Problem 7 Solution: Welcome to Recursion Hell

By Ruijie Fang

The solution to this problem is somewhat unusual. There are 4 people who scored full points on both parts of this problem, and their solutions all differ in varying degrees from our solution. We attach their solutions as well, and comment on them in section 3. We'd like to highlight that in general, you shouldn't attempt to use the techniques you developed for this problem in practice, since handling segmentation faults and stack overflows are rather sensitive issues, and a million things could go wrong in an unrestricted environment.

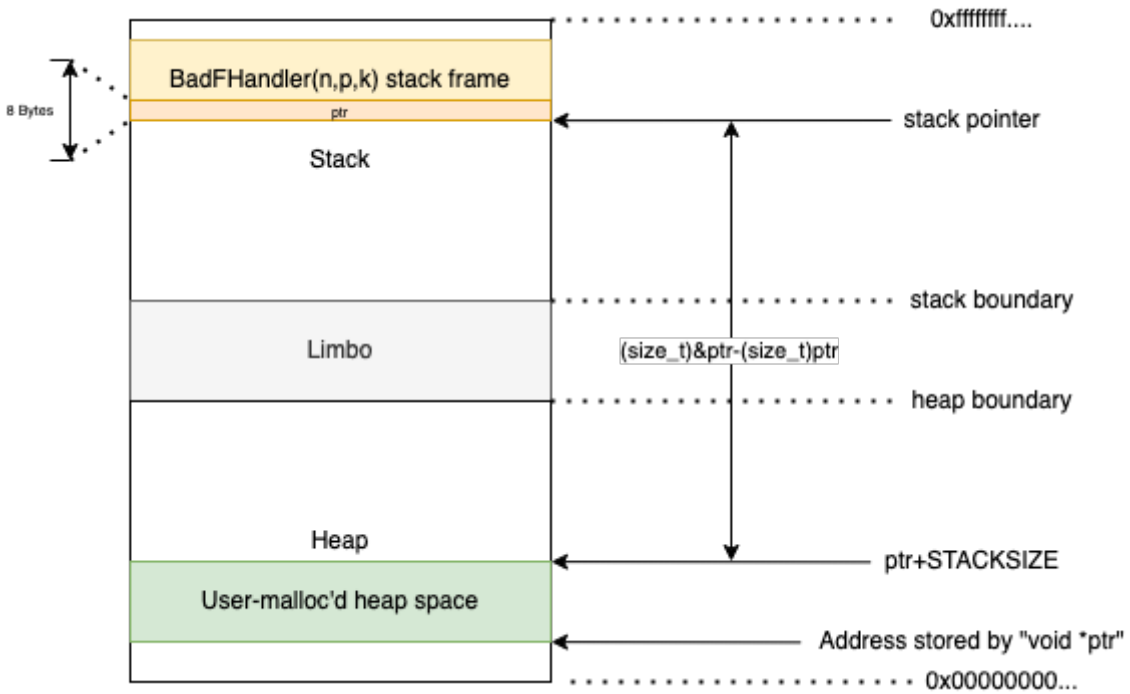
1 Solution to Part A

A stack overflow happens when the function call stack depth exceeds its limit. To mitigate such situation one might attempt to 1) reconfigure the maximum stack size limit via a system call, or 2) relocate the stack pointer to point to some bigger user-allocated memory address. You were not allowed to do the first in this problem. The solution, then, is to utilize a much larger chunk of memory somewhere else, e.g. in the *heap*, to create our own larger function call stack. To do so, we would first want to allocate a heap region that is sufficiently large to perform the deep recursion of `BadF`. Then, we would relocate the stack pointer `rsp` to point to the beginning of this region, moving our entire call stack onto the heap.

How can we do this, though? There are multiple ways. This can be done in assembly manually by moving `rsp`, but you need to ensure that you're doing the proper bookkeeping (like taking care of calling conventions). An alternative route (written in plain C) is to heap-allocate a correctly aligned pointer address using `malloc` in C, and then offset `rsp` using C99-style variable-length stack array initialization as follows:

```
#define STACKSIZE 0xffffffff
void *ptr = (char*) malloc(STACKSIZE) + STACKSIZE;
char offset[(size_t) &ptr - (size_t) ptr];
unsigned long long result = BadF(n,p,k);
```

A graphical illustration for the above code is attached next page to help your understanding.



2 Solution to Part B

There are two ingredients to our solution. First, we need to register a signal handler with the system to handle SIGSEGV signals emitted by a call stack overflow. Second, we need to resume normal control flow from the signal handler using `setjmp`/`longjmp`.

The first part is non-trivial, since when handling stack overflow-type SIGSEGV signals, the OS will *refuse* to invoke the signal handler function, since the call stack has already been blown up (so we cannot call functions on the call stack anymore). Fortunately, the `<signal.h>` header file anticipates this conundrum and provides utilities for us to register a memory region as a second, special call stack, on which the signal handler function will be called when handling errors. It thus suffices to register some user-allocated memory region that is large enough to call the signal handler function on as the second call stack.

A convenient result of this approach is that we can detect the maximum size of the stack, as well as the memory address of the "absolute top" of the stack, by poking inside the signal handler context. All this can be done entirely using functions from the C standard library, *without* using UNIX/POSIX-specific APIs provided by `sys/mman.h` such as `getrlimit` and `mprotect`. Note that *in general* it is not safe to do what we'd like to do in this situation, since the C standard mandates that any function we call during segfault handling must be reentrant-safe. Unfortunately the majority of "useful" functions, e.g. `printf` or `longjmp`, are not. A functional solution is below:

```
#define SEGV_STACK_SIZE (1 << 20)

static char RECOVERY_STACK[1 << 20]; // stack for SEGV handler
static jmp_buf RecoveryContext; // longjmp buffer
static unsigned long long N, P, K;
stack_t segv_stack; struct sigaction sa;

// Provided function
void StackOverflowException(unsigned long long n,
                            unsigned long long p, unsigned long long k);

// Provided function
unsigned long long BadF(unsigned long long n,
                        unsigned long long p, unsigned long long k);

// Signal handler for SEGV recovery
void HandleSIGSEGV(int signal, siginfo_t *si, void *arg) {
    longjmp(RecoveryContext, 1);
    // Go back to forward branch of if-statement in TestSO
}

// TestSO handles call stack blowup by registering a SIGSEGV handler
int TestSO(unsigned long long n, unsigned long long p
                                 unsigned long long k) {
    memset(&sa, 0, sizeof(struct sigaction));
    sigemptyset(&sa.sa_mask);
```


```
    sa.sa_sigaction = HandleSIGSEGV;
    sa.sa_flags = SA_SIGINFO | SA_ONSTACK | SA_RESTART | SA_NODEFER;
    sigaction(SIGSEGV, &sa, NULL);

    memset(RECOVERY_STACK, 0, 1 << 20);
    // Set up signal handler
    segv_stack.ss_sp = RECOVERY_STACK;
    segv_stack.ss_flags = 0;
    segv_stack.ss_size = SEGV_STACK_SIZE;

    // Set up the auxiliary stack
    sigaltstack(&segv_stack, NULL);

    if (!setjmp(RecoveryContext)) {
        // execute BadF for the first time
        unsigned long long r = BadF(n, p, k);
        return r; // Will return if there is no stack overflow
    }
    else {
        StackOverflowException(N, P, K);
        return 0;
    }
}

unsigned long long BadFHandler(unsigned long long n, unsigned long long p,
                                                     unsigned long long k) {
    size_t b;
    N = n; P = p; K = k;
    unsigned long long t = TestSO(n, p, k); // Test stack overflow
    return t;
}
```

Problem Statistics and Notes. Out of all contestants, fifteen teams attempted a submission to our interactive grader. Out of the fifteen that submitted, five submitted a non-trivial solution for either part A or part B. Out of the 5 people who submitted non-trivial solutions, 4 teams obtained full scores (15/15) on *both* parts of the problem. Congratulations to the teams who received a full score on both parts of the problem (listed below):

Team	Division	Submitting NetID
Joseph Xu, Kevin Huang, Jeremy Dapaah	UG	kevinhuang
Grigory Chirkov, August Ning, Marcelo Vera	Graduate	gchirkov
Xiaoqi Chen, Yuping Luo, Dingli Yu	Graduate	yupingl
Shunyu Yao, Qinshi Wang, Kaifeng Lyu	Graduate	kflyu

Grading Procedure. Our grader file is provided as `grader.c`. We compiled it together with a contestant's given input program (we also had a script that parsed the input to check for certain hazards) using the following command on the test server:

```
gcc-7 -O0 -fPIC contestant.c grader.c -oRecHell.out
```

The grader feeds a number of test cases to the function `BadFHandler`, and outputs the result to stdout. We then diff the output with the output of our official solution to see if the outputs match. The sample grader outputs for Parts A and B are provided as `part_a_out.txt`, `part_b_so_out.txt` and `part_b_out.txt`. The grader file is reusable across both Part 1 and Part 2 of this problem.

Contestant Solutions to Parts A and B. The code of top-scorers are included in `contestant_PartA` folder and `contestant_PartB` folder, labelled by the submitting NetID.

Comments on selected solutions to part A: For this part, Grigory Chirkov (grad) employed a very different approach, that is more similar to our solution to Part B. He registers a large pre-allocated memory region for the "fall back call stack" for the SIGSEGV signal handler, recovers the arguments to the last stack frame before the stack overflowed in the SIGSEGV handler using the recovery context, and continues recursion from there on the fallback stack.

Yuping Luo and Kaifeng Lyu submitted solutions using inline assembly, by manually moving rsp and calling BadF with appropriate calling conventions. (This isn't as simple as it sounds! Check out the code samples to see the details you have to pay attention to.)

Comments on selected solutions to part B: Kevin Huang's submission utilizes a very different approach that reuses the code for Part A for Part B. In doing so, he avoids the need for a signal handler, thus making his solution much shorter. He first offsets `rsp` to a large heap-allocated region in which recursion can be done correctly, just like Part A, and then he guesses a value for the stack top (relative to the `rsp` base address on the heap) and manually sets it to some sentinel value. He then calls `BadF`, which performs recursion on the heap region. The cleverness is in the fact that if the recursion is too deep, *it will destroy his sentinel value*. Hence, he can correctly "guess" if a stack overflow happened (had the recursion been on the actual call stack) just by checking if his sentinel value remains the same.