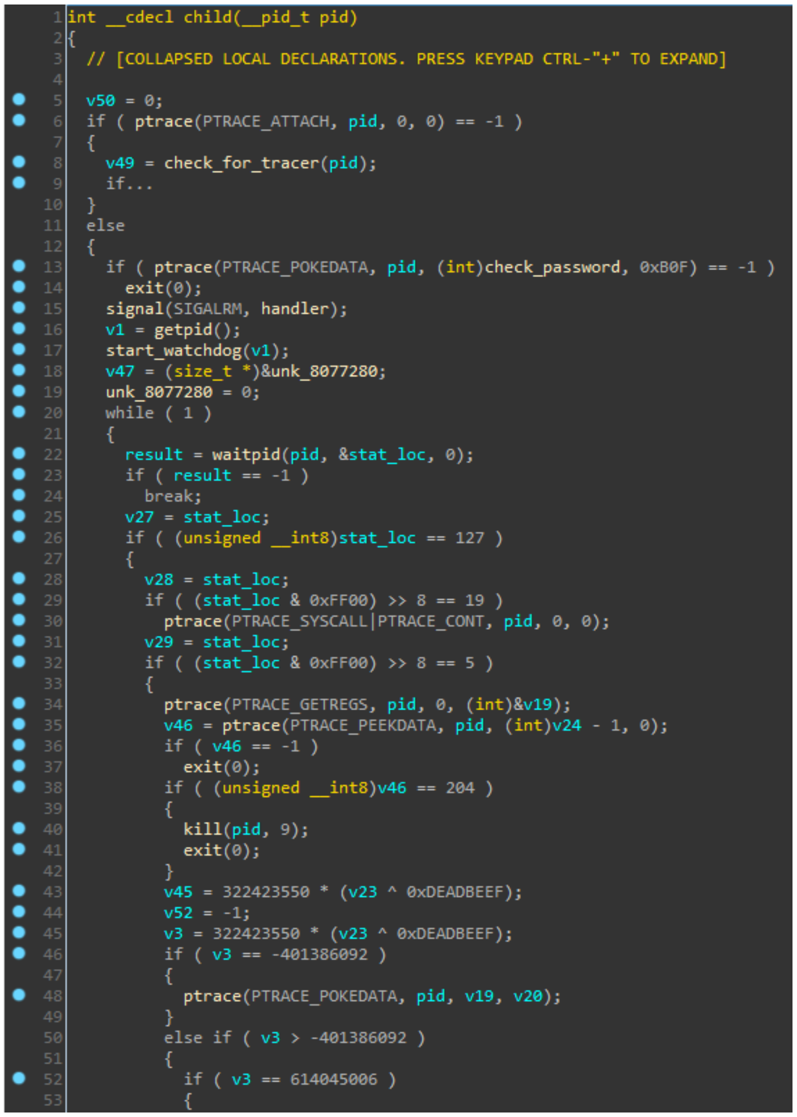
Figure 2: Output after giving the program the first fake flag

Clearly something more is going on here. We can examine the constructors array and notice that there is a function there (Figure 3), which appears to `fork()` and then call another, quite large function, seen in Figure 4. That large function also forks, and one of the subprocesses calls another large function.



```
int sub_8048FC5()
{
  int v0; // eax
  int pid; // [esp+Ch] [ebp-Ch]

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 2, 0);
  pid = getpid();
  dword_8077260 = pid;
  if ( !fork() )
  {
    child(pid);
    exit(0);
  }
  prctl('Yama', pid, 0, 0, 0);
  nanosleep(requested_time, 0);
  v0 = nice(0xAA);
  return printf("%s", -v0);
}
```

Figure 3: Constructor which launches the child



```
int __cdecl child(__pid_t pid)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v50 = 0;
  if ( ptrace(PTRACE_ATTACH, pid, 0, 0) == -1 )
  {
    v49 = check_for_tracer(pid);
    if...
  }
  else
  {
    if ( ptrace(PTRACE_POKEDATA, pid, (int)check_password, 0xB0F) == -1 )
      exit(0);
    signal(SIGALRM, handler);
    v1 = getpid();
    start_watchdog(v1);
    v47 = (size_t *)&unk_8077280;
    unk_8077280 = 0;
    while ( 1 )
    {
      result = waitpid(pid, &stat_loc, 0);
      if ( result == -1 )
        break;
      v27 = stat_loc;
      if ( (unsigned __int8)stat_loc == 127 )
      {
        v28 = stat_loc;
        if ( (stat_loc & 0xFF00) >> 8 == 19 )
          ptrace(PTRACE_SYSCALL|PTRACE_CONT, pid, 0, 0);
        v29 = stat_loc;
        if ( (stat_loc & 0xFF00) >> 8 == 5 )
        {
          ptrace(PTRACE_GETREGS, pid, 0, (int)&v19);
          v46 = ptrace(PTRACE_PEEKDATA, pid, (int)v24 - 1, 0);
          if ( v46 == -1 )
            exit(0);
          if ( (unsigned __int8)v46 == 204 )
          {
            kill(pid, 9);
            exit(0);
          }
          v45 = 322423550 * (v23 ^ 0xDEADBEEF);
          v52 = -1;
          v3 = 322423550 * (v23 ^ 0xDEADBEEF);
          if ( v3 == -401386092 )
          {
            ptrace(PTRACE_POKEDATA, pid, v19, v20);
          }
          else if ( v3 > -401386092 )
          {
            if ( v3 == 614045006 )
            {
```

Figure 4: Child process

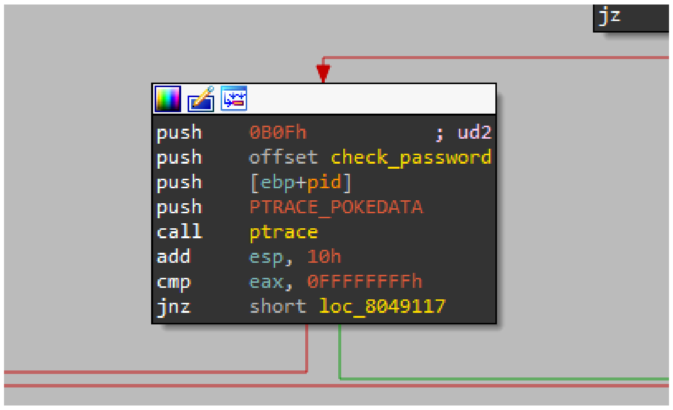
Challenge Architecture

Before attempting to reverse engineer any of the flag specific logic it is helpful to look at these large functions to gain an understanding of how this challenge is architected. There are three processes created during the runtime of the challenge:

- The parent, which is the original process and executes the `main()` function
- The child, which is the second process created and ptraces the parent
- The watchdog, which is the third process created and ptraces the child

The code is written in such a way that the parent will not function correctly without the presence of the child debugging it, and the child will not function correctly without the presence of the watchdog debugging it.

After the child forks and attaches to the parent, it overwrites the first instruction of the function called by main (Figure 1) with a `ud2` instruction (Figure 5). This instruction is intentionally undefined and will cause the process that executes it to crash with a `SIGILL` signal (illegal instruction). The child then starts the watchdog, resumes execution of the parent and waits for the parent to raise a signal.



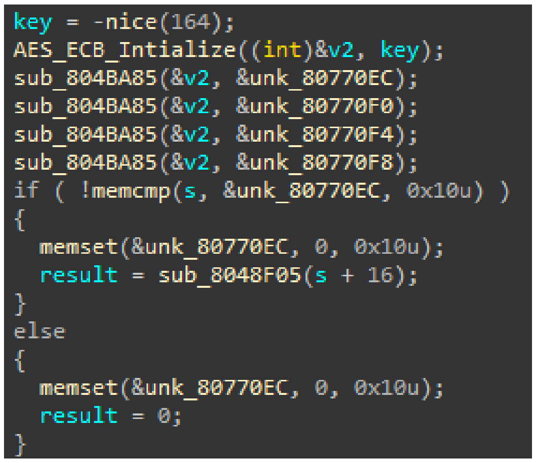
```
push    0B0Fh               ; ud2
push    offset check_password
push    [ebp+pid]
push    PTRACE_POKEDATA
call    ptrace
add     esp, 10h
cmp     eax, 0FFFFFFFFh
jnz     short loc_8049117
```

Figure 5: Overwriting the first instruction of check_password

Unlike most debuggers, the child does not use `PTRACE_CONT` to resume execution of the parent. IDA unhelpfully does not have the correct enum for the operation used but looking it up in the Linux headers we see that it is using `PTRACE_SYSEMU` to resume the parent. This is a special, x86 32-bit only flag that is uncommonly used. When a tracee is running under `PTRACE_SYSEMU` it will raise `SIGTRAP` whenever a syscall is about to be executed, but it will not actually execute the syscall. The syscall can then be emulated by the debugger, which is exactly what the child does. Instead of emulating the syscall exactly, it executes some code and acts as a sort of RPC mechanism for the parent. The syscall numbers are hashed with a simple algorithm as an anti-analysis technique.

The watchdog does something similar. When the child segfaults (raises `SIGSEGV`) by attempting to call a NULL pointer, the watchdog checks the arguments and executes some code. This is another form of RPC, except for the child. The RPC calls in both the child and the watchdog are sufficiently complex that reimplementing them in a GDB script so you can debug the parent would be difficult.

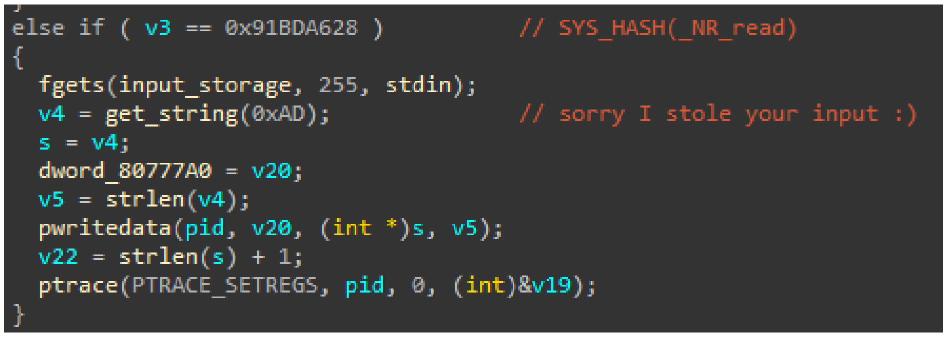
The parent typically calls syscalls by calling a glibc function, rather than calling the syscall directly. This leads to several interesting obfuscation techniques. For example, the parent calls the glibc function `nice()` very often, and from examining the code around it (Figure 6) we can see that it is seemingly used as a string decryption function. It turns out that the glibc function `nice()` does not actually call the `nice()` syscall – it instead calls `getpriority()` and `setpriority()`. The correct string decryption logic is implemented in the child's handlers for those two syscalls, and the child also includes a second fake string table that it decodes in the handler for the `nice()` syscall – which is never called, and merely exists to waste the player's time. The decrypted contents of both string tables are provided in Appendix A.



```
key = -nice(164);
AES_ECB_Intialize((int)&v2, key);
sub_804BA85(&v2, &unk_80770EC);
sub_804BA85(&v2, &unk_80770F0);
sub_804BA85(&v2, &unk_80770F4);
sub_804BA85(&v2, &unk_80770F8);
if ( !memcmp(s, &unk_80770EC, 0x10u) )
{
  memset(&unk_80770EC, 0, 0x10u);
  result = sub_8048F05(s + 16);
}
else
{
  memset(&unk_80770EC, 0, 0x10u);
  result = 0;
}
```

Figure 6: Using nice() to decode strings

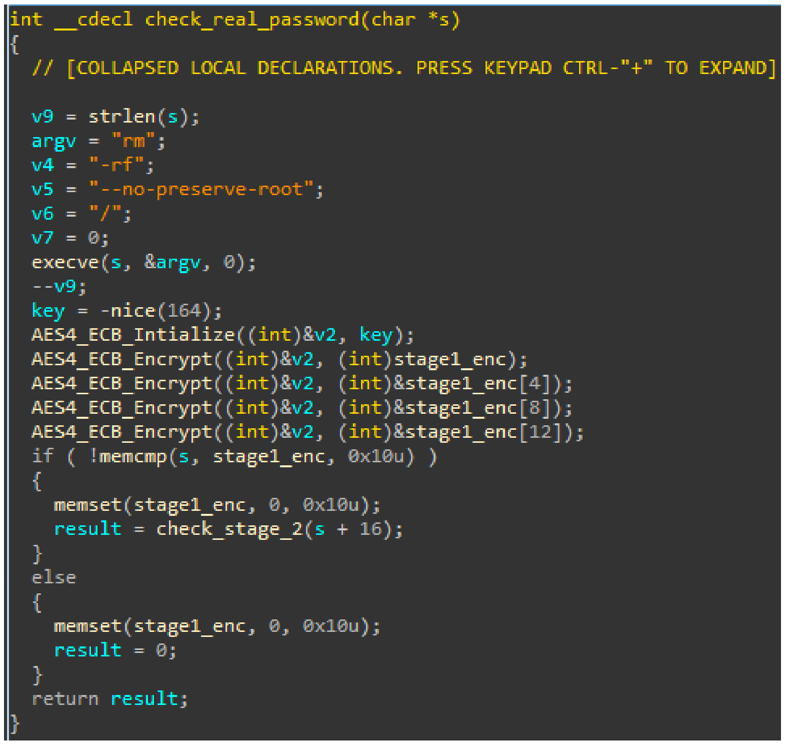
If we reverse the handler for the `read()` syscall (Figure 7), we see that the child reads some input from the player, stores it in the child's memory, and then writes a decoded string to the parent's memory space (the decoded string ends up being 'sorry I stole your input ☺'). This solves the mystery of where our input to the program went.



```
else if ( v3 == 0x91BDA628 )          // SYS_HASH(_NR_read)
{
  fgets(input_storage, 255, stdin);
  v4 = get_string(0xAD);              // sorry I stole your input :)
  s = v4;
  dword_80777A0 = v20;
  v5 = strlen(v4);
  pwritedata(pid, v20, (int *)s, v5);
  v22 = strlen(s) + 1;
  ptrace(PTRACE_SETREGS, pid, 0, (int)&v19);
}
```

Figure 7: Handler for the read() syscall

When the parent crashes with a `SIGILL` (intentionally triggered by the child as described earlier), the child will rewrite EIP to another function and write the stored input to a special location in memory. This other function is the first stage of checking the flag, seen in Figure 8.



```
int __cdecl check_real_password(char *s)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v9 = strlen(s);
  argv = "rm";
  v4 = "-rf";
  v5 = "--no-preserve-root";
  v6 = "/";
  v7 = 0;
  execve(s, &argv, 0);
  --v9;
  key = -nice(164);
  AES4_ECB_Intialize((int)&v2, key);
  AES4_ECB_Encrypt((int)&v2, (int)stage1_enc);
  AES4_ECB_Encrypt((int)&v2, (int)&stage1_enc[4]);
  AES4_ECB_Encrypt((int)&v2, (int)&stage1_enc[8]);
  AES4_ECB_Encrypt((int)&v2, (int)&stage1_enc[12]);
  if ( !memcmp(s, stage1_enc, 0x10u) )
  {
    memset(stage1_enc, 0, 0x10u);
    result = check_stage_2(s + 16);
  }
  else
  {
    memset(stage1_enc, 0, 0x10u);
    result = 0;
  }
  return result;
}
```

Figure 8: The stage 1 checker function

Stage 1 – AES-128-32-ECB

The first stage begins by calling `execve(input, {"rm", "-rf", "—no-preserve-root", "/"}, 0)`, which is terrifying. Thankfully, this call is caught by the child and is not executed, instead the `execve()` handler makes sure the input is null-terminated and does not end with a newline. The parent receives a string using `nice()`, and initializes a key structure. Analyzing the following function calls makes it appear that the challenge is just using normal AES, but actually it is using a modified version of AES with a block of size of 32 bits (as opposed to the normal 128 bits). This would be a pain to reimplement, but thankfully the challenge does not actually do anything with the input – instead it decrypts (it actually uses the AES encryption operation for this as another obfuscation trick) a ciphertext and compares it with the input. If we could attach a debugger to the parent, we could easily set a breakpoint here and just dump memory – but we cannot due to the presence of the child debugger.

The easiest way around this is to patch an infinite loop into the binary right after the ciphertext is decrypted (the challenge contains no anti-patching functionality, aside from a trick in stage 3), then dump memory from `/proc/<pid>/mem` (Figure 9). It is also possible to patch in an infinite loop, send a `SIGKILL` to the child (`SIGTERM`/etc are ignored and handled by the watchdog), then attach to the parent with GDB and read memory that way.

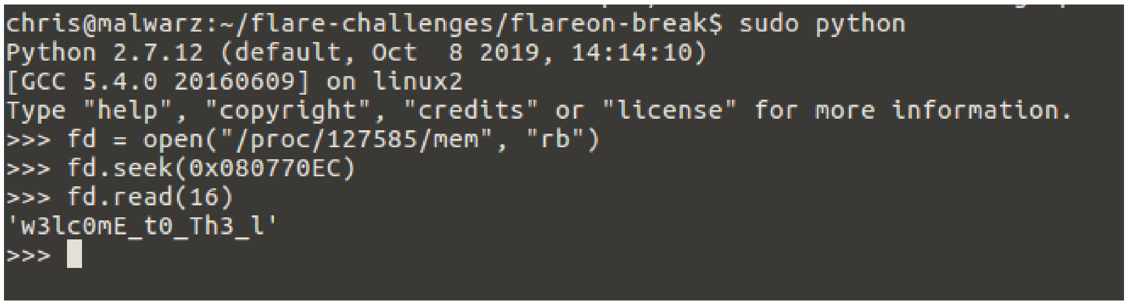

```
chris@malwarz:~/flare-challenges/flareon-break$ sudo python
Python 2.7.12 (default, Oct  8 2019, 14:14:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> fd = open("/proc/127585/mem", "rb")
>>> fd.seek(0x080770EC)
>>> fd.read(16)
'w3lc0mE_t0_Th3_l'
>>>
```

Figure 9: Dumping memory from the parent to get the first part of the flag

If the input matches the decrypted ciphertext (`w3lc0mE_t0_Th3_l`), then the next stage is started.

Stage 2 – Custom ARX Feistel Cipher

At a first glance it appears that stage 2 (see Figure 10) is similar in structure to stage 1. Unfortunately, stage 2 implements the flag check in the correct way: it encrypts the input you give it and compares it with a known encrypted ciphertext. Furthermore, the encryption function is very obfuscated and makes heavy use of syscall RPCs from the child, so it is difficult to determine the algorithm at a glance (the algorithm is custom, so we will need to implement it ourselves anyway). In fact, the execution of the encryption algorithm is 'weaved' throughout the parent, child, and watchdog, so examining all three processes is necessary. While it is possible to debug the challenge with some creativity, this writeup will examine the static analysis solution for determining what the algorithm is.

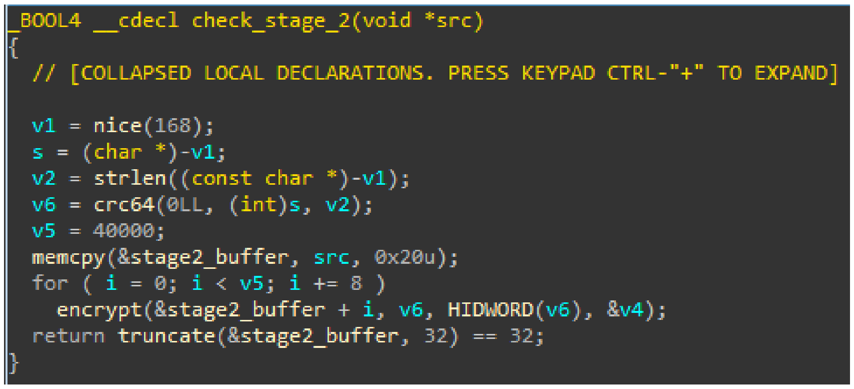
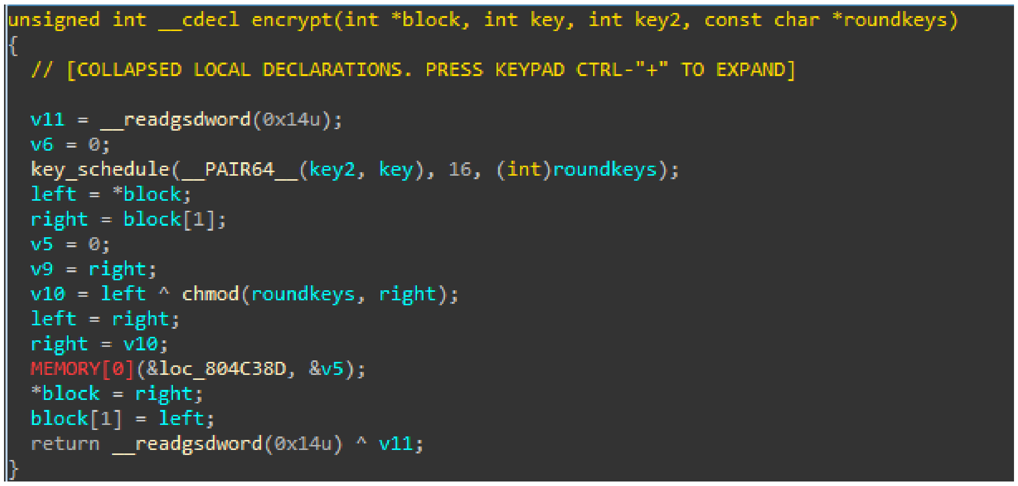

```
_BOOL4 __cdecl check_stage_2(void *src)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v1 = nice(168);
  s = (char *)-v1;
  v2 = strlen((const char *)-v1);
  v6 = crc64(0LL, (int)s, v2);
  v5 = 40000;
  memcpy(&stage2_buffer, src, 0x20u);
  for ( i = 0; i < v5; i += 8 )
    encrypt(&stage2_buffer + i, v6, HIDWORD(v6), &v4);
  return truncate(&stage2_buffer, 32) == 32;
}
```

Figure 10: Stage 2

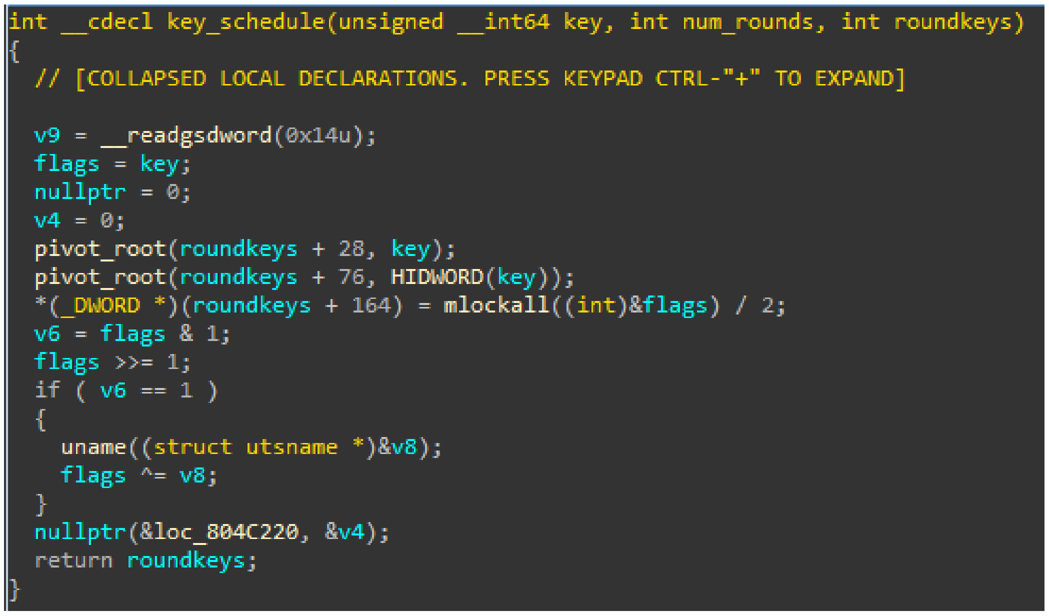
The encryption function (Figure 11) starts off by calling another function, which is the key scheduling algorithm. By examining the arguments to the encryption function and how they are used, we can deduce that this is a block cipher used in ECB mode, with a 64 bit block size and a 64 bit key.



```
unsigned int __cdecl encrypt(int *block, int key, int key2, const char *roundkeys)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v11 = __readgsdword(0x14u);
  v6 = 0;
  key_schedule(__PAIR64__(key2, key), 16, (int)roundkeys);
  left = *block;
  right = block[1];
  v5 = 0;
  v9 = right;
  v10 = left ^ chmod(roundkeys, right);
  left = right;
  right = v10;
  MEMORY[0](&loc_804C38D, &v5);
  *block = right;
  block[1] = left;
  return __readgsdword(0x14u) ^ v11;
}
```

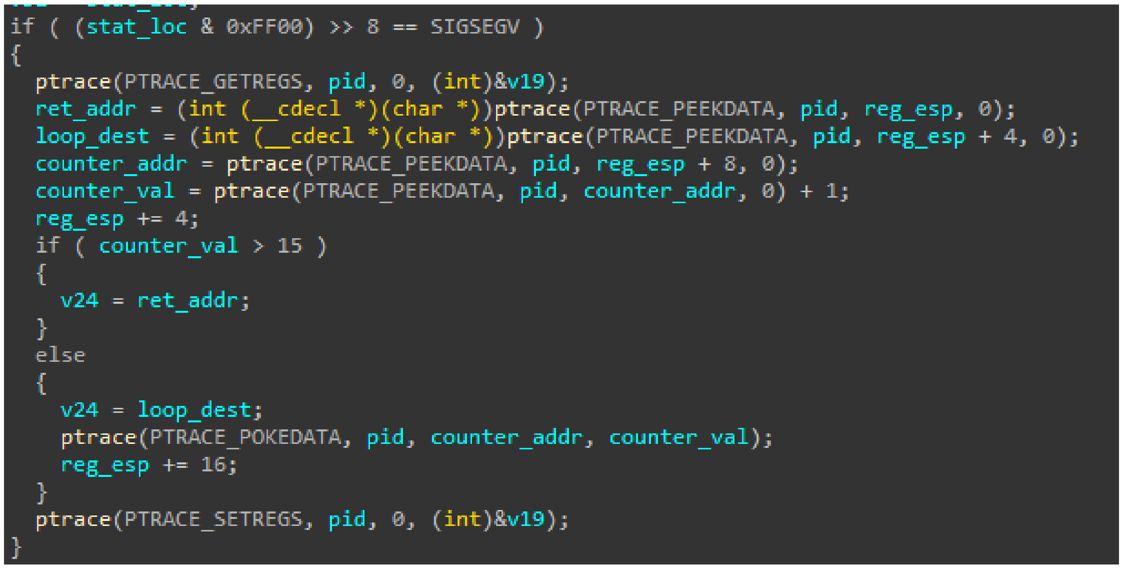
Figure 11: The encryption function

The key schedule looks very simple at first, but we can notice at the end it calls a NULL pointer and causes a segfault (Figure 12). Since this function is running in the parent, we can look at the handler for `SIGSEGV` in the child and determine that this acts as a loop, with the first argument to the NULL pointer being the instruction to loop back to, and the second argument a pointer to the current loop iteration. The number of iterations is hardcoded to 16, which happens to be the number of rounds in this cipher (Figure 13).



```
int __cdecl key_schedule(unsigned __int64 key, int num_rounds, int roundkeys)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v9 = __readgsdword(0x14u);
  flags = key;
  nullptr = 0;
  v4 = 0;
  pivot_root(roundkeys + 28, key);
  pivot_root(roundkeys + 76, HIDWORD(key));
  *(_DWORD *)(roundkeys + 164) = mlockall((int)&flags) / 2;
  v6 = flags & 1;
  flags >>= 1;
  if ( v6 == 1 )
  {
    uname((struct utsname *)&v8);
    flags ^= v8;
  }
  nullptr(&loc_804C220, &v4);
  return roundkeys;
}
```

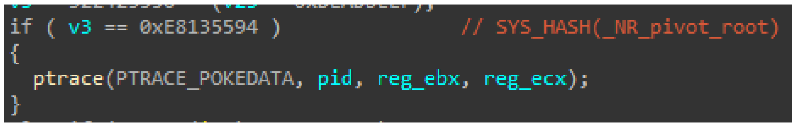
Figure 12: The key schedule, with segfault



```
if ( (stat_loc & 0xFF00) >> 8 == SIGSEGV )
{
  ptrace(PTRACE_GETREGS, pid, 0, (int)&v19);
  ret_addr = (int (__cdecl *)(char *))ptrace(PTRACE_PEEKDATA, pid, reg_esp, 0);
  loop_dest = (int (__cdecl *)(char *))ptrace(PTRACE_PEEKDATA, pid, reg_esp + 4, 0);
  counter_addr = ptrace(PTRACE_PEEKDATA, pid, reg_esp + 8, 0);
  counter_val = ptrace(PTRACE_PEEKDATA, pid, counter_addr, 0) + 1;
  reg_esp += 4;
  if ( counter_val > 15 )
  {
    v24 = ret_addr;
  }
  else
  {
    v24 = loop_dest;
    ptrace(PTRACE_POKEDATA, pid, counter_addr, counter_val);
    reg_esp += 16;
  }
  ptrace(PTRACE_SETREGS, pid, 0, (int)&v19);
}
```

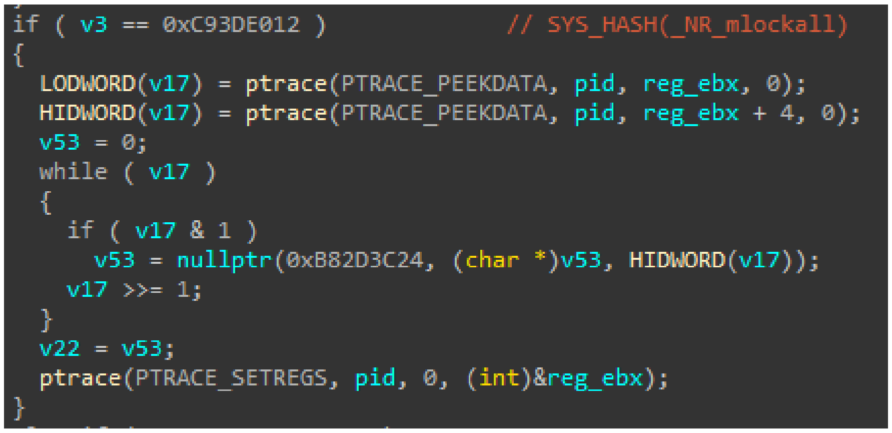
Figure 13: Segfault handler in the child

The key scheduling function calls two glibc functions (see Figure 12), `pivot_root` and `mlockall`. The `pivot_root` handler in the child is simple – it writes the second argument into the first argument (Figure 14). The `mlockall` handler is much more complicated and is the first time the watchdog gets involved (Figure 15). It checks each bit of the argument (a 64 bit integer), and does an RPC call to the watchdog if the bit is set. The watchdog handler for this RPC ends up just incrementing one of the arguments (Figure 16).



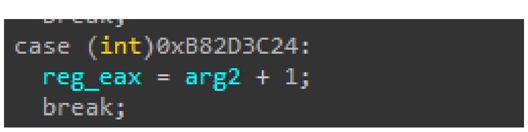
```
if ( v3 == 0xE8135594 )             // SYS_HASH(_NR_pivot_root)
{
  ptrace(PTRACE_POKEDATA, pid, reg_ebx, reg_ecx);
}
```

Figure 14: pivot_root handler



```
if ( v3 == 0xC93DE012 )             // SYS_HASH(_NR_mlockall)
{
  LODWORD(v17) = ptrace(PTRACE_PEEKDATA, pid, reg_ebx, 0);
  HIDWORD(v17) = ptrace(PTRACE_PEEKDATA, pid, reg_ebx + 4, 0);
  v53 = 0;
  while ( v17 )
  {
    if ( v17 & 1 )
      v53 = nullptr(0xB82D3C24, (char *)v53, HIDWORD(v17));
    v17 >>= 1;
  }
  v22 = v53;
  ptrace(PTRACE_SETREGS, pid, 0, (int)&reg_ebx);
}
```

Figure 15: mlockall handler



```
case (int)0xB82D3C24:
  reg_eax = arg2 + 1;
  break;
```

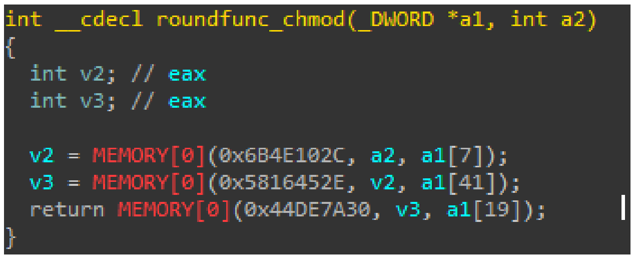
Figure 16: watchdog handler for the RPC used in the mlockall handler

In effect, the `mlockall` counts the number of bits that are set in the state and sets one of the key components to that. Each round key has three components: the high 32 bits of the state, the low 32 bits of the state, and the number of set bits in the state. After calculating one round key, the state is advanced (Figure 12). The state is shifted right by 1, and if the least significant bit of the state is 1 then a value is retrieved from the child using the `uname` syscall, and then XORed with the state. The value retrieved with `uname` ends up being `0x9e3779b9C6EF3720`. The original, unobfuscated C source code for the key schedule is presented in Figure 17.

```
struct roundkey* keysched(uint64_t key, uint32_t rounds, struct roundkey* rks)
{
        uint64_t state = key;
        for(int i = 0; i < rounds; i++)
        {
                //output state
                rks[i].addkey = (uint32_t)(state & 0xffffffff);
                rks[i].xorkey = (uint32_t)(state >> 32);
                rks[i].rorkey = count_1s(state) / 2;
                //advance state
                int lsb = state & 1;
                state = state >> 1;
                if(lsb == 1)
                {
                        state = state ^ 0x9e3779b9C6EF3720;
                }
        }
        return rks;
}
```

Figure 17: Source code of the key schedule

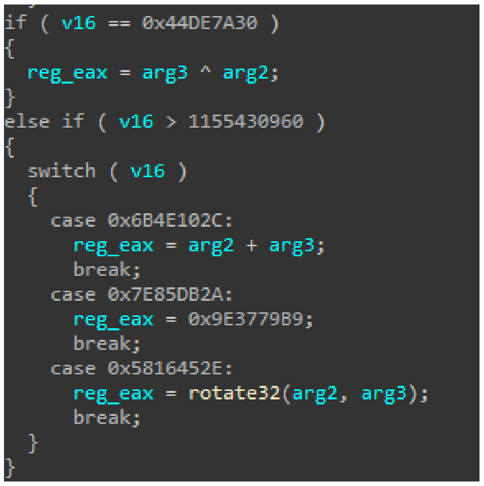
After computing the key schedule, the encryption function does another loop with 16 iterations, implemented by calling a null pointer in the same way as the key schedule. The encryption function operates on the lower 32 bits of the input/state (the left block), and the upper 32 bits of the input/state (the right block). The left block is XORed with the result of the `chmod()` syscall, which is given the arguments (current round key, right block). The order of the blocks is then flipped (Figure 11).



```
int __cdecl roundfunc_chmod(_DWORD *a1, int a2)
{
  int v2; // eax
  int v3; // eax

  v2 = MEMORY[0](0x6B4E102C, a2, a1[7]);
  v3 = MEMORY[0](0x5816452E, v2, a1[41]);
  return MEMORY[0](0x44DE7A30, v3, a1[19]);
}
```

Figure 18: The round function

The `chmod()` syscall is handled in the child, and is the round function (Figure 18). The round function takes the three components of the round key, and does 3 RPC calls to the watchdog FIG. Examining the RPC calls we see that they are pretty simple – one adds, one rotates right, and one XORs (Figure 19). This is a simple ARX round function, and with that we have everything we need to decrypt this part of the flag. The original, unobfuscated C code for the encryption function is presented in Figure 20.



```
if ( v16 == 0x44DE7A30 )
{
  reg_eax = arg3 ^ arg2;
}
else if ( v16 > 1155430960 )
{
  switch ( v16 )
  {
    case 0x6B4E102C:
      reg_eax = arg2 + arg3;
      break;
    case 0x7E85DB2A:
      reg_eax = 0x9E3779B9;
      break;
    case 0x5816452E:
      reg_eax = rotate32(arg2, arg3);
      break;
  }
}
```

Figure 19: The round function handlers in the watchdog

```
uint32_t roundfunc(struct roundkey* key, uint32_t block)
{
        uint32_t b = block;
        b += key->addkey;
        b = rotr32(b, key->rorkey);
        b = b ^ key->xorkey;
        return b;
}
void encrypt(uint32_t* block, uint64_t key, uint32_t rounds, struct roundkey*
rks)
{
         keysched(key, rounds, rks);
        uint32_t l = block[0];
        uint32_t r = block[1];
        for(int i = 0; i < rounds; i++)
        {
                uint32_t nl = r;
                uint32_t nr = l ^ roundfunc(&rks[i], r);
                l = nl;
                r = nr;
        }
        block[0] = r;
        block[1] = l;
}
```

Figure 20: Source code of the encryption function

Fans of cryptography will recognize the general structure of the encryption algorithm as a Feistel Cipher, which work by splitting the current block into two parts, running one part through a round function and XORing it with the other part, and then swapping the two parts for the next round (Figure 21). While the challenge only contains code for encrypting data, we can use a helpful property of Feistel Ciphers to derive the decryption method. For a Feistel Cipher, the decryption function is the same as the encryption function, except the key schedule is reversed. We don't even have to re-implement the key schedule, as we can dump the output of it from the program itself (using an infinite loop patching method like was used to recover the stage 1 flag) and then reverse it. Only the encryption algorithm and round function need to be implemented.

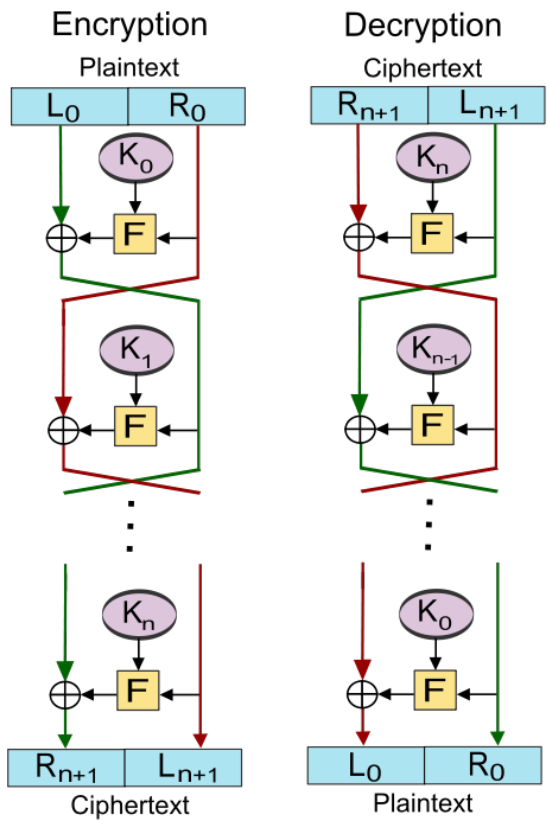
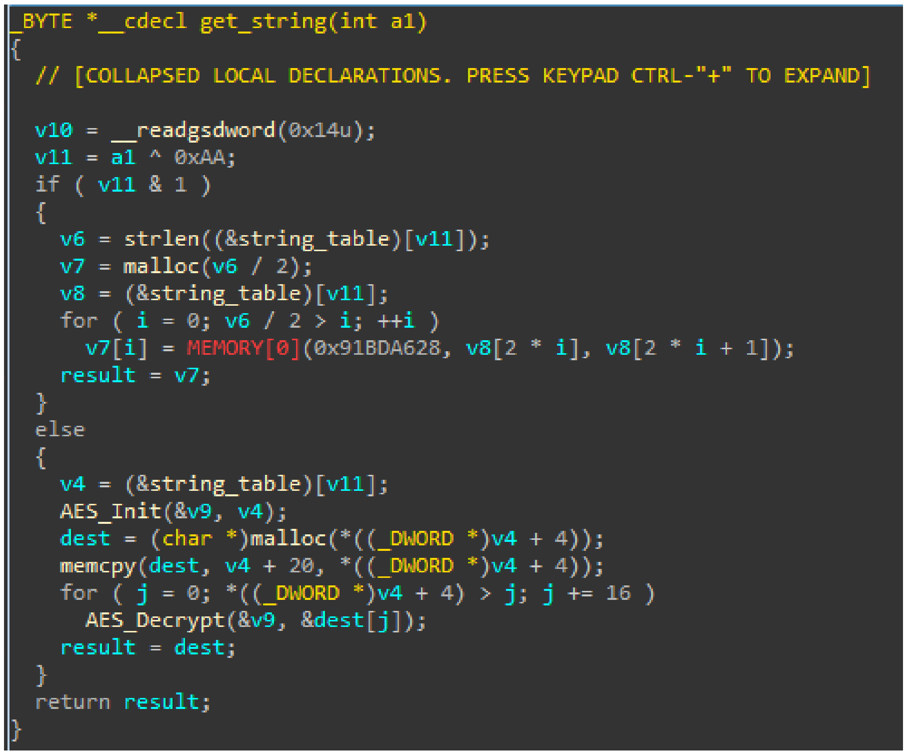


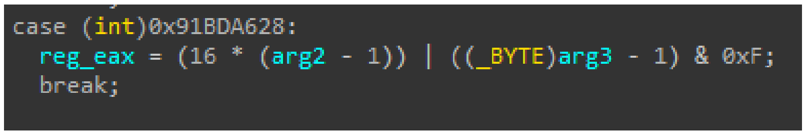
Figure 21: Diagram of a Feistel Cipher (from https://en.wikipedia.org/wiki/Feistel_cipher#/media/File:Feistel_cipher_diagram_en.svg)

Now we need to grab the key and ciphertext from the challenge. The key is easy to extract via an infinite loop patch at `0x8048F47`, but a static solution is presented here for variety. A string is retrieved from the child using the `nice()` glibc function, which as described earlier retrieves a string from a global string table. Then, the crc64 of that string is calculated and used as the key (see Figure 10). `nice()` ends up calling `get_priority()` and `set_priority()`, and the handler for `set_priority()` calls a function at `0x804C401` (get_string). The string decoding function contains two ways of decoding strings, which method is used is determined by if the index into the string table is even or odd. If the index is even, a 16-byte key is loaded from the string table and the string is decrypted with AES-128-ECB as shown in Figure 22. If the index is odd, the child parses the encrypted string two characters at time and makes an RPC call to the watchdog, the handler for which is shown in Figure 23. The handler decodes the two characters as hex digits with a swapped character set (a-q instead of 0-f). For added fun the case bit of the characters is randomized for each encoded string (this has no effect on the string decoding).



```
_BYTE *__cdecl get_string(int a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v10 = __readgsdword(0x14u);
  v11 = a1 ^ 0xAA;
  if ( v11 & 1 )
  {
    v6 = strlen((&string_table)[v11]);
    v7 = malloc(v6 / 2);
    v8 = (&string_table)[v11];
    for ( i = 0; v6 / 2 > i; ++i )
      v7[i] = MEMORY[0](0x91BDA628, v8[2 * i], v8[2 * i + 1]);
    result = v7;
  }
  else
  {
    v4 = (&string_table)[v11];
    AES_Init(&v9, v4);
    dest = (char *)malloc(*((_DWORD *)v4 + 4));
    memcpy(dest, v4 + 20, *((_DWORD *)v4 + 4));
    for ( j = 0; *((_DWORD *)v4 + 4) > j; j += 16 )
      AES_Decrypt(&v9, &dest[j]);
    result = dest;
  }
  return result;
}
```

Figure 22: get_string

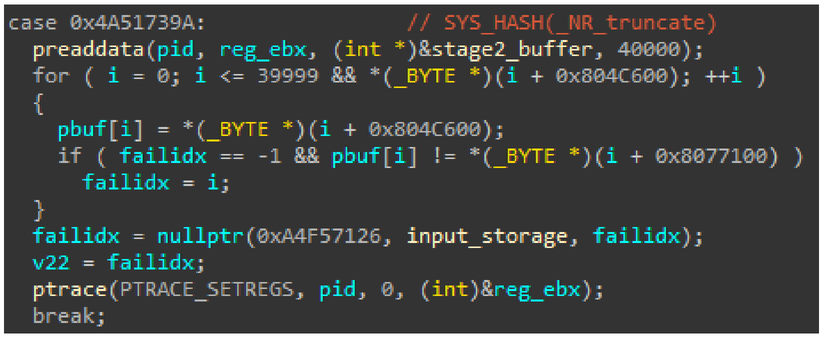


```
case (int)0x91BDA628:
  reg_eax = (16 * (arg2 - 1)) | ((_BYTE)arg3 - 1) & 0xF;
  break;
```

Figure 23: Watchdog handler for string decryption

Decoding the string used for the key material yields “This string has no purpose and is merely here to waste your time.”, the crc64 of is `7442794226307913737`. The parent does not directly compare the encrypted input with the ciphertext, instead it passes the input to the child via the `truncate()` syscall, along with the number 32 (which we can infer is the length of this part of the flag). It is interesting to note that the parent encrypts more than 32 bytes (it actually encrypts 40kb of data). Dumping the data that is encrypted after the input shows that the data in the binary encrypts to the start of the script from the ‘Bee Movie’. While this data is important for stage 3, for now it all it does is make the encryption take a long time to compute (30 seconds to a minute on the test VM). Players are thus incentivized to shorten the length if they want to rapidly iterate over the program, which has repercussions in stage 3.

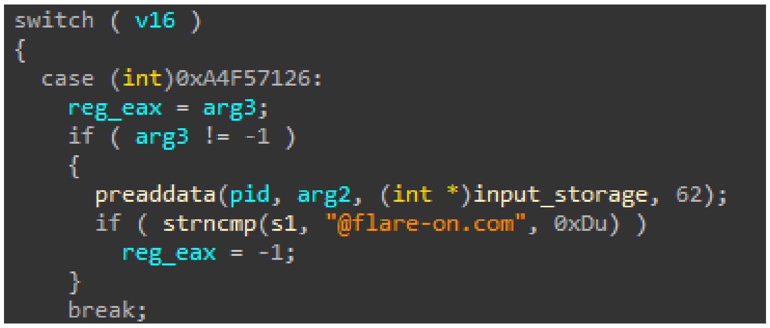
The handler for `truncate()` in the child is a convoluted, hybrid memcmp/strcpy, shown in Figure 24. The child reads the encrypted input into memory, and then copies into a buffer on the stack, comparing with a known ciphertext stored in the binary. Importantly, the operation does not end until a NULL byte is encountered in the input. This can lead to a stack-based buffer overflow, which will be explored more in stage 3. Decrypting the ciphertext with the key gives us the plaintext: ‘`4nD_0f_De4th_4nd_d3strUct1oN_4nd`’.



```
case 0x4A51739A:                    // SYS_HASH(_NR_truncate)
  preaddata(pid, reg_ebx, (int *)&stage2_buffer, 40000);
  for ( i = 0; i <= 39999 && *(_BYTE *)(i + 0x804C600); ++i )
  {
    pbuf[i] = *(_BYTE *)(i + 0x804C600);
    if ( failidx == -1 && pbuf[i] != *(_BYTE *)(i + 0x8077100) )
      failidx = i;
  }
  failidx = nullptr(0xA4F57126, input_storage, failidx);
  v22 = failidx;
  ptrace(PTRACE_SETREGS, pid, 0, (int)&reg_ebx);
  break;
```

Figure 24: Handler for truncate

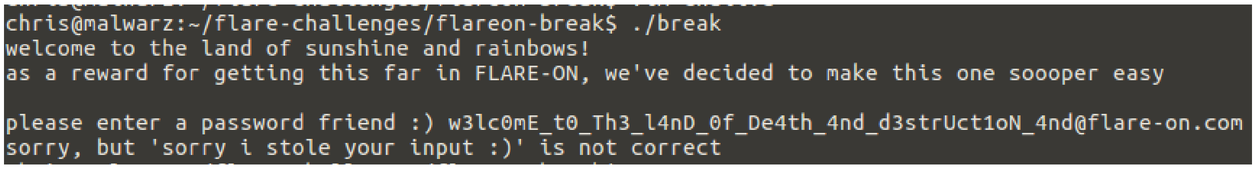
After copying the input to the stack, the child makes an RPC call to the watchdog, giving the original plaintext input entered by the user and the number of correct characters compared with. The RPC handler, shown in Figure 25, checks to see if the number of correct characters is not -1, and that the string ends with “@flare-on.com”, the flag format. Interestingly, it does not check to see if the number of correct characters is 32 as we would expect. This gives us the flag “`w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd@flare-on.com`”.



```
switch ( v16 )
{
  case (int)0xA4F57126:
    reg_eax = arg3;
    if ( arg3 != -1 )
    {
      preaddata(pid, arg2, (int *)input_storage, 62);
      if ( strncmp(s1, "@flare-on.com", 0xDu) )
        reg_eax = -1;
    }
    break;
```

Figure 25: Watchdog handler that checks if the flag meets the flag format

However, when entering this flag into the program it still rejects our input (see Figure 26). Attaching to the watchdog and setting a breakpoint on the RPC call that checks the end of the flag shows us that the code path is never hit.

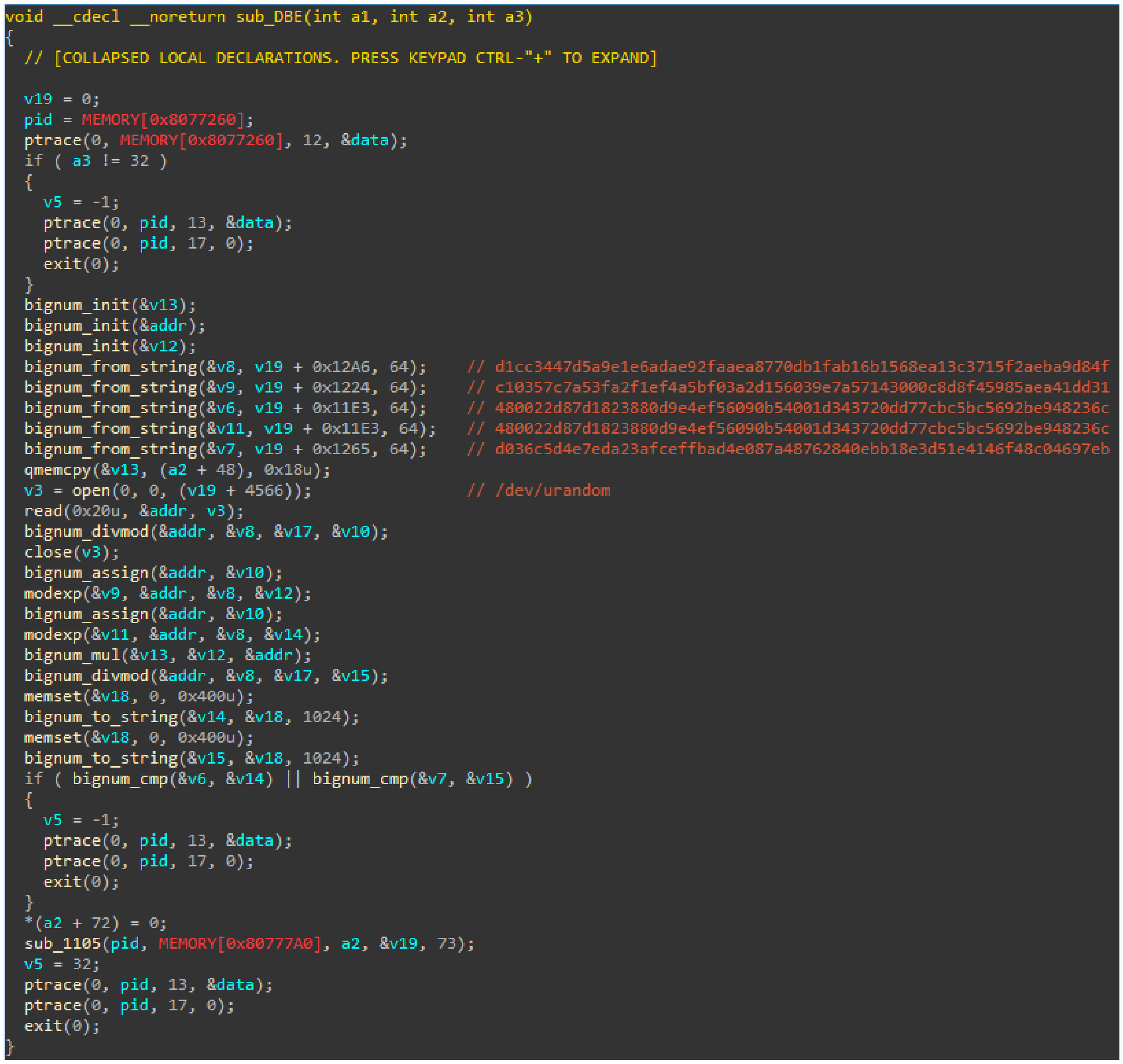


```
chris@malwarz:~/flare-challenges/flareon-break$ ./break
welcome to the land of sunshine and rainbows!
as a reward for getting this far in FLARE-ON, we've decided to make this one soooper easy

please enter a password friend :) w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd@flare-on.com
sorry, but 'sorry i stole your input :)' is not correct
```

Figure 26: Running the challenge with the flag we have so far

Stage 3 – Bug in bignum shellcode library leads to break of ElGamal

Earlier, we discussed that the handler for `truncate()` in the child has a stack based buffer overflow if the encrypted data after the input (which is the script from the 'Bee Movie') does not contain a NULL byte within the first 16000 bytes. It turns out the input contains the first NULL byte at byte 16169, and immediately before that are 4 non-ascii bytes that look like an address in the program (more 'Bee Movie' script follows the NULL bytes). The way the stack is set up, this input will end up being written where the NULL pointer used for RPC calls to the watchdog is stored (on the stack). So instead of segfaulting and calling the watchdog to check the flag format, it will call this address instead. Dumping memory at that address (can be done in either the parent or child) shows us some very interesting shellcode (Figure 27). Note that patching the size of the encrypted data to make the challenge run faster will cause the bug to disappear and cause the challenge to accept an incorrect flag.



```
void __cdecl __noreturn sub_DBE(int a1, int a2, int a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v19 = 0;
  pid = MEMORY[0x8077260];
  ptrace(0, MEMORY[0x8077260], 12, &data);
  if ( a3 != 32 )
  {
    v5 = -1;
    ptrace(0, pid, 13, &data);
    ptrace(0, pid, 17, 0);
    exit(0);
  }
  bignum_init(&v13);
  bignum_init(&addr);
  bignum_init(&v12);
  bignum_from_string(&v8, v19 + 0x12A6, 64);    // d1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba9d84f
  bignum_from_string(&v9, v19 + 0x1224, 64);    // c10357c7a53fa2f1ef4a5bf03a2d156039e7a57143000c8d8f45985aea41dd31
  bignum_from_string(&v6, v19 + 0x11E3, 64);    // 480022d87d1823880d9e4ef56090b54001d343720dd77cbc5bc5692be948236c
  bignum_from_string(&v11, v19 + 0x11E3, 64);   // 480022d87d1823880d9e4ef56090b54001d343720dd77cbc5bc5692be948236c
  bignum_from_string(&v7, v19 + 0x1265, 64);    // d036c5d4e7eda23afceffbad4e087a48762840ebb18e3d51e4146f48c04697eb
  qmemcpy(&v13, (a2 + 48), 0x18u);
  v3 = open(0, 0, (v19 + 4566));                // /dev/urandom
  read(0x20u, &addr, v3);
  bignum_divmod(&addr, &v8, &v17, &v10);
  close(v3);
  bignum_assign(&addr, &v10);
  modexp(&v9, &addr, &v8, &v12);
  bignum_assign(&addr, &v10);
  modexp(&v11, &addr, &v8, &v14);
  bignum_mul(&v13, &v12, &addr);
  bignum_divmod(&addr, &v8, &v17, &v15);
  memset(&v18, 0, 0x400u);
  bignum_to_string(&v14, &v18, 1024);
  memset(&v18, 0, 0x400u);
  bignum_to_string(&v15, &v18, 1024);
  if ( bignum_cmp(&v6, &v14) || bignum_cmp(&v7, &v15) )
  {
    v5 = -1;
    ptrace(0, pid, 13, &data);
    ptrace(0, pid, 17, 0);
    exit(0);
  }
  *(a2 + 72) = 0;
  sub_1105(pid, MEMORY[0x80777A0], a2, &v19, 73);
  v5 = 32;
  ptrace(0, pid, 13, &data);
  ptrace(0, pid, 17, 0);
  exit(0);
}
```

Figure 27: Decrypted shellcode payload

Thankfully, this shellcode does not use any of the RPC obfuscation mechanisms used in stages 1 and 2 (except for a couple ptrace calls to fix things up when the shellcode returns). Although the shellcode contains a lot of functions, almost all of them are bignum operations from https://github.com/kokke/tiny-bignum-c. The only exceptions are the main function, and the function at offset `0x9c3`. This function can be quite hard to label, as it is an optimized modular exponentiation function (specifically, it is the "Modular Exponentiation by Squaring" algorithm described in https://eli.thegreenplace.net/2009/03/28/efficient-modular-exponentiation-algorithms). However, the details of this function end up being unimportant due to a bug in this function (described later). With all the functions labeled, we can create pseudocode for the mathematical operations in the main function, shown in Figure 28. This is a straightforward implementation of the ElGamal cryptosystem, albeit used in an unusual way.

```
P = d1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba9d84f
H = c10357c7a53fa2f1ef4a5bf03a2d156039e7a57143000c8d8f45985aea41dd31
C1 = 480022d87d1823880d9e4ef56090b54001d343720dd77cbc5bc5692be948236c
G = 480022d87d1823880d9e4ef56090b54001d343720dd77cbc5bc5692be948236c
C2 = d036c5d4e7eda23afceffbad4e087a48762840ebb18e3d51e4146f48c04697eb
Y = <random bignum between 1 and P>
M = <last part of player's input>
S = H ^ Y mod P
C1* = G ^ Y mod P
C2* = M * S mod P
If C1 == C1* and C2 == C2*:
       Return success
Else:
       Return failure
```

Figure 28: Pseudocode for shellcode

At first glance, it should be impossible for this check to succeed. C1* is based on a random value, but is compared with a hardcoded C1 (and there is no bug in the random number generation). Additionally, C1 is equal to G, which seems impossible unless Y = 1. However, examining the modular exponentiation in a debugger shows that it always returns the first argument unmodified. The C source code for the broken function is shown in Figure 29.

The bug here is in how the bignums are compared. The bignum implementation contains a method, bignum_cmp, which treats two bignums as little endian integers and compares them. However, this function uses its own code to compare two bignums, by treating them as big endian integers. Since the size of bignums in the library is much larger than the numbers used in the challenge ever get, the highest bits of each bignum are always 0, and the first two if statements are taken which causes the function to return immediately after assigning r to a.

```
void modexp(struct bn* a, struct bn* b, struct bn* p, struct bn* r)
{
        struct bn t;
        struct bn const_2;
        struct bn const_1;
        struct bn const_0;
        struct bn unused;
        bignum_init(&t);
        bignum_from_int(r, 1);
        bignum_from_int(&const_2, 2);
        bignum_from_int(&const_1, 1);
        bignum_from_int(&const_0, 0);
        bignum_divmod(b, &const_2, &unused, &t);
        if(t.array[BN_ARRAY_SIZE-1] == const_1.array[BN_ARRAY_SIZE-1])
        {
                bignum_assign(r, a);
                bignum_div(b, &const_2, &t);
                bignum_assign(b, &t);
                if(b->array[BN_ARRAY_SIZE-1] == const_0.array[BN_ARRAY_SIZE-1])
                {
                        return;
                }
                bignum_mul(a, a, &t);
                bignum_divmod(&t, p, &unused, a);

        }
        while(1)
        {
                bignum_divmod(b, &const_2, &unused, &t);
                if(t.array[BN_ARRAY_SIZE-1] != const_0.array[BN_ARRAY_SIZE-1] )
                {
                        bignum_mul(r, a, &t);
                        bignum_divmod(&t, p, &unused, r);
                }
                bignum_div(b, &const_2, &t);
                bignum_assign(b, &t);
                if(b->array[BN_ARRAY_SIZE-1] == const_0.array[BN_ARRAY_SIZE-1])
                {
                        break;
                }
                bignum_mul(a, a, &t);
                bignum_divmod(&t, p, &unused, a);
        }
        return;
}
```

Figure 29: C source for broken modular exponentiation function

Now that we know the bug, we can produce a version of the pseudocode for the algorithm that is true to what actually happens, shown in Figure 30.

```
P = d1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba9d84f
H = c10357c7a53fa2f1ef4a5bf03a2d156039e7a57143000c8d8f45985aea41dd31
C1 = 480022d87d1823880d9e4ef56090b54001d343720dd77cbc5bc5692be948236c
G = 480022d87d1823880d9e4ef56090b54001d343720dd77cbc5bc5692be948236c
C2 = d036c5d4e7eda23afceffbad4e087a48762840ebb18e3d51e4146f48c04697eb
Y = <random bignum between 1 and P>
M = <last part of player's input>
S = H
C1* = G
C2* = M * S mod P
If C1 == C1* and C2 == C2*:
      Return success
Else:
      Return failure
```

Figure 30: Correct pseudocode for shellcode

We can see that C1* will always equal C1, since G == C1. We can calculate the original message M', and thus what we need to input to get this program to return success. Since C2 = M' * S mod P, to get M' we simply compute M' = C2 / S mod P. This is easiest to compute by first computing T = S ^ -1 mod P, and then computing M' = C2 * T mod P. Wolfram Alpha will correctly compute T, many other math libraries calculate it incorrectly. This shows us the final part of the flag: `_n0_puppi3s@flare-on.com`. This gives us the final flag:
"`w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd_n0_puppi3s@flare-on.com`", running the program with this as the input causes the challenge to finally output the success message.

Appendix A – Decoded strings

The real string table contains the following strings:

```
sorry i stole your input :)
This string has no purpose and is merely here to waste your time. # stage 2 key
\xe3vi\x92\xc7\\\xe8\xf5\x85\xc5M\x11\x16\xfa\xf4\xe8\x00 # stage 1 key
\x2f\xd4\x7f\x00\x98\x1c\x9c\x29\x3f\xce\xbf\xb6\xa1\xd4\xbf\x6b # stage 1 enc
This string also has no purpose and again, is here merely to waste your time.
This string is used to calculate the XXTEA key to decrypt the fourth part of the
flag.
sm0l_bin_b1g_h34rt@flare-on.com
fake_flag@flare-on.com
not_a_fake_flag@flare-on.com
okay_1_sw34r_th1s_1s_th3_r34l_0ne@flare-on.com
elf_0n_a_sh3lf@on-flare.com
moc.no-eralf@gn1hs1n1f_n0_st4rgn0c
/proc/%d/status
TracerPid:
TracerPid: %d
OOPSIE WOOPSIE!! Uwu We made a fucky wucky!!!!
I HAVE THE CONCH DON'T INTERRUPT ME
Like a phoenix, I rise from the ashes
winners never quit
```

The fake string table contains the following strings, in addition to the entire script from the 'Bee Movie':

```
@flare-on.com
Thank you for playing FLARE-ON!
sm(shellcraft.sh()) + "
Look! Some key material is coming up!
<empty string>
Wow. That was some good key material.
Please fill out the review sheet for this challenge:
https://twitter.com/gf_256/status/1209012768147460096/photo/1
By reading this string I have successfully stolen around 2 of your life. How
does that make you feel?
Wasting your time1
Wasting your time2
Wasting your time3
Wasting your time4
Wasting your time5
Wasting your time6
Wasting your time7
Wasting your time8
Wasting your time9
Wasting your time10
Wasting your time11
Wasting your time12
Wasting your time13
Wasting your time14
Wasting your time15
Wasting your time16
Wasting your time17
Wasting your time18
Wasting your time19
Wasting your time20
Wasting your time21
Wasting your time22
Wasting your time23
Wasting your time24
Wasting your time25
NaN
Error: Unable to debug child
```