Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра ПМиК

Лабораторная работа №3

«Применение потоковых шифров HC-128, Rabbit, Salsa20 и Sosemanuk»

Выполнил:
студент гр. МГ-211 _____ / Бурдуковский И.А./
подпись

Проверил:
Профессор
кафедры ПМиК _____ / Фионов А.Н./

Новосибирск

2023 г.

ОГЛАВЛЕНИЕ

Задание

1. Для всех четырёх шифров (HC-128, Rabbit, Salsa20 и Sosemanuk) сравнить длительность функций инициализации, и время генерации ключевого потока.
2. Выбрать один понравившийся потоковый шифр и реализовать на его основе приложение для передачи файла в зашифрованном виде по сети. Сравнить с соответствующей реализацией на основе блочного шифра.

Выполнение

1. Реализованы необходимые алгоритмы. Для сравнения было решено взять результаты за 256 повторений. Результирующее время в тиках. Рассчитывается как среднее арифметическое.

   Результаты:

```
hc-128
Initialization Avg: 94895 ticks.
Generation key Avg: 1455 ticks.
```

```
Rabbit
Initialization Avg: 3450 ticks.
Generation key Avg: 1708 ticks.
```

```
Salsa-20
Initialization Avg: 128 ticks.
Generation key Avg: 4709 ticks.
```

```
Sosemanuk
Initialization Avg: 6690 ticks.
Generation key Avg: 2089 ticks.
```

   Подводя итоги, можно сказать, что по результатам в инициализации лидирует – Salsa20 с заметным отрывом. По времени кодирования – HC-128, но Rabbit очень близок к результатам HC-128.

2. Для преобразования в клиент-серверное приложение был взят алгоритм HC-128.
   Структуру сообщения которая будет включать шифротекст и вектор инициализации:

   Сервер отдает и принимает пакеты для каждого клиента в потоке. Клиент устанавливает вектор инициализации и шифрует сообщение которое потом кладется в поле message. Другой клиент получает пакет и также устанавливает вектор инициализации и декодирует сообщение.

Листинг

```
struct myMessageStruct
{
 char username[32];
 u8 iv[16];
 u8 message[64];
 char pad_0[144];
};

std::string GetLastErrorAsString()
{
 //Get the error message, if any.
 DWORD errorMessageID = ::GetLastError();
 if (errorMessageID == 0)
       return std::string(); //No error message has been recorded

 LPSTR messageBuffer = nullptr;
 size_t size = FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
       NULL, errorMessageID, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
(LPSTR)&messageBuffer, 0, NULL);

 std::string message(messageBuffer, size);

 //Free the buffer.
 LocalFree(messageBuffer);

 return message;
}

std::string WSAGetLastErrorAsString()
{
 //Get the error message, if any.
 DWORD errorMessageID = ::WSAGetLastError();
 if (errorMessageID == 0)
       return std::string(); //No error message has been recorded

 LPSTR messageBuffer = nullptr;
 size_t size = FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
       NULL, errorMessageID, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
(LPSTR)&messageBuffer, 0, NULL);

 std::string message(messageBuffer, size);

 //Free the buffer.
 LocalFree(messageBuffer);

 return message;
}

void ClearScreen()
{
 HANDLE                     hStdOut;
 CONSOLE_SCREEN_BUFFER_INFO csbi;
 DWORD                      count;
 DWORD                      cellCount;
 COORD                      homeCoords = { 0, 0 };

 hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
 if (hStdOut == INVALID_HANDLE_VALUE) return;

 /* Get the number of cells in the current buffer */
 if (!GetConsoleScreenBufferInfo(hStdOut, &csbi)) return;
 cellCount = csbi.dwSize.X * csbi.dwSize.Y;
```

```
 /* Fill the entire buffer with spaces */
 if (!FillConsoleOutputCharacter(
       hStdOut,
       (TCHAR)' ',
       cellCount,
       homeCoords,
       &count
 )) return;

 /* Fill the entire buffer with the current colors and attributes */
 if (!FillConsoleOutputAttribute(
       hStdOut,
       csbi.wAttributes,
       cellCount,
       homeCoords,
       &count
 )) return;

 /* Move the cursor home */
 SetConsoleCursorPosition(hStdOut, homeCoords);
}



int main(int argc, char** argv)
{
 WSADATA wsaData;
 SOCKET ConnectSocket = INVALID_SOCKET;
 struct addrinfo* result = nullptr,
       * ptr = nullptr,
       hints{};
 char receive_buffer[_chatPacketSize];

 // Initialize Winsock
 int i_result = WSAStartup(MAKEWORD(2, 2), &wsaData);
 if (i_result != 0) {
       printf("WSAStartup failed with error: %d\n", i_result);
       return 1;
 }

 ZeroMemory(&hints, sizeof(hints));
 hints.ai_family = AF_UNSPEC;
 hints.ai_socktype = SOCK_STREAM;
 hints.ai_protocol = IPPROTO_TCP;

 // Resolve the server address and port
 i_result = getaddrinfo("localhost", _chatPort, &hints, &result);
 if (i_result != 0) {
       printf("getaddrinfo failed with error: %d\n", i_result);
       WSACleanup();
       return 1;
 }

 // Attempt to connect to an address until one succeeds
 for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

       // Create a SOCKET for connecting to server
       ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
             ptr->ai_protocol);
       if (ConnectSocket == INVALID_SOCKET) {
             printf("socket failed with error: %ld\n", WSAGetLastError());
             WSACleanup();
             return 1;
       }

       // Connect to server.
```

```
	i_result = connect(ConnectSocket, ptr->ai_addr, (int)ptr-
>ai_addrlen);
	if (i_result == SOCKET_ERROR) {
		closesocket(ConnectSocket);
		ConnectSocket = INVALID_SOCKET;
		continue;
	}
	break;
 }

 freeaddrinfo(result);

 if (ConnectSocket == INVALID_SOCKET) {
	printf("Unable to connect to server!\n");
	WSACleanup();
	return 1;
 }

 std::string s_name;
 std::cout << "Please enter your name: ";
 std::getline(std::cin, s_name);

 ECRYPT_init();
 ECRYPT_keysetup(&ctx, &(key[0]), 128, 128);

 while (true)
 {
	// Receive chat data from server
	i_result = recv(ConnectSocket, receive_buffer, _chatPacketSize, 0);
	if (i_result > 0)
	{
		ClearScreen();
		const auto chat_history =
reinterpret_cast<myMessageStruct*>(receive_buffer);
		for (int i = _chatHistoryLength - 1; i > -1; i--)
		{
			ECRYPT_ivsetup(&ctx, &(chat_history[i].iv[0]));

			u8 data_out[64];

			ECRYPT_encrypt_bytes(&ctx, &(chat_history[i].message[0]),
&data_out[0], 64);

			std::cout << std::setw(sizeof(chat_history[i].username))
<< chat_history[i].username << " | " <<
				(char*)data_out << std::endl;
		}
	}
	else if (i_result == 0)
	{
		std::cout << "[i] Connection closed!" << std::endl;
	}
	else
	{
		std::cout << "[!] (Receive) failed with error: " <<
WSAGetLastErrorAsString() << std::endl;
		closesocket(ConnectSocket);
		WSACleanup();
		return 1;
	}

	std::string s_message;
	std::cout << "> ";
	std::getline(std::cin, s_message);

	if (s_message == "!quit")
	{
```

```
			std::cout << "[i] Closing connection!" << std::endl;
			break;
		}

		myMessageStruct packet{};
		s_name.resize(sizeof(packet.username), '\0');
		packet.username[0] = 0;
		s_name.copy(packet.username, s_name.size());
		packet.username[sizeof(packet.username) - 1] = '\0';

		memcpy(packet.iv, iv, 16);

		u8 data_in_m[64];
		data_in_m[0] = 0;
		s_message.resize(sizeof(packet.message), '\0');
		memcpy(data_in_m, s_message.c_str(), s_message.size());
		ECRYPT_ivsetup(&ctx, &(iv[0]));
		ECRYPT_encrypt_bytes(&ctx, &data_in_m[0], &packet.message[0], 64);

		/*s_message.resize(sizeof(packet.message), '\0');
		packet.message[0] = 0;
		s_message.copy(packet.message, s_message.size());
		packet.message[sizeof(packet.message) - 1] = '\0';*/

		// Send message to the server
		i_result = send(ConnectSocket, reinterpret_cast<char*>(&packet),
_chatMessageSize, 0);
		if (i_result == SOCKET_ERROR) {
			std::cout << "[!] (Send) failed with error: " <<
WSAGetLastErrorAsString() << std::endl;
			closesocket(ConnectSocket);
			WSACleanup();
			return 1;
		}
	}

	// shutdown the connection since no more data will be sent
	i_result = shutdown(ConnectSocket, SD_SEND);
	if (i_result == SOCKET_ERROR) {
		std::cout << "[!] (Shutdown) failed with error: " <<
WSAGetLastErrorAsString() << std::endl;
		closesocket(ConnectSocket);
		WSACleanup();
		return 1;
	}

	closesocket(ConnectSocket);
	WSACleanup();

	return 0;
}
```