Project 1: Thread Migrator System

-Abhijeet Kumar(amk7371), Abhishek Kumar (azk6085)

Before we dive deep into the project, I just wanted to quickly describe the memory layout of a C program and touch base with the organization of the stored data inside the stack memory. In figure 1 as we can each C program has code segment, stack and heap as major segment in the memory layout. Stack grows downward and heap grows upward.

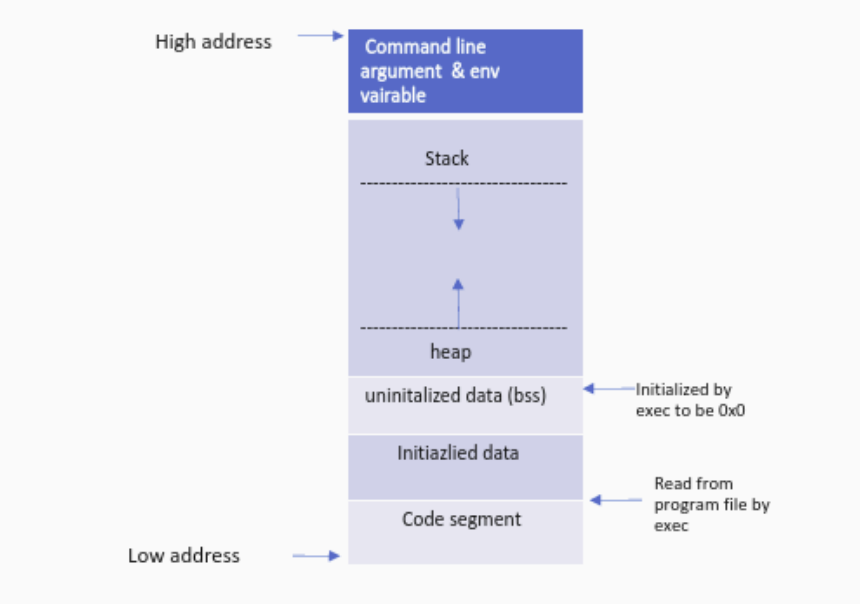


Figure 1: Memory layout of a C program

Figure 2 illustrates the stack memory of a program each function call creates a new frame In the stack memory where we store the local variable and saved EBP & other registers as well as RAS.

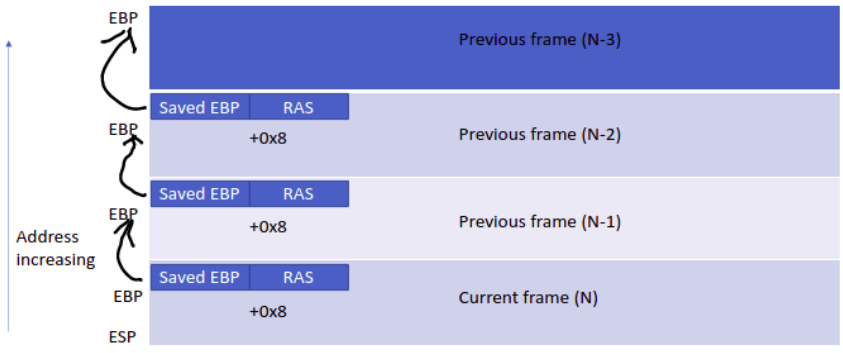


Figure 2: Stack layout of program in x86 64bit architecture

Built upon the intuition and knowledge gained from the above context, we built a thread migration library which through TCP link transfer stack frame to the server machine and set the value accordingly.

In the later part of the writeup, we briefly discuss the usage and functioning of each function calls.

psu_thread_setup_init(int mode)

This function expects the mode (0 for client; 1 for server) as an argument. And it creates the socket, initializes them and bind the socket for server mode.

psu_thread_create(void * (*user_func)(void*), void *user_args)

This function demands the function pointer & its arguments to be passed as the argument. Internally it creates the thread for client mode using pthread_create() and for server mode it listen to the socket using helper function called server_listen().

psu_thread_migrate(const char *hostname)

This function requires the host ip to be passed as the argument. In case of server mode it does nothing but in client mode, it gets current context extract Return Address (RAS), saved base pointer (EBP), from the stack and send the whole stack along with numbers of frames and each stack size to the server. In the following table (Table 1) , we try to explain the functioning of the important code snippets.

code	Description

if(getcontext(cp) == -1)
{
printf("getcontext() error\n");
exit(EXIT_FAILURE);
}
	It tries to store the current context wherever pointer cp is pointing to. If get() helper function fails it returns –1.

uint64_t _FPSP = cp->uc_mcontext.gregs[REG_EBP] + GREGS_SIZE;
uint64_t *higher_nibble_addr = (cp->uc_mcontext.gregs[REG_EBP]+ FBP_H_ADDR);
uint64_t *lower_nibble_addr = cp->uc_mcontext.gregs[REG_EBP];
uint64_t _FPBP = ((*higher_nibble_addr << 32) | *lower_nibble_addr) ;
	This code snippet tries to find the previous frame base pointer and stack pointer.
	Previous frame stack is easy to find as it current frame base pointer + 0x10 (for 64 bit). But for previous frame base pointer, we must look into the stack memory and get the saved value out of it.

uint64_t num_stack = 1;
uint64_t *ptr = _FPBP;
while(*ptr != 0x0){
ptr = *ptr;
num_stack++;
}
	For thread_create() we realize that our base pointer is 0x0. Using this code snippet, we are trying to figure out how many frames (num_stack) we need to copy before thread_create() frame begins.

Table 1: Important code snippet inside psu_thread_migrate()

server_listen()

This function listens to the socket and unpacks the received object to extract RAS, stack size, num frames and the passed stack from the client side. In its current context, it overrides the EIP register value with the RAS value received from the client and updates the current frame base pointer and stack pointer with a newly created one. In the newly created stack, I need to update the previous frame base pointer as the address changed between two machines.

code	Description
curr_context.uc_mcontext.gregs[REG_EIP] = recv_ras;	In the current context at the server side, this piece of code tries to set the received return address as EIP (next instruction pointer).

uint64_t *_prevRA = curr_context.uc_mcontext.gregs[REG_EBP] + RAS_ADDR;
uint64_t *lower_nibble_addr = (curr_context.uc_mcontext.gregs[REG_EBP]);
 	
int num_stack = 1;
for(num_stack = 1; num_stack < recv_num_stack; num_stack++){
uint64_t prev_ebp = ((char*) recv_buffer + (num_stack+1)*recv_ss + (num_stack)*GREGS_SIZE);
memcpy(recv_buffer + num_stack*recv_ss + (num_stack-1)*GREGS_SIZE, &prev_ebp, sizeof(uint64_t));
}
 	
memcpy(recv_buffer+ num_stack*recv_ss + (num_stack - 1)*GREGS_SIZE, lower_nibble_addr, sizeof(uint64_t));
memcpy(recv_buffer + num_stack*recv_ss + (num_stack - 1)*GREGS_SIZE + RAS_ADDR, _prevRA, sizeof(uint64_t));
	The first two line of code extracts the current return address and saved EBP from the curr_context. It is eventually used in the very last frame so once that returns control is passed to the prior function in stack.
	
	The for loop is used to update the previous frame ebp in the newly created stack as at the moment all addresses are from machine 1 which might not be correct anymore.

curr_context.uc_mcontext.gregs[REG_EBP] = recv_buffer + recv_ss;
curr_context.uc_mcontext.gregs[REG_ESP] = &recv_buffer;
setcontext(&curr_context);
	This code snippet updates the EBP and ESP with newly created stack and calls setcontext().
	
	As EBP is higer address thus pointing it to the last address whereas ESP points to the first address of the buffer.

Table 2: description of each code inside server_listen()

Challenges: -

In this project one major challenge and most important was dealing with the consequences of updating the stack on the server side. It is most important as if any of the saved register value accessed inside the function points to the address from machine1, it will raise a segfault. We started with the idea of making use of /proc/self/maps to understand the mmap of the program and later after reading the manual of {set,get}-context and going through multiple gdb sessions to understand the stack memory better we came up with a final solution. Stepping into multiple gdb session and investigating each and very location in the vicinity of ESP and EBP and try to map it to disassemble part of the C function (to ensure what all registers are accessed) helped us overcome the challenges around stack migration.

Migrating stack became more challenging when it involves sending stack memory of multiple frames through TCP link (recursive calls). We backtracked through each function frame to come up with a scalable solution (number of recursive calls can be large number too) which involves an optimal way to copying all the stack frame and pass it over to the server (in table 1, row 3).

Ownership-

Abhijeet & Abhishek employed pair programming model for thread migration project. We both brainstormed the problem together on a whiteboard so that we discuss what is the most optimal way of solving the problem and point out all the showstopper bugs which we could foresee at the start. After investing a considerable amount of time in understanding the problem statement and brainstorming, Abhishek took the full ownership (coding & debugging) of the socket programming and APP1 which basically enables thread migration milestone whereas Abhijeet took the full ownership (coding & debugging) of APP2 which involves stack migration and setting up the previous frame base pointer correctly. Since APP3 was tricky to answer as it involves multiple frames and we wanted to have a scalable solution with no hardcoding we both debugged the APP3 together and came up with a very simple and yet very effective solution to address it.

Command Line -

./app < ip address > <mode = {0: client}, {1: server}>

Example

Server: -

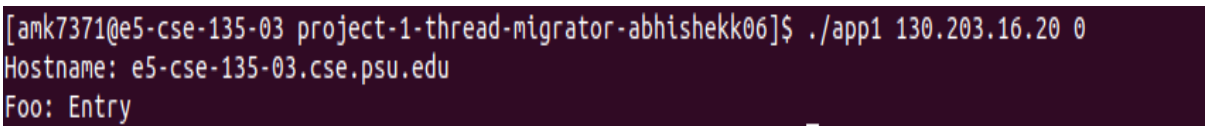
./app1 130.203.16.21 1

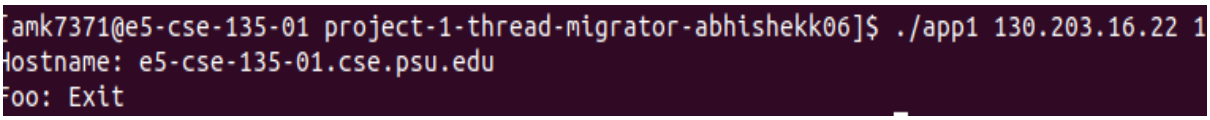
Client: -

./app1 130.203.16.20 0

Sample output-

For APP 1-





For APP2-

```
[amk7371@e5-cse-135-03 project-1-thread-migrator-abhishekk06]$ ./app2 130.203.16.20 0
Hostname: e5-cse-135-03.cse.psu.edu
Foo: Value of A = 10
```

```
[amk7371@e5-cse-135-01 project-1-thread-migrator-abhishekk06]$ ./app2 130.203.16.22 1
Hostname: e5-cse-135-01.cse.psu.edu
Foo:Value of A = 10
```

For APP3-

```
[amk7371@e5-cse-135-03 project-1-thread-migrator-abhishekk06]$ ./app3 130.203.16.20 0
Hostname: e5-cse-135-03.cse.psu.edu
Bar: Value of A = 1
Hostname: e5-cse-135-03.cse.psu.edu
Bar: Value of A = 2
Hostname: e5-cse-135-03.cse.psu.edu
Bar: Value of A = 3
```

```
[amk7371@e5-cse-135-01 project-1-thread-migrator-abhishekk06]$ ./app3 130.203.16.22 1
Hostname: e5-cse-135-01.cse.psu.edu
Bar: Value of A = 4
Hostname: e5-cse-135-01.cse.psu.edu
Bar: Value of A = 5
Hostname: e5-cse-135-01.cse.psu.edu
Bar: Exit Value of A = 5
Hostname: e5-cse-135-01.cse.psu.edu
Bar: Exit Value of A = 4
Hostname: e5-cse-135-01.cse.psu.edu
Bar: Exit Value of A = 3
Hostname: e5-cse-135-01.cse.psu.edu
Bar: Exit Value of A = 2
Hostname: e5-cse-135-01.cse.psu.edu
Bar: Exit Value of A = 1
```