Timothy Ferrell

Leo Linsky

Yao Xiao

**Lab 12: Team GotYourNodes**

**Key Generation**

The first step in any RSA key exchange is generating the RSA keys themselves. Although we could have generated the numbers ourselves, there is no need to. There are existing, open-source implementations of RSA key generation. In order to generate these keys, we utilized the linux ssl library. This has built in functions to generate keys of any length. We then converted these keys into 64-bit words in a custom struct, keyPair, so that we could easily send the netFPGA our public key, as well as the modulus we are using. Full source code will be included in the appendix of this report, but we will include some code snippets here to help understanding. Following is our function to generate RSA key pairs:



We then send our public key to the netFPGA, which processes and sends back an encrypted public key according to our key exchange protocol. This protocol is detailed in the next section.

**Key Exchange**

The key exchange suggested in the lab assignment is not a great idea. The double RSA key exchange suggested does not add any extra security, but only adds complexity. Our key exchange protocol will be a more classic RSA key exchange:

1. The node generates a network packet that contains its own public RSA key.
2. The node sends the packet to the NetFPGA port
3. When the NetFPGA receives the packet, its network processor extracts the public RSA key of the node and stores it in its memory (memory address like 250 or something outside of packet space)
4. The network processor uses the public RSA key (which it just received) of the node to encrypt a random symmetric key it generates. This can be any 64 bit random string. We can simply hardcode this in the memory also, outside of the packet space)
5. The network processor modifies the received packet to send a response packet containing the encrypted symmetric key to the sender.
6. The node receives the packet then decrypts the symmetric key using its own private key.
7. Now all future packets through the port are encrypted with the shared symmetric key.

This will provide a high level of security for our packets in the future. We will use this shared symmetric key to implement encrypted communications in Lab 13.

**Protocol and Implementation**

<Yao insert short description and schematics of hardware changes to support RSA encryption here>

As described earlier, the node in this protocol can be thought of as the Key Exchange Requester. It sends a Key Exchange Request Packet to the NetFPGA. This packet contains both the public key and the modulus we have generated. The format of a Key Exchange Request packet as seen in the NetFPGA data memory can be found in Appendix A. The NetFPGA in this protocol can be thought of as the Key Exchange Responder. It receives a Key Exchange Request packet, extracts the public key and modulus, and uses them to encrypt a random, 64 bit symmetric key. It then encapsulates this encrypted symmetric key in a Key Exchange Response packet, and sends this back to the requester. The format of a Key Exchange Response packet as seen in the NetFPGA data memory can be found in Appendix B. We generate the symmetric key with the following C code:

Then we pass the symmetric key to the netFPGA using a software register.

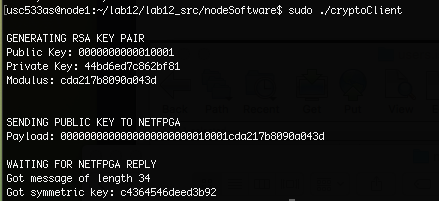
**Results:**

Following are some pictures of a key exchange, from the point of view of the end node. This shows successful implementation of our key exchange.

Here is a screenshot of the symmetric key being generated:



And then a screenshot of the successful key exchange:



As you can see, we have successfully exchanged a secret symmetric key.

**TASKS LEFT TO DO:**

ON NODE:

* Receive and decrypt symmetric key: MOSTLY DONE - decryption implementation not done, but we should wait until everything else is done. probably can just use the ME and MM functions included in crypto.h

ON NETFPGA

* Write assembly code (EXTEND ipProtoFilter) to then modify (or generate) a packet back that now contains the encrypted symmetric key. We will probably need to modify module header, as well as IP and possibly ethernet header. See below for packet format in NetFPGA. we could just hardcode these header into data memory and overwrite to make it easy.
* Implement hardware/software to encrypt using RSA key

**Appendix A: Key Exchange Request protocol message format in NetFPGA memory**

|  |  |
| --- | --- |
| **DMEM ADDRESS** | **CONTENTS OF WORD IN MEMORY** |
| **0** | MODULE HEADER - see https://github.com/NetFPGA/netfpga/wiki/ReferenceRouterWalkthrough |
| **1** | ETHERNET HEADER bytes 0-7 |
| **2** | ETHERNET HEADER bytes 8-13 | IP HEADER bytes 0-1 |
| **3** | IP HEADER bytes 2-9 |
| **4** | IP HEADER bytes 10-17 |
| **5** | IP HEADER bytes 18-19 | 6 BYTES PADDING ADDED BY US |
| **6** | GENERATED PUBLIC KEY |
| **7** | GENERATED MODULUS |

**Appendix B: Key Exchange Reply protocol message format in NetFPGA memory**

|  |  |
| --- | --- |
| **DMEM ADDRESS** | **CONTENTS OF WORD IN MEMORY** |
| **0** | MODULE HEADER - see https://github.com/NetFPGA/netfpga/wiki/ReferenceRouterWalkthrough |
| **1** | ETHERNET HEADER bytes 0-7 |
| **2** | ETHERNET HEADER bytes 8-13 | IP HEADER bytes 0-1 |
| **3** | IP HEADER bytes 2-9 |
| **4** | IP HEADER bytes 10-17 |
| **5** | IP HEADER bytes 18-19 | 6 BYTES PADDING ADDED BY US |
| **6** | ENCRYPTED SYMMETRIC KEY |

**Appendix C. Node Software Source Code**

cryptoClient.c: Sends Key Exchange request and waits for reply

#include "crypto.h"

#define PAYLOAD\_LENGTH 16

#define KEY\_LENGTH 8

#define MODULO\_LENGTH 8

#define PADDING\_LENGTH 6

#define HEADER\_SIZE 20

int main(){

int sd;

struct sockaddr\_in dest;

dest.sin\_family = AF\_INET;

unsigned char payload[(PAYLOAD\_LENGTH+PADDING\_LENGTH)];

bzero(payload,(PAYLOAD\_LENGTH+PADDING\_LENGTH));

unsigned char encryptedSymKey[PAYLOAD\_LENGTH];

unsigned char recvbuf[1024];

int got\_reply = 0;

int recv\_size = 0;

socklen\_t sockaddr\_size = sizeof(dest);

//generate RSA keys

printf("\nGENERATING RSA KEY PAIR\n");

keyPair64 keypair = generateRSAKeys(64);

//Print both the RSA struct and our custom struct for debugging

print\_keyPair64(keypair);

//Set up socket and sent packet

memcpy(payload+PADDING\_LENGTH,keypair.publicKey,KEY\_LENGTH);

memcpy(payload+PADDING\_LENGTH+KEY\_LENGTH,keypair.modulus,MODULO\_LENGTH);

if(inet\_pton(AF\_INET,"10.1.1.2",&(dest.sin\_addr)) != 1){

printf("Address not found...\n");

return(1);

}

if((sd = socket(AF\_INET,SOCK\_RAW,253)) <0){

printf("Couldn't create socket..\n");

}

printf("\nSENDING PUBLIC KEY TO NETFPGA\n");

printf("Payload: ");

print\_char2hex(payload,PAYLOAD\_LENGTH+PADDING\_LENGTH);

if(sendto(sd,payload,(PAYLOAD\_LENGTH + PADDING\_LENGTH),0,(struct sockaddr\*) &dest,sizeof(struct sockaddr))<0) {

printf("Send failed\n");

return(1);

}

//Now wait for reply

printf("\nWAITING FOR NETFPGA REPLY\n");

while(!got\_reply){

recv\_size = recvfrom(sd,recvbuf,1024,0,(struct sockaddr\*)&dest,&sockaddr\_size);

printf("Got message of length %d\n",recv\_size);

//for now this will be only message we possibly get on proto 253

got\_reply = 1;

}

//print encrypted symmetric key

printf("Got encrypted symmetric key: ");

print\_char2hex(recvbuf+HEADER\_SIZE+PADDING\_LENGTH,KEY\_LENGTH);

//print decrypted symmetric key

//Free RSA key

return(0);

}

crypto.h: Contains crypto-related functions

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* This file contains functions and

\* definitions for our custom crypto

\* library

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <openssl/rsa.h>

#include <openssl/pem.h>

#include <openssl/bn.h>

#include <strings.h>

#include <string.h>

#include <openssl/rand.h>

#include <stdio.h>

typedef struct keyPair64 {

unsigned char publicKey[8];

unsigned char privateKey[8];

unsigned char modulus[8];

} keyPair64;

void print\_char2hex(unsigned char\* array,int length){

int i;

for(i=0;i<length;i++){

printf("%02x",array[i]);

}

printf("\n");

}

void print\_keyPair64(keyPair64 keyPair){

printf("Public Key: ");

print\_char2hex(keyPair.publicKey,sizeof(keyPair.publicKey));

printf("Private Key: ");

print\_char2hex(keyPair.privateKey,sizeof(keyPair.privateKey));

printf("Modulus: ");

print\_char2hex(keyPair.modulus,sizeof(keyPair.privateKey));

printf("\n");

}

/\* Function to generate a 64 bit RSA public/privatekey pair \*/

/\* May need to add writing to file to save RSA key if custom implementation does not work\*/

keyPair64 generateRSAKeys(int size){

RSA\* rsa = NULL;

BIGNUM\* bn = NULL;

keyPair64 rsaKeyPair;

//Zero out bytes

bzero(rsaKeyPair.publicKey,sizeof(rsaKeyPair.publicKey));

bzero(rsaKeyPair.privateKey,sizeof(rsaKeyPair.privateKey));

bzero(rsaKeyPair.modulus,sizeof(rsaKeyPair.modulus));

unsigned long e=RSA\_F4;

bn=BN\_new();

int success = BN\_set\_word(bn,e);

if(success!=1){

printf("Issue with BN\_set\_word\n");

}

rsa=RSA\_new();

success = RSA\_generate\_key\_ex(rsa,size,bn,NULL);

if(success!=1){

printf("Issue with RSA generation\n");

}

//Store the key in our custom struct to simplify for netfpga

int offset=(sizeof(rsaKeyPair.publicKey)-BN\_num\_bytes(rsa->e));

BN\_bn2bin(rsa->e,&rsaKeyPair.publicKey[offset]);

offset=(sizeof(rsaKeyPair.privateKey)-BN\_num\_bytes(rsa->d));

BN\_bn2bin(rsa->d,&rsaKeyPair.privateKey[offset]);

offset=(sizeof(rsaKeyPair.modulus)-BN\_num\_bytes(rsa->n));

BN\_bn2bin(rsa->n,&rsaKeyPair.modulus[offset]);

BN\_free(bn);

RSA\_free(rsa);

return rsaKeyPair;

}

// Montgomery Multiplication

int MM(int x, int y, int m) {

int tmp = 0;

int i,j;

for(i = 1, j = 0; j < 64; i = i<<1, j++) {

int ui = ((tmp&1) + ((x&i) >> j) \* (y&1)) % 2;

tmp = (tmp + ((x&i)>>j)\*y + ui\*m) >> 1;

}

if (tmp >= m) {

tmp = tmp - m;

}

return tmp;

}

// Montgomery Exponentiation

// result = x^e mod m

// we should change length(m) to a known value once we know m;

int ME(int x, int e, int m) {

int xx=MM(x, (2^(2\*length(m))%m), m);

int a = (2^length(m))%m;

int i;

for(i = 63; i >= 0; i--) {

int j = 1 << i;

a = MM(a, a, m);

if ((e&j) >> i) a = MM(a, xx, m);

}

return MM(1, a, m);

}\*/

**Appendix D. NetFPGA Assembly Code**