CSE 511 Project 1:
Thread Migrator System

Prepared by:

Mahmudur Hera – mbr5797@psu.edu
A. Burak Gulhan – abg6029@psu.edu

Implementation Logic

In this program all the assignment deliverables, struct psu_thread_info, psu_thread_setup(), psu_thread_create() psu_thread_migrate() and were implemented. Three helper functions server(), server_wrapper() and client() were also implemented. These are briefly explained below.

The most important part in this implementation is sending the correct information over to the server. This information is the thread context, which contains registers (EBP, ESP, EIP) and the stack of the client. To copy the stack frames, EBP is accessed several times in subsequently deeper frames in order to get the previous frames up to a predefined recursion depth. The indexes of the EBP values are also stored. These values are then sent to the server.

Since the server and client stacks don't start at the same memory values, addresses inside of the received stack must be corrected. This is done using the EBP index values we previously stored. The bottommost frame in the received stack is corrected to show EBP of the server, the second from the bottom frame's EBP is corrected to show the bottommost frame's EBP and so on. Then ESP is set to the top of the received stack and EIP is recovered from the received information. Finally, the context of the thread is set.

struct psu_thread_info

This struct is passed through the socket from client to server. This struct contains information about the thread including context, stack, base pointers, etc.

```
typedef struct psu_thread_info {

    ucontext_t uctx;
    uint32_t stack_raw_data[SIZE_OF_STACK];
    //uint32_t args[100];
    uint32_t restart_point;
    int num_elements;
    int restart_point_offset;
    int num_blank_entries;
    int base_pointer_positions[MAX_RECURSION_DEPTH];
    int num_frames;

} psu_thread_info_t;
```

uctx stores the value of getcontext()

stack_raw_data[] stores the stack of the client up to MAX_RECURSION_DEPTH frames.

restart_point stores the address where the server should resume thread execution at.

num_elements stores the size of *stack_raw_data.*

restart_point_offset stores the offset for *stack_raw_data* in the case where instruction addresses are different on server and client. This is added to *restart_point.*

num_blank_entries keeps the amount of entries that will not be written to in *stack_raw_data*

base_pointer_positions[] keeps the indexes of *stack_raw_data*[] entries that contain base pointers.

num_frames shows the amount of frames copied to stack.

void psu_thread_setup_init(int mode)

This function initializes the global variable *server_mode* as the input parameter mode, and the global struct member *psuthreadinfo.num_blank_entries* of type *psu_thread_info*

int psu_thread_create(psu_thread_info_t* t_info, void* (*start_routine)(void*), void* args)

This function makes a new thread using pthread_create() This function first gets a local context using getcontext() and stores the EBP to the global variable *base_pointer_create.* This is used to prevent going past the *main* function when handling recursive thread calls. Depending on whether the caller of psu_thread_create() is a server or client (by checking the global variable *server_mode*) pthread_create() is called either with server_wrapper() in which the server is started and the socket initialized, or pthread_create is called with the *user_func* parameter.

int psu_thread_migrate(char* node)

This function fills out all members in *psu_thread_info_t* of type struct *psu_thread_info.* At the end of the function, another function client() is called. Here the client opens a socket and connects to the server and *psu_thread_info_t* is sent to this server.

void server()

This function opens a socket and waits for the thread information (of type struct psu_thread_info) to arrive from the client. Once received, the EBP values in the received stack are corrected and the last frame's return address, EBP, ESP, and EIP are set correctly. Then setcontext() is called with this received context.

void server_wrapper()

Calls server() then pthread_exit(). psu_thread_create() makes a thread in this function if a server is calling it.

void client(const char * hostname)

This function opens a socket and connects to *hostname* (the server). In this function the previously initialized *psu_thread_info_t* structure is sent to the client.

Difficulties Faced

The first and biggest difficulty we faced was learning what the context functions were doing. We originally thought that we could just send the results of getcontext() to the server and resume using setcontext(), but this wasn't the case. We had to read the man pages of these functions and then read the header files of *ucontext* to see what these functions were doing, what values they were storing and to access the registers stored in this structure. Compiling with the -E flag was very helpful in this case.

The next difficulty was sending the correct values to the server and being able to resume the thread using these values. For this we had to refresh our knowledge on how function calls are made on the stack, what the EBP, ESP, EIP registers are used for and how variables are stored in the stack. However, learning these wasn't enough, we also had to deal with many segfaults. The method of debugging using print statements was insufficient for these and we had to learn how to use gdb. One such issue we faced was not allocating enough space for the stack and then getting segfaults from seemingly unrelated parts of the code, such as a line with printf().

The last difficulty we faced was sending multiple frames over to the server and resuming correctly for recursive calls, such as in app3. The issue here was that the EBP values in the client stack were pointing to different locations in the server stack, since the stacks are initialized at different locations every time. We solved this issue by marking which lines contained EBP and then iteratively setting them to the correct locations in the server.

Special Cases and Extra work

We have made some hard coded limits in our program. These are:

- A maximum of 20 arguments
- A maximum of 1400 entries in the stack
- A maximum recursion depth of 20

We also assume that the thread won't overflow the stack and that this program will be ran on the W135 Lab machines.

While these limits and assumptions are met, our program should work for all cases. These limits can also be changed in the program before compiling, if they are insufficient.

Distribution of Work

Mahmudur Hera: Implementing sockets, sending/receiving threads

Burak Gulhan: Writing report, sending/receiving threads