KV5002: Computer Networks, Security and Operating Systems Report

By Andrew Nattress
andrew.nattress@northumbria.ac.uk
W17009550

Contents

Lunar Lander Controller

Software Architecture

My controller is built to run constantly on two threads, I use one thread for receiving user inputs and the other to update the user dashboard. I then have 2 other threads which are created whenever needed, one is used to send and receive Lunar Lander server messages and the other is used to log inputs and the status of the lander to the data log file.

The decision to have the user input and dashboard update on threads of their own came as this allows both to run concurrently, without having to wait through the others delays, for example the dashboard update method does not have to wait for a user input in order to send the update message. I think this is a good implementation of both methods as they are simple, easy to understand and function well. Both threads are created after the server address is checked for errors and the data log file is emptied.

The get input thread uses begins with a while loop, which lasts until the program is terminated. The while loop scans for user input then a switch statement decides what should be done with the input. If the input is a control for the lunar lander a new thread is started in order to send the server command required. Before the thread is created a semaphore checks that the thread is not already in use, if it is the command has to wait until the thread is ready. This thread then sends the command, receives the response and then joins the thread back to the get input thread. Next a thread is started to log the users input to the data log file. Again the program uses a semaphore to first check if the logging thread is in use, if it isn't the operation proceeds.

The logging thread first prints the passed parameter from the function which states the control key which was pressed, for example throttle up. Next the function requests the state and terrain from the server, as it always does the server request uses its own thread, checking if anything is using that thread and waiting for it to finish. The condition is also logged, however since the dashboard retrieves that every 0.1s I stored that in a variable and use the most recent condition in order to place slightly less stress on the server. This information is all appended onto the end of the data log file.

The dashboard runs on its own thread which was created at the same time as the get input thread, again it runs a while loop until the program is stopped. The while loop checks if the server thread is in use, using a semaphore, then requests the condition from the lunar lander server. Next it sends the condition to the dashboard, updating it. Finally, I use the usleep command to delay the next loop from occurring for 0.1s.

I decided to handle user input using scanf as is it a built in method that works, however I considered using the ncurses library to detect if the key is held down. I decided against it in case the assignment did not allow different packages. This meant that each input requires an enter press afterwards and because I used a switch statement each control input had to be a single character. I found, however, that this worked very well for gameplay and was easy to adjust to.

Dashboard Communication

The protocol I chose to implement, for handling the dashboard communication was very simple. I figured out that from C I could pass on the returned condition string to the dashboard, with minimal changes. The only change I make to the string is to replace any “%” symbols with an empty space. This allows the Java program to understand the string without any modification, and works without any problems.

I did not add any error checking to make sure that the information had been transmitted accurately, which might be a problem when transmitting via UDP, but only when over sending the messages over long distances. I ultimately decided that it was unnecessary as the messages are sent so rapidly that an error would be hardly noticeable. However, had I decided to do it I would have included a checksum check, in order to verify that the message was transmitted accurately.

Data Logging

I decided to log the state, condition, and terrain information every time the player controls the lander. I decided all of this information was important as no single factor lead to the lander crashing, it was all of them. I decided to log on every key press as the specification declared that the data log should record the user input. I also decided to log everything one final time when the lander lands or crashes, this will let the player see the landers details at the moment of impact, which will be important in determining the cause of the outcome. This method of logging works very well for file sizes as it limits the amount of writes to a number small enough to never take up too much space, the average file size for me was 19KB, meaning that even if the player completed 1000 attempts in one sitting the file would only be 19MB. I determined this by playing the game 30 times and calculating the average. I also clear the file every time the game is launched, this stops the game from continuously writing to the same file and taking up too much room on the hard drive. I chose to log my data in a “.txt” file as it means that the user can easily open it and review their data.

OS Theory and Concepts

Hardware Abstraction Layer

The hardware abstraction layer is a layer of programming, between the hardware and the software of a computer, which enables an operating system to communicate with a hardware device (Rouse, 2005). The operating system's job is to make the hardware easy and simple to use for high level (application) programmers. It does this by providing an abstract view of the hardware that is standardised so that programs can interact with any version of the same type of hardware without requiring a specific implementation.

An example of this is that most PC operating systems, such as Linux, provide a driver for SATA port drives. This driver allows the computer to utilise hardware connected to its SATA ports, without needing to understand the fundamental workings of each specific type of SATA drive as discussed in a lecture by (Kendall, 2019). This enables the file system to be viewed in high level systems such as windows explorer easily. The hardware abstraction layer can be accessed either via the operating system's kernel or from a device specific driver that is sometimes used by a piece of hardware such as expensive computer mice or printers. In either case, the program's interactions with the hardware device are more general and simpler than they would otherwise be (Rouse, 2005).

Another example is gamepad controllers, there are many types of controller that work on windows PCs. The hardware abstraction layer is used to take inputs from these controllers, standardise the data sent from the joysticks and buttons. This allows programmers to capture the inputs, from many types of controller, in a single format, making their development process clean and simple as explained by (Eizikovich, 2016).

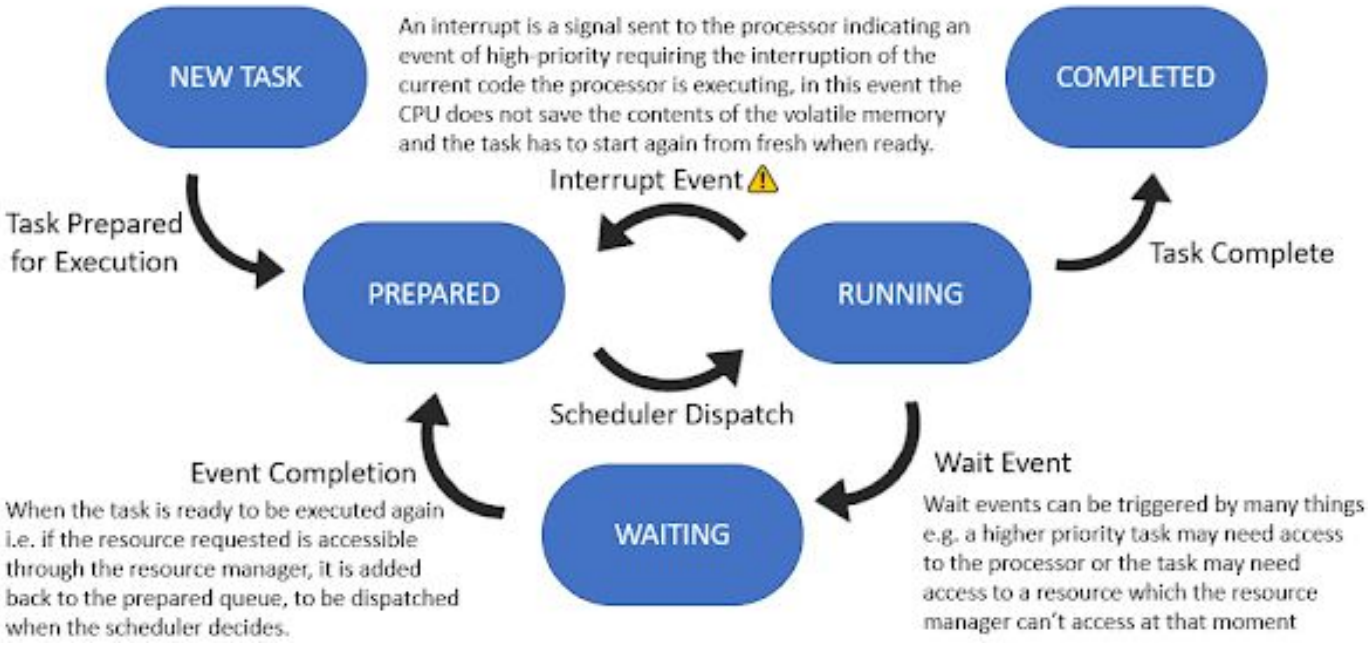
Resource Manager

Resource management is another key function of a computer's operating system. It is required to systematically allocate access to the many components of a computer, such as the processor, memory, network interfaces. The efficiency of the resource manager is very important to the speed and functionality of a computer, as the correct prioritisation of tasks keeps a computer running efficiently. The operating system needs to allocate resources so that any of the processes running simultaneously on the machine do not interfere with each other (Kendall, 2019).

As discussed by (Tanenbaum and Bos, 2014) there are two types of resource management mechanisms; time multiplexing and space multiplexing. When a resource is time multiplexed, processes queue to use it one at a time. An example of this is in the CPU, only one process can be active on each core at any time. This means that the resource manager has to limit all of the active tasks' access to the CPU. Determining which process should go next is the task of the operating system, different tasks have different priorities and a good resource manager will make sure that the high priority tasks get access to the processor as quickly as possible, while making sure low priority tasks still get an opportunity to progress. Space multiplexing instead shares the resources, allowing each process to access part of the resource it needs.

A good example of this is memory, which is divided up amongst all of the active processes by the resource manager, normally a process is allowed as much memory as required, but if more memory is demanded than is available in the system it is allocated to higher priority tasks, and tasks that are low priority are left with less than they requested or sometimes paged into storage until memory is freed up.

Context Switching



In most computers, or other modern hardware such as phones, there will be lots of processes active at the same time. One will be running on each core of the computer's processor (in a single threaded CPU, in most modern desktop CPUs more than one thread can run simultaneously due to the development of multithreading technology), the others will be queued within the scheduler. The scheduler performs context switches periodically to allow each task its chance to perform its operation, this is triggered when the threads quantum runs out. A quantum is the maximum amount of time a process is allowed to run before being switched out for another thread of similar priority. Usually this switch is fast, making it appear as though the processes are running simultaneously (Brockway, 2019).

However, on many occasions a process can require access to an additional resource which is not readily available, for example access to network I/O or a hard disk drive. If the resource manager cannot grant access to this quickly, the process is paused and a context switch occurs. The scheduler saves the context of the current thread into a waiting state, where it waits for the resource to become available, while it waits the next process in the queue is loaded into the processor and worked through (Neville-Neil and McKusick, 2005). While the process of context switching is expensive, due to the volatile data like registers, the program counter and memory data having to be transferred into storage, it can speed up computers as less time is spent waiting for access to resources and it is a vital process in any multitasking machine.

A context switch can also be caused hardware interrupt. A hardware interrupt is a signal from a hardware device, for example a mouse or keyboard, to the operating system kernel that an event has occurred, which can be anything from a key press to network data arrival as discussed by (The Linux Information Project, 2006).

Secure Socket Layer

Secure Socket Layer (SSL) is a widely used security protocol, supported by almost all web browsers and major operating systems. It has become the security standard for establishing an encrypted link between a web server and a browser. It provides authentication, confidentiality and integrity to internet users (Kurose and Ross, 2016).

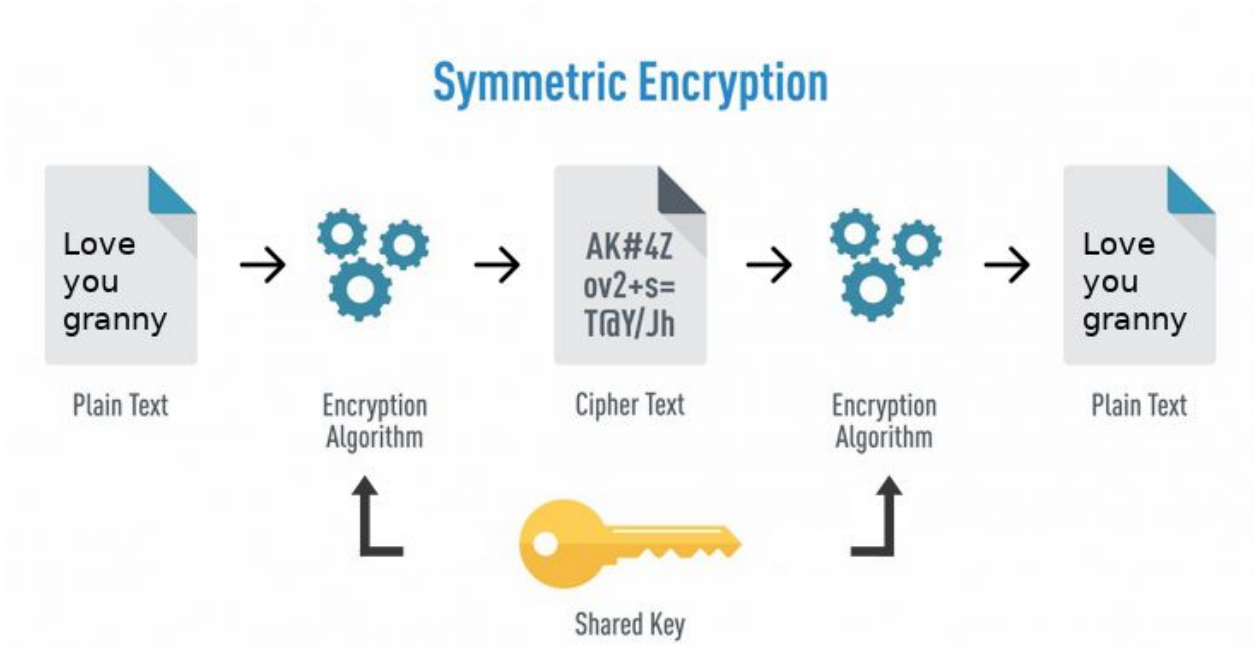
When authenticating the server, the client uses the server's public key to encrypt the data that is used to compute the secret key. The server can then decrypt the message including the secret key only if it can decrypt that data with its private key. For client authentication, the server uses the public key in the client's certificate to decrypt the data the client sends during the handshake. The exchange of finished messages that are encrypted with the secret key confirms that authentication is complete as stated by (IBM Knowledge Center, 2019a).

SSL uses a combination of symmetric and asymmetric encryption to ensure message privacy and confidentiality. During the handshake, the SSL client and server agree an encryption algorithm and a shared secret key to be used for one session only. All messages sent between the client and server, using SSL, are encrypted using the chosen algorithm and key, ensuring that the message remains private even if it is intercepted (IBM Knowledge Center, 2019a).

SSL can confirm the integrity of any sent data by calculating a message digest. The message digest is a fixed size numeric representation of the contents of a message, computed by a hash function. This message digest can then be encrypted, and used as a digital signature on every message sent (IBM Knowledge Center, 2019b).

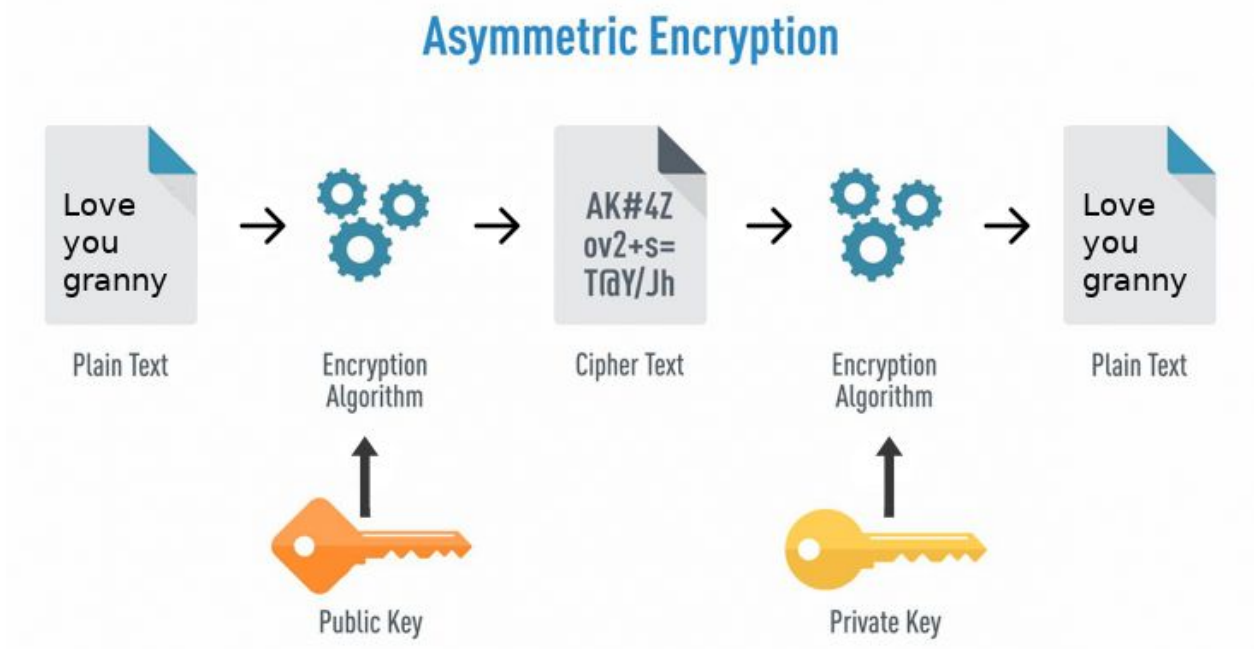
Secure Socket Layer Cryptography

SSL encrypts sensitive information so that only the intended recipient can access it, confirming that all data that is sent is protected. This is important because the information you send on the internet is passed across many computers in order to reach the destination server. Without SSL's encryption any computer between yours and the server would be able to read your private data including; your credit card details, passwords, and other sensitive information. When SSL encryption is used, the data becomes unreadable to anyone except for the server you are sending the information to as detailed by (SSL Shopper, n.d.). There are 2 types of SSL encryption, symmetric and asymmetric. Asymmetric keys are bigger than symmetric keys, meaning the data that is encrypted is tougher to crack. However, this does not mean that asymmetric encryption is the only type that should be used.



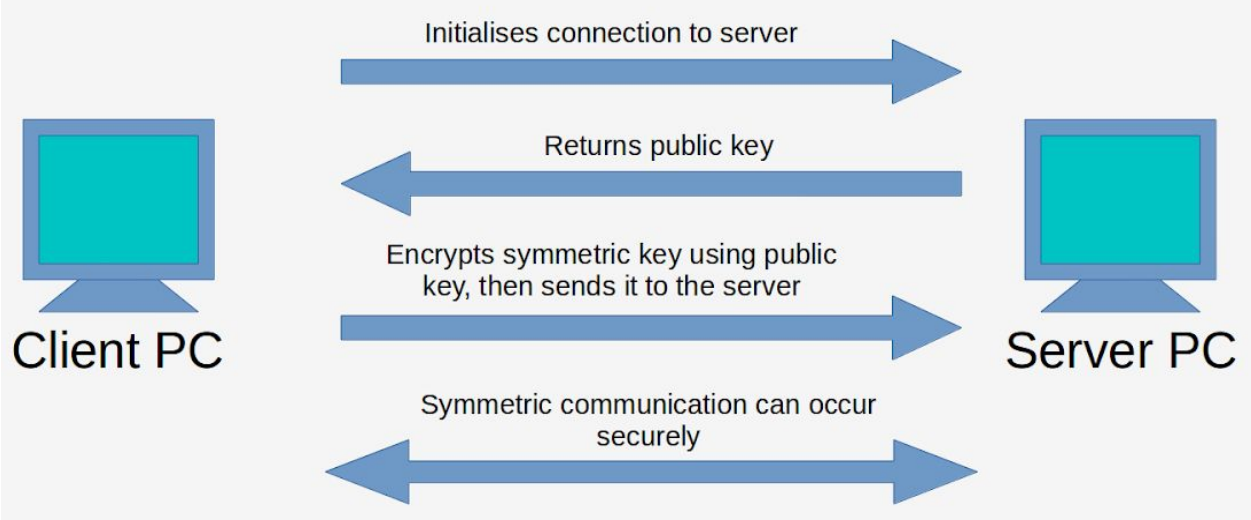
Symmetric Encryption Diagram (Almeida, 2017).

As described by (Digicert, n.d.), symmetric keys require less computational burden to encrypt and decrypt and the same key is used for symmetric encryption and decryption. This makes symmetric encryption significantly quicker for communication. Due to this, both the client and the server need the key. This means that, if you need to send the key, serious security risks arise as if the key is intercepted anyone with the key can decrypt the data. For this reason, whenever a symmetric key is sent over a network is is usually encrypted using asymmetric encryption first.



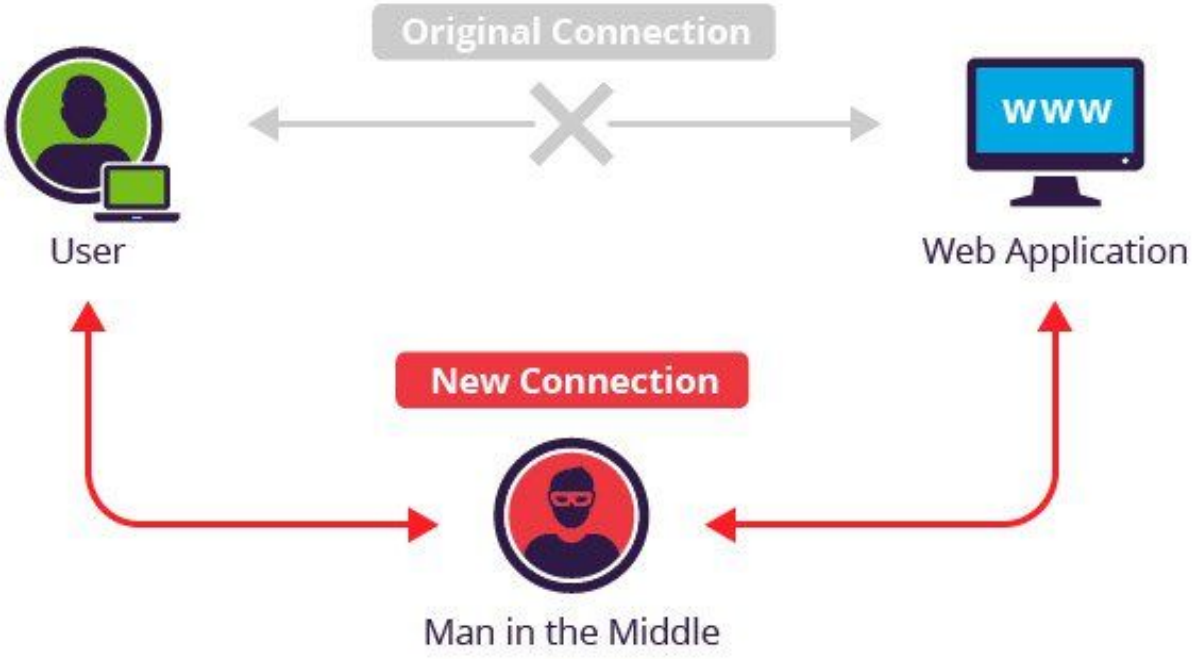
Asymmetric Encryption Diagram (Almeida, 2017).

When using asymmetric encryption key security is no longer a concern. As long as the private key is kept secret, no one can decrypt the messages except the intended recipient. Your public key can be sent to anyone, without worrying about any security risk. Anyone who has the public key can encrypt data, but only the machine with the private key can decrypt it.



Site that utilise SSL usually use a combination of symmetric and asymmetric encryption. Asymmetric encryption is used to make the initial connection. When the client computer initialises a connection to the server, the server returns its public key. Then the client generates a symmetric encryption based session, encrypts the key using the server's public key and sends it back to the server. The server now decrypts the received key using its private asymmetric key and uses it to communicate with the client in a secure symmetric encryption session as described by (Digicert, n.d.).

Man-in-the-Middle Attacks



Man in the Middle Attack Diagram (Imperva, n.d.).

A man-in-the-middle attack is when the communication between two systems is intercepted by a third party. The man-in-the-middle attack can intercept the public keys that are exchanged during the SSL handshake and switch them with their own, as described by (Oracle, 2010). This makes it appear to the client and server that each is in contact with the other. This allows the man-in-the-middle to change any information passed between the client and server. However, in this case, he cannot read any of the data sent.

SSL can be vulnerable to man-in-the-middle attacks, and according to (Mutton, 2016) 95% of HTTPS servers are vulnerable to trivial attacks. This, however, is usually due to the server host's ignorance and not any security vulnerability in SSL. Man-in-the-middle vulnerabilities on SSL secured sites are usually caused by one of SSL's security preconditions being broken. An example of this would be if the server's private key decryption key is stolen, the attacker can pretend to be the server without the client ever knowing. Another cause of SSL vulnerability is when server certificates are not kept up to date or are improperly configured, for example, according to (Taylor, 2018) a large number of SSL are susceptible to attacks such as the DROWN attack. This attack allows any communication between users and the server to be read by the attacker, including passwords and credit card details. As of 2016 as many as 33% of SSL secured servers were vulnerable to the attack, (Drown Attack Team, 2019).

Finally, SSL stripping can be used to intercept packets of data during transmission between the client and server. The attacker alters the SSL secured address requests to the HTTP equivalent of the page, which forces the host to transmit data unencrypted. In order to gain access to the original HTTPS request the hacker must establish a connection to the client's PC through its firewall (if enabled) then they can see and modify the request before it is sent. When modified to HTTP sensitive information can be read as it is transmitted in plain text. To prevent this, servers can force the use of HTTPS when connected, this means that any data transmitted is always encrypted. A similar system can be employed by client computers, which is particularly useful for web users who often access HTTP based sites, as most modern browsers support plugins which enforce HTTPS on all requests to servers, as discussed by (Rapid7, n.d.).

References

Rouse, M. (2005) *Hardware Abstraction Layer (HAL)*. Available at: https://whatis.techtarget.com/definition/hardware-abstraction-layer-HAL (Accessed: 08/05/19).

Kendall, D. (2019) *Computer Networks, Security, and Operating Systems [Lecture]*, KV5002. Northumbria University. 28/01.

Eizikovich, S. (2016) *Standard versus Virtual Joystick*. Available at: http://vjoystick.sourceforge.net/site/index.php/dev216/112-system-architecture216 (Accessed: 08/05/19).

Tanenbaum, A. S. and Bos, H. (2014) *Modern Operating Systems*. New Jersey: Pearson. Volume 4.

Brockway, M. (2019) *Operating Systems & Concurrency: Process Concepts [Lecture]*, KV5002. Northumbria University. 28/01.

Neville-Neil, G. V. and McKusick, M. K. (2005) *FreeBSD Process Management*. Available at: http://www.informit.com/articles/article.aspx?p=366888&seqNum=3 (Accessed: 09/05/19).

The Linux Information Project (2006) *Context Switch Definition*. Available at: http://www.linfo.org/context_switch.html (Accessed: 09/05/19).

Kurose, J. and Ross, K. (2016) Computer Networking: A Top Down Approach. New Jersey: Pearson. 7th Edition.

IBM Knowledge Center (2019a) *How SSL and TLS provide identification, authentication, confidentiality, and integrity.* Available at: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.sec.doc/q009940_.htm (Accessed: 10/05/19)

IBM Knowledge Center (2019b) *Message digests and digital signatures.* Available at: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10510_.htm (Accessed: 10/05/19)

Almeida, R. (2017) *Symmetric and Asymmetric Encryption.* Available at: https://hackernoon.com/symmetric-and-asymmetric-encryption-5122f9ec65b1 (Accessed: 11/05/19)

SSL Shopper (no date) *Why SSL? The Purpose of using SSL Certificates.* Available at: https://www.sslshopper.com/why-ssl-the-purpose-of-using-ssl-certificates.html (Accessed: 12/05/19)

Digicert (no date) *Behind the Scenes of SSL Cryptography. Available at:* https://www.digicert.com/ssl-cryptography.htm (Accessed: 12/05/19)

Imperva (no date) *Man in the middle (MITM) attack.* Available at: https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/ (Accessed: 14/05/19)

Oracle (2010) *Man-In-the-Middle Attack.* Available at: https://docs.oracle.com/cd/E19656-01/821-1507/aakhd/index.html (Accessed: 12/05/19)

Mutton, P. (2016) *95% of HTTPS servers vulnerable to trivial MITM attack.* Available at: https://news.netcraft.com/archives/2016/03/17/95-of-https-servers-vulnerable-to-trivial-mitm-attacks.html (Accessed: 12/05/19)

Taylor, D. (2018) *HTTPS & SSL Does Not Mean You Have a Secure Website.* Available at: https://www.semrush.com/blog/https-a-modern-false-sense-of-security/ (Accessed: 13/05/19)

Drown Attack Team (2019) *The DROWN Attack.* Available at: https://drownattack.com/ (Accessed: 13/05/19)

Rapid7 (no date) *Man-in-the-Middle (MITM) Attacks.* Available at: https://www.rapid7.com/fundamentals/man-in-the-middle-attacks/ (Accessed: 14/05/19)

Appendices

Controller.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#include <pthread.h>
#include <semaphore.h>
#include <assert.h>

#include <netinet/in.h>
#include <arpa/inet.h>

#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <stdbool.h>

#define BUFF_SIZE 4096

FILE *out_file;
int throttle = 0;

char condition[BUFF_SIZE];
char terrain[BUFF_SIZE];
char state[BUFF_SIZE];
struct addrinfo *address;
struct addrinfo *address2;
int fd;

bool isActive = true;

pthread_t logThread, serverCommandThread;
int rc3, rc4;

sem_t logSemaphore;
sem_t serverSemaphore;

void *sendCommand(void *arg);
void *getState(void *arg);
void *getTerrain(void *arg);
void *getCondition(void *arg);

void logCommand(char msg[BUFF_SIZE]);
```

```
//Logs data into a text file
void *logData(void *arg){
  char *msg = (char*)arg;
  out_file = fopen("DataLog.txt","a+");

  fprintf(out_file, "%s\n", msg);

  sem_wait(&serverSemaphore);
     rc4 = pthread_create( &serverCommandThread, NULL, getState, NULL);
     assert(rc4 == 0);
     pthread_join(serverCommandThread, NULL);
  sem_post(&serverSemaphore);

  fprintf(out_file, "%s\n", state);

  fprintf(out_file, "%s\n", condition);

  sem_wait(&serverSemaphore);
     rc4 = pthread_create( &serverCommandThread, NULL, getTerrain, NULL);
     assert(rc4 == 0);
     pthread_join(serverCommandThread, NULL);
  sem_post(&serverSemaphore);

  fprintf(out_file, "%s\n", terrain);

  fclose(out_file);

  return 0;
}

//gets input and sends prepares commands for the server
void *getInput(void *arg){
            char input;
            while (isActive){
            char outgoing[BUFF_SIZE];
            scanf("%s", &input);

    switch (input){
            case 'a':
             if(throttle <= 90)
                        throttle += 10;
             sprintf(outgoing, "command:!\nmain-engine:%d", throttle);
             sem_wait(&serverSemaphore);
                        rc4 = pthread_create( &serverCommandThread, NULL, sendCommand, (void *)
&outgoing);
                        assert(rc4 == 0);
                        pthread_join(serverCommandThread, NULL);
                        sem_post(&serverSemaphore);
             logCommand("throttle up \n");
             printf("throttle: %d \n", throttle);
             break;
```

```
          case 's':
           if(throttle >= 10)
                    throttle -= 10;
           sprintf(outgoing, "command:!\nmain-engine:%d", throttle);
           sem_wait(&serverSemaphore);
                    rc4 = pthread_create( &serverCommandThread, NULL, sendCommand, (void *)
&outgoing);
                    assert(rc4 == 0);
                    pthread_join(serverCommandThread, NULL);
                    sem_post(&serverSemaphore);
           logCommand("throttle down \n");
           printf("throttle: %d \n", throttle);
           break;
          case 'q':
           printf("spin left \n");
           strcpy(outgoing, "command:!\nrcs-roll:-0.5");
           sem_wait(&serverSemaphore);
                    rc4 = pthread_create( &serverCommandThread, NULL, sendCommand, (void *)
&outgoing);
                    assert(rc4 == 0);
                    pthread_join(serverCommandThread, NULL);
                    sem_post(&serverSemaphore);
           logCommand("spin left \n");
           break;
          case 'w':
           printf("stop turning \n");
           strcpy(outgoing, "command:!\nrcs-roll:0");
           sem_wait(&serverSemaphore);
                    rc4 = pthread_create( &serverCommandThread, NULL, sendCommand, (void *)
&outgoing);
                    assert(rc4 == 0);
                    pthread_join(serverCommandThread, NULL);
                    sem_post(&serverSemaphore);
           logCommand("stop spinning \n");
           break;
          case 'e':
           printf("turn right \n");
           strcpy(outgoing, "command:!\nrcs-roll:0.5");
           sem_wait(&serverSemaphore);
                    rc4 = pthread_create( &serverCommandThread, NULL, sendCommand, (void *)
&outgoing);
                    assert(rc4 == 0);
                    pthread_join(serverCommandThread, NULL);
                    sem_post(&serverSemaphore);
           logCommand("spin right \n");
           break;
          case 'r':
           sem_wait(&serverSemaphore);

           rc4 = pthread_create( &serverCommandThread, NULL, getState, NULL);
           assert(rc4 == 0);
           pthread_join(serverCommandThread, NULL);
```

```
                    sem_post(&serverSemaphore);
            printf("%s", state);
            break;
           case 'd':
            sem_wait(&serverSemaphore);
            rc4 = pthread_create( &serverCommandThread, NULL, getTerrain, NULL);
            assert(rc4 == 0);
            pthread_join(serverCommandThread, NULL);

                    sem_post(&serverSemaphore);

            printf("%s", terrain);
            break;
           case 'l':
            isActive = false;
            printf("exiting...\n");
            break;
           case 'h':
            printf("a: throttle up\n");
            printf("s: throttle down\n");
            printf("q: spin left\n");
            printf("w: stop spinning\n");
            printf("e: spin right\n");
            printf("r: get the current state of the lander \n");
            printf("d: get terrain details beneath the lander \n");
           default:
            printf("invalid input, enter h for controls. \n");
            break;
    }
          }
          pthread_exit(NULL);
}
//Updates dashboard every 0.1s
void *updateDash(void *arg){
  while(isActive){
    sem_wait(&serverSemaphore);
    rc4 = pthread_create( &serverCommandThread, NULL, getCondition, NULL);
      assert(rc4 == 0);
      pthread_join(serverCommandThread, NULL);
    sem_post(&serverSemaphore);
    char cond[BUFF_SIZE];
    strcpy(cond, condition);
    //removes % sign which is causing parsing error in Java file
    for (int i = 0; cond[i] != '\0'; i++)
    {
           if (cond[i] == '%')
                    cond[i] = ' ';
    }
          sendto(fd, cond, strlen(cond), 0, address2->ai_addr, address2->ai_addrlen);
                    usleep (100000);
  }
  return 0;
}
```

```
//initialises and closes the program
int main ( int argc, char *argv[] )
{
	char *host = "127.0.1.1";
	char *port = "65200";
	char *port2 = "65250";

	out_file = fopen("DataLog.txt", "w");

	if (out_file == NULL)
	{
   printf("Error! Could not open file\n");
	}
	//empties file on start
	fclose(out_file);

	pthread_t inputThread, updateDashThread;

	int rc1, rc2;

	sem_init(&serverSemaphore, 0, 1);
	sem_init(&logSemaphore, 0, 1);
	const struct addrinfo hints = {
	.ai_family = AF_INET,
	.ai_socktype = SOCK_DGRAM,
	};
	const struct addrinfo hints2 = {
	.ai_family = AF_INET,
	.ai_socktype = SOCK_DGRAM,
	};
	int err, err2;
	err = getaddrinfo( host, port, &hints, &address);
	if (err) {
	fprintf(stderr, "Error getting address: %s\n", gai_strerror(err));
	exit(1);
	}
	err2 = getaddrinfo( host, port2, &hints2, &address2);
	if (err) {
	fprintf(stderr, "Error getting address: %s\n", gai_strerror(err2));
	exit(1);
	}
	fd = socket(AF_INET, SOCK_DGRAM, 0);
	if (fd == -1) {
	fprintf(stderr, "error making socket: %s\n", strerror(errno));
	exit(1);
	}

	rc1 = pthread_create( &inputThread, NULL, getInput, NULL);
	rc2 = pthread_create( &updateDashThread, NULL, updateDash, NULL);


	assert(rc1 == 0);
```

```
        pthread_join(inputThread, NULL);
        assert(rc2 == 0);
        pthread_join(updateDashThread, NULL);

        sem_destroy(&serverSemaphore);
        sem_destroy(&logSemaphore);
}

//gets the state of the lander and stores it in a variable
void *getState(void *arg){
   char msg[BUFF_SIZE];
         char incoming[BUFF_SIZE];
   size_t msgsize;



   strcpy(msg, "state:?");
   sendto(fd, msg, strlen(msg), 0, address->ai_addr, address->ai_addrlen);

   msgsize = recvfrom(fd, incoming, BUFF_SIZE, 0, NULL, 0);
         incoming[msgsize] = '\0';

   char *s;
         s = strstr(incoming, "state:=");

   if(s != NULL){
         strcpy(state, incoming);
   }

   return 0;
}
//gets the terrain information beneath the lander and stores it in the variable
void *getTerrain(void *arg){
   char msg[BUFF_SIZE];
         char incoming[BUFF_SIZE];
   size_t msgsize;

   strcpy(msg, "terrain:?");
   sendto(fd, msg, strlen(msg), 0, address->ai_addr, address->ai_addrlen);

   msgsize = recvfrom(fd, incoming, BUFF_SIZE, 0, NULL, 0);
         incoming[msgsize] = '\0';

   char *s;
         s = strstr(incoming, "terrain:=");

   if(s != NULL){
         strcpy(terrain, incoming);
   }
   return 0;
}
//gets the condition and stores it in the variable
void *getCondition(void *arg){
```

```
    char msg[BUFF_SIZE];
            char incoming[BUFF_SIZE];
    size_t msgsize;

    strcpy(msg, "condition:?");
    sendto(fd, msg, strlen(msg), 0, address->ai_addr, address->ai_addrlen);

    msgsize = recvfrom(fd, incoming, BUFF_SIZE, 0, NULL, 0);
            incoming[msgsize] = '\0';

    char *s;
            s = strstr(incoming, "condition:=");

    if(s != NULL){
            strcpy(condition, incoming);
    }
    return 0;

}
//initiallises the data log thread
void logCommand(char msg[BUFF_SIZE])
{
  sem_wait(&logSemaphore);
      rc3 = pthread_create( &logThread, NULL, logData, msg);
            assert(rc3 == 0);
             pthread_join(logThread, NULL);
  sem_post(&logSemaphore);
}

// sends a control  command to the server
void *sendCommand(void *arg)
{
    char *msg;
    msg = (char*) arg;
            char incoming[BUFF_SIZE];
    size_t msgsize;

    sendto(fd, msg, strlen(msg), 0, address->ai_addr, address->ai_addrlen);
    msgsize = recvfrom(fd, incoming, BUFF_SIZE, 0, NULL, 0);
            incoming[msgsize] = '\0';
    return 0;
}


void finished(int sig)
{
            exit(0);
}

//static int fd;
void cleanup(void)
{
            close(fd);
```

```
}
```