NetBurner

Networking in 1 Day!

NetBurner Security Libraries

TABLE OF CONTENTS

Revision History

Date	Revision	Description
6/2/2010	1.00	Initial release. AES and SSL sections moved from NetBurner Runtime Library and are unchanged. Initial release of SSH section.
7/21/2010	1.01	Added more detailed descriptions in SSH section.

1. Introduction

This document is a reference manual for the NetBurner software libraries, and is intended to be used in conjunction with the NetBurner Network Programming Guide for network platforms, or the Mod5213 Programmers Guide for non-network platforms. These guides provide background, details and examples on how the functions in this document can be used in practice. All NetBurner documents are located in the documents directory created during installation. The default location is c:\nburn\docs.

This reference guide contains all the API function calls, some of which may not apply to your specific hardware platform. For example, the CAN functions are available on the Mod5213, but TCP/IP functions are not supported because it does not have a network interface. This reference manual also contains optional software APIs, such as SSL, that are purchased separately and are not part of the standard NetBurner development kit.

Hardware-specific software functions and information are provided in the c:\nburn\docs<platform> directory, where <platform> is the hardware platform you are using, such as a Mod5282. The platform documents contain schematics, memory maps, and any software features that are specific to the hardware platform you are using.

The software included in your NetBurner Development Kit is licensed to run only on processor hardware manufactured by NetBurner, such as the modules and serial to Ethernet devices. If your application involves manufacturing your own processor based hardware (ie you are not going to purchase NetBurner modules for production), please contact NetBurner Sales for details on a Royalty-Free Software License.

Additional Documentation

All NetBurner License Documentation is located by default in your C:\Nburn\docs directory.

- NBEclipse Getting Started Guide
- NetBurner Runtime Libraries
- NetBurner Network Programming Guide
- Mod5213 Programming Guide
- NetBurner PC Tools Guide
- Freescale microprocessor manuals
- Embedded Flash File System (EFFS) Programmers Guide
- Embedded Flash File System (EFFS) Reference Manual
- Platform Documents – The hardware specific documents for your device (eg Mod5282)

2. NetBurner License Information

The software included in your NetBurner Development Kit is licensed to run on hardware manufactured by NetBurner. if you wish to design your own processor board please contact NetBurner Sales.

All embedded software and source code provided in this Network Development Kit is subject to one of four possible licenses: the NetBurner Tools License (most restrictive), the NetBurner Embedded Software License, the GNU Public License and the Newlib License (least restrictive). The GNU development executables provided in the C:\Nburn\GCC-M68k directory branch are subject to the GNU Public License (GPL).

The Runtime Libraries and include files provided in the C:\Nburn\GCC-M68k directory branch are subject to the Newlib License.

The Compcode application provided in the C:\Nburn\pctools\compcode directory is subject to the GNU public license (GPL).

All other programs are subject to the NetBurner Tools License provided below.

All other provided Source Code and Libraries are subject to the NetBurner Embedded Software License provided below.

2.1. The NetBurner Tools Software License

Copyright 1998 - 2010 NetBurner, Inc., All Rights Reserved.

Permission is hereby granted to purchasers of the NetBurner Network Development Kit to use these programs on one computer, and only to support the development of embedded applications that will run on NetBurner provided hardware.

No other rights to use this program or its derivatives, in part or in whole, are granted. It may be possible to license this or other NetBurner software for use on non NetBurner hardware. NetBurner makes no representation or warranties with respect to the performance of this computer program, and specifically disclaims any responsibility for any damages, special or consequential, connected with the use of this program.

2.2. The NetBurner Embedded Software License

Copyright 1998 - 2010 NetBurner, Inc., All Rights Reserved.

Permission is hereby granted to purchasers of NetBurner hardware to use or modify this computer program for any use as long as the resultant program is only executed on NetBurner provided hardware. No other rights to use this program or its derivatives, in part or in whole, are granted. It may be possible to license this or other NetBurner software for use on non NetBurner hardware.

NetBurner makes no representation or warranties with respect to the performance of this computer program, and specifically disclaims any responsibility for any damages, special or consequential, connected with the use of this program.

2.3. Life Support Disclaimer

NetBurner’s products both hardware and software (including tools) are not authorized for use as critical components in life support devices or systems, without the express written approval of NetBurner, Inc. prior to use. As used herein:

Life support devices or systems are devices or systems that (a) are intended for surgical implant into the body or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. If you have any questions/concerns, please contact our Sales Department for more information.

2.4. Anti-Piracy Policy

NetBurner, Inc. vigorously protects its copyrights, trademarks, patents and other intellectual property rights.

In the United States and many other countries, copyright law provides for severe civil and criminal penalties for the unauthorized reproduction or distribution of copyrighted material. Copyrighted material includes, but is not limited to computer programs and accompanying sounds, images and text.

Under U.S. law, infringement may result in civil damages of up to $150,000, and/or criminal penalties of up to five years imprisonment, and/or a $250,000 fine. In addition, NetBurner, Inc. may seek to recover its attorneys' fees.

3. AES Library (Advanced Encryption Standard)

3.1. Introduction

AES is a “block cipher” that has been adopted as an encryption standard, and is one of the most popular algorithms used in symmetric key cryptography today. It was invented by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and is also referred to as “Rijndael”. AES is the successor to the Data Encryption Standard (DES).

A “block cipher” is a “symmetric key cipher” that operates on a fixed length group of bits. A typical block size is 64 or 128 bits. Multiple blocks are used to process data larger than 128 bits, and padding is used to process data less than 128 bits.

3.2. Function Call Summary

Create encryption/decryption keys:

```
void aes_set_key( aes_context *ctx, unsigned char *key, int keysize )
```

Encrypt a block of data:

```
void aes_encrypt( aes_context *ctx, unsigned char input[16], unsigned char output[16] )
```

Decrypt a block of data:

```
void aes_decrypt( aes_context *ctx, unsigned char input[16], unsigned char output[16] )
```

Encrypt a block of data using Cipher Block Chaining (CBC)

```
void aes_cbc_encrypt( aes_context *ctx, unsigned char iv[16], unsigned char *input,
                      unsigned char *output, int len )
```

Decrypt a block of data using Cipher Block Chaining (CBC)

```
void aes_cbc_decrypt( aes_context *ctx, unsigned char iv[16], unsigned char *input,
                      unsigned char *output, int len )
```

3.3. Examples

See the AES example located in c:\nburn\examples.

3.4. aes_set_key

Header File:

#include <aes.h>

Synopsis:

void aes_set_key(aes_context *ctx, unsigned char *key, int keysize);

Description:

Create a set of encrypt/decrypt keys

Parameters:

*ctx	Pointer to a structure that will contain the encryption and decryption keys
*key	Pointer to the secret key
keysize	Size of key, must be 128, 256 or 512 bits

3.5. aes_encrypt

Header File:

```
#include <aes.h>
```

Synopsis:

```
void aes_encrypt( aes_context *ctx,
                  unsigned char input[16],
                  unsigned char output[16] )
```

Description:

Encrypt a 16 byte block of data.

Parameters:

*ctx	Pointer to a structure containing the encryption and decryption keys
input[16]	Array of 16 bytes of data to be encrypted
output[16]	Array of 16 bytes to contain the encrypted data

3.6. aes_decrypt

Header File:

```
#include <aes.h>
```

Synopsis:

```
void aes_decrypt( aes_context *ctx,
                unsigned char input[16],
                unsigned char output[16] );
```

Description:

Decrypt a 16 byte block of data.

Parameters:

*ctx	Pointer to a structure containing the encryption and decryption keys
input[16]	Array of 16 bytes of data to be decrypted
output[16]	Array of 16 bytes to contain the decrypted data

3.7. aes_cbc_encrypt

Header File:

```
#include <aes.h>
```

Synopsis:

```
void aes_cbc_encrypt( aes_context *ctx,
                      unsigned char iv[16],
                      unsigned char *input,
                      unsigned char *output,
                      int len );
```

Description:

Encrypt a 16 byte block of data using Cipher Block Chaining (CBC)

Parameters:

*ctx	Pointer to a structure containing the encryption and decryption keys
iv[16]	Initialization vector (modified after use)
input[16]	Pointer to buffer holding the data to encrypt
output[16]	Pointer to buffer to hold the encrypted data
len	Length of data to be encrypted

3.8. aes_cbc_decrypt

Header File:

```
#include <aes.h>
```

Synopsis:

```
void aes_cbc_decrypt( aes_context *ctx,
                      unsigned char iv[16],
                      unsigned char *input,
                      unsigned char *output,
                      int len );
```

Description:

Encrypt a 16 byte block of data using Cipher Block Chaining (CBC)

Parameters:

*ctx	Pointer to a structure containing the encryption and decryption keys
iv[16]	Initialization vector (modified after use)
input[16]	Pointer to buffer holding the encrypted data
output[16]	Pointer to buffer to hold the unencrypted data
len	Length of data to be decrypted

4. SSL Library

4.1. Introduction

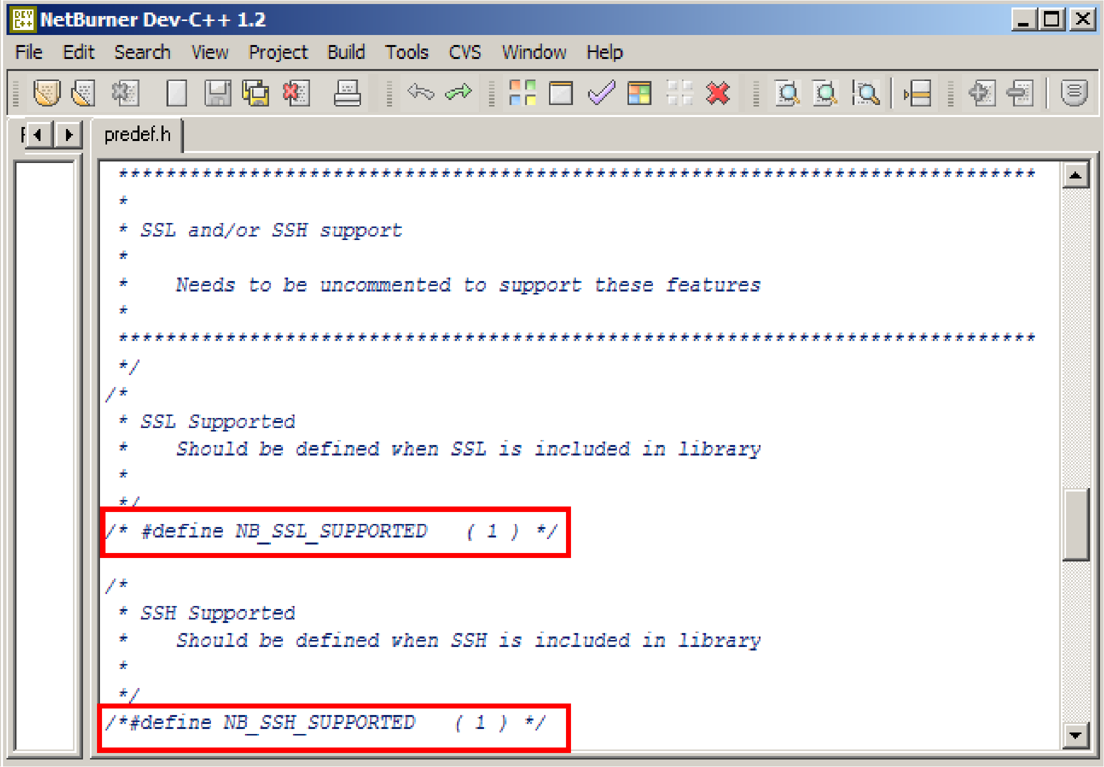
The NetBurner SSL package is sold as a licensed option only, and is not part of the standard development kit package. Please contact our **Sales** Department to purchase the SSL package.

Implementing SSL in an embedded system will require some knowledge of SSL certificates. Please read the following four SSL sections:

- SSL overview
- Creating SSL server certificates
- Diagram: Creating a code module - SSL Server Key and Certificates
- Creating the list of acceptable client certificates

Before you can use the SSL accept function, you will need to add a Server certificate to your project. Before you can use the SSL connect function, you will need to add a list of Client certificates to your project.

Important: Before you compile any programs, open predef.h (located in C:\Nburn\include) with any text editor, and uncomment “#define NB_SSL_SUPPORTED and “#define NB_SSH_SUPPORTED” to get your applications to compile when using the NetBurner SSL Module. After editing predef.h (i.e. uncomment the mentioned lines), you must execute the make clean command (at the command line) in your C:\Nburn\system directory.



```
 *****************************************************************************
 *
 * SSL and/or SSH support
 *
 *    Needs to be uncommented to support these features
 *
 *****************************************************************************
 */
/*
 * SSL Supported
 *    Should be defined when SSL is included in library
 *
 */
/* #define NB_SSL_SUPPORTED   ( 1 ) */

/*
 * SSH Supported
 *    Should be defined when SSH is included in library
 *
 */
/*#define NB_SSH_SUPPORTED   ( 1 ) */
```

Warning: If you do not edit predef.h, your applications will not compile (GATHER_RANDOM error)

Header File

```
   #include <ssl.h>        // Found in C:\Nburn\include\crypto
```

SSL Server Functions

- `StartHTTPs` --- Starts the secure Web Server
- `SSL_accept` --- SSL mirror of the TCP accept function

File Descriptor (fd) Information Functions

- `IsSSLfd` --- Is the file descriptor an SSL file descriptor or some other kind
- `SSL_GetSocketRemoteAddr` --- Returns the remote address of this connected socket
- `SSL_GetSocketRemotePort` --- Returns the remote port of this connected socket
- `SSL_GetSocketLocalAddr` --- Returns the local address of this connected socket
- `SSL_GetSocketLocalPort` --- Returns the local port of this connected socket

Socket Option Functions

- `SSL_setsockoption` --- Set the socket option
- `SSL_clrsockoption` --- Clear the socket option
- `SSL_getsockoption` --- Get the socket option

SSL Client Function

- `SSL_connect` --- SSL mirror of the TCP connect call

SSL Error Codes

In addition to the TCP error codes, SSL functions can return the following error codes:

```
#define SSL_ERROR_FAILED_NEGOTIATION         (-256)
#define SSL_ERROR_HASH_FAILED                (-257)
#define SSL_ERROR_CERTIFICATE_UNKNOWN        (-258)
#define SSL_ERROR_WRITE_FAIL                 (-259)
#define SSL_ERROR_CERTIFICATE_NAME_FAILED    (-260)
#define SSL_ERROR_CERTIFICATE_VERIFY_FAILED  (-261)
```

SSL Email Function

- SSL_SendMail --- Send an email message using SSL

SSL Email Error Reporting Variables

- NB_Mail_Error_Code --- Returns 0 or error code
- NB_Mail_Error_String[] --- Last error string reported by email functions, displayed on the debug serial port.
- Server_Mail_Log_String[] --- Last error string received from SMTP server

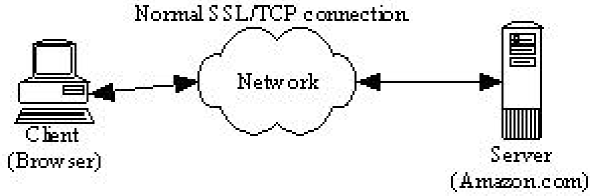
SSL Email Error Codes (located in c:\nburn\include\crypto\ssl_mailto.h)

```
#define STATUS_OK                      (0)
#define CONNECT_TO_SMTP_SERVER_FAILED  (-1)
#define INITIAL_SERVER_REPLY_FAILED    (-2)
#define HELO_SERVER_REPLY_FAILED       (-3)
#define MAIL_FROM_SERVER_REPLY_FAILED  (-4)
#define RCPT_TO_SERVER_REPLY_FAILED    (-5)
#define DATA_SERVER_REPLY_FAILED       (-6)
#define DATA_END_SERVER_REPLY_FAILED   (-7)
#define AUTH_LOGIN_SERVER_REPLY_FAILED (-8)
#define USER_ID_SERVER_REPLY_FAILED    (-9)
#define PASSWORD_SERVER_REPLY_FAILED   (-10)
#define CONNECT931_SMTP_SERVER_FAILED  (-11)
```

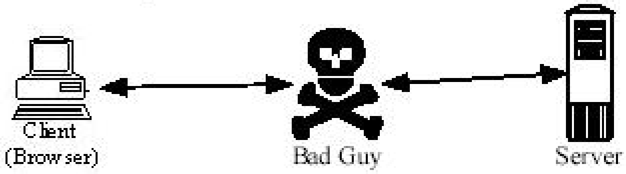
4.2. SSL Overview

The NetBurner SSL library makes SSL as easy as it can be, but SSL requires a system of trusted certificates. The NetBurner SSL package is sold as a licensed option only, and is not part of the standard development kit package. Please contact our Sales Department if you want to purchase the SSL package.

When you use SSL to connect to http://www.amazon.com (for example) with a normal web browser, you will not need to know anything about certificates. This is because Amazon purchased a certificate from Verisign and your browser vendor preinstalled Verisign, as an entity that can sign trusted certificates.



Above, is a picture of a perfectly normal TCP or SSL connection; the client (most often a browser) has connected through the network to a server. If we do not have any entities doing bad things on our network then there is no need for SSL. However, if the data we are sending is worth stealing, we might have a very different network picture (below).



If our connection is routed through a third party (a normal TCP connection), we have no guarantee that this third party is not a "bad guy" trying to steal or modify our data. The SSL protocol was designed to eliminate this man in the middle attack. SSL is designed not only to make sure that the data we send over the network is hidden from snooping eyes, but it is also designed to make sure we are connected to the proper server without any "bad guys" in the middle. This verification is done with Public Key (PK) Cryptography and a hierarchy of trust.

Why do we trust a doctor when we go to the emergency room? We trust the doctor we have never met because we trust the hospital to employ qualified doctors. The hospital vouches for his skills and we trust the hospital. His medical school also vouches for him by giving him a diploma with his name and the schools seal or signature. We trust the school, we trust the hospital, and thus we trust the doctor.

SSL works in a very similar way. When a client connects to the server the server sends the client a certificate. This certificate has three major elements:

A name (i.e. who is this server)
A public key (e.g. think of an open padlock)
A signature (by a trusted third party that vouches for the name and the public key)

A doctor's diploma is very similar; it also has three major elements: The doctor's name, the type of degree and the medical school (that vouches for the doctor). For example, Bob and George both graduate from Harvard Medical School. They both have Harvard diplomas. However, the diplomas are unique to each doctor. The diplomas are not interchangeable. Bob's diploma would be of no use to George and vice versa. This illustrates the first key point about SSL.

Key Point # 1: Each and every SSL server must have a unique certificate. Note: You cannot reuse an SSL server certificate. The certificates are distinguished by the "common name" or "CN" on the certificate.

If you went into a new doctor's office and saw a diploma from Harvard medical school, you would feel comfortable with the doctor's skills. You trust Harvard and Harvard will vouch for this doctor. If instead, the diploma were from the Medical School of Zaire, you would probably be more skeptical. We do not have the same inherent trust of this school as we did with Harvard.

In SSL as a client, we have to decide who we will trust to sign our certificates. This list of trusted certificate authorities must be explicitly configured into the client. When the web browser or OS was installed on your PC, it probably installed a list of trusted certificate authorities. With the NetBurner SSL library, we have to explicitly decide whom we are going to trust to sign server certificates. This leads to key point #2.

Key Point # 2: An SSL client must be pre-configured with a list of Certificate Authorities (CAs) that it will trust to sign server certificates. This list can be common across all the clients and does not have to be unique.

What do I need to do to make SSL work?

- You must create or choose a certificate authority. Note: If you create a certificate authority, you will also have to create a set of public/private keys for this authority.
- You must create public/private keys and a certificate for each SSL server and have the certificate signed by the certificate authority you have chosen.
- You must configure the clients with the list of certificate authorities it should trust.

How do I find or create a certificate authority?

Using the medical school example, you can go to Harvard pay lots of $$$ and get a diploma that is trusted by everyone. You can also choose to start your own medical school and issue diplomas. Almost everyone in the world would trust a Harvard diploma. Almost no one will trust a "Bob's Medical School" diploma, unless you spend the time convincing them that it is a quality medical school. In the end, you will likely only be able to convince your family, and then only for non-life threatening needs.

SSL certificates are a lot like medical schools; you can go and purchase server certificates. To see what a certificate looks like, open your web browser (e.g. Internet Explorer), and connect to https://www.NetBurner.com (notice the s on the end of https). On Internet Explorer’s menu, choose File then Properties. Now, click the Certificates button, and look at all the tabs shown in this section.

How do I know whom my browser trusts?

On your (Internet Explorer) browser's menu - choose Tools then Internet Options. Open the Content tab, click the Certificates button, and open the Trusted Root Certificate Authorities tab. Add Verisign or Thawte and every browser in the world will trust your certificate and your server.

If you want to save some money and create your own certificate authority then you can do so. However, none of the clients will accept your certificate until you convince them to add "Bob’s Certificate Authority" to their list of trusted certificate authorities. If, the users using the embedded SSL system you are deploying are all in one business entity, then it is relatively simple to add your own certificate authority to the list of trusted authorities. If you are responsible for both the client and server end of the connection, it is even easier; you can configure the clients to accept a single server authority - yours.

SSL is based on Public Key Cryptography (PK) and a little bit of background on PK is necessary in order to deploy a secure SSL solution. Public Key Cryptography is different from Symmetric Key Cryptography. In PK, the keys used for encryption are broken into two parts, much like a padlock (the public part) and a key (the private part). If you give someone an open padlock and a steel box, they can put things into the box, close the lid, and lock the lock. Unless they have the key to the lock, they cannot open the box. They can be confident that if they mail you the box, none of the mailmen along the way can look inside. Only the person who holds the (private) key to the padlock can open the box. For additional information on Public Key Cryptography, please read the Cryptography FAQ (http://isc.faqs.org/faqs/cryptography-faq)

When the SSL client connects to a server, the server sends back a certificate with a public key (open padlock). This certificate also includes the name of the server and a signature vouching for both the public key and the name. If any part of the certificate is changed, the signature will compute to be invalid.

So, if we have a "bad guy" in the middle, he can watch the padlock going from the server to the client. But, when the client puts his secret information into the box and locks it, the "bad guy" cannot see inside. He only knows that the client sent something in the box to the server. The secrets in the box are safe from the prying eyes of the "bad guy". This safety only exists if the server has done a good job of protecting the private key. If the "bad guy" sneaks into the server room, logs on the server console, and makes a copy of the private key, he can intercept all of the traffic. He can also change the content at will. This leads to key point #3.

Key Point # 3: When using Public Key Cryptography (as SSL does), the system is only as secure as the security of the private key. Since a server needs access to the private key to unlock the data from the client, the private key must exist on the server.

Key Point # 3 Corollary: If the private key exists on the server, then the system is only as secure as the physical security of the server. If the server is not physically secure, then someone (i.e. the “bad guy”) can attach an emulator or other hardware to the server and read out the private key.

For example, suppose the "bad guy" wants to intercept your credit card number when you send it to Amazon to order a book. We have already shown that he cannot read the data unless he has Amazon's private key. However, he has one other option - he can pretend to be Amazon and offer his own certificate to you, the client. If this certificate is properly signed by a Certificate Authority the client trusts, then client will accept the connection. If any "Certificate Authority" in the list of trusted authorities is compromised, then the system is insecure. If the "bad guy" has the ability to add a new "Certificate Authority" to the client, then he can completely compromise the system. This leads to key point #4.

Key Point # 4: If the ability to add a "Certificate Authority" to the client’s list of trusted authorities is not secure, then system is not secure.

Key Point # 4 Corollary: If the list of trusted "Certificate Authorities" exists on the client, then the system is only as secure as the physical security of the client. If the client is not physically secure, then someone (i.e. the “bad guy”) can attach an emulator or other hardware to the client and add a "trusted" authority.

These last two key points imply that it is not possible to build a system that is more secure than the physical security of the device being secured. Important: All the cryptography in the world will not help if someone can gain access to your computer and hide a bug inside the keyboard; or even easier, add or modify a system file to record your keystrokes and periodically send them over the internet to some nefarious foe. Note: If your data is valuable enough to be attractive to a skilled adversary, then you must learn to be truly paranoid.

Recommended Reading

For an excellent overview of computer security

- Secrets and Lies by Bruce Schneier (ISBN 0-471-25311-1)

For a detailed review of cryptography

- Applied Cryptography by Bruce Schneier (ISBN 0-471-11709-9)

For a detailed description of the SSL protocol

- SSL and TLS by Eric Rescorla (ISBN 0-201-61598-3)

For a reference on the math and methods in cryptography (this is a heavy duty book):

- Handbook of Applied Cryptography by Menezes, Oorschot and Vanstone (ISBN 0-8493-8523-7)

4.3. Creating a Code Module for SSL Server Certificates

Introduction

The NetBurner SSL library provides some open source tools for the generation and maintenance of SSL keys and certificates. These key and certificate management tools are based on the fine openssl package available from http://www.openssl.org. These tools are subject to the openssl License. The embedded SSL library code is derived from other sources and is subject to the standard NetBurner License (located by default in C:\Nburn\docs)

Setting up the Environment

Important: Before you can perform any of these steps, you must set up the environment.

- Make sure the **openssl.exe** provided with the NetBurner SSL Library is in your path (by default it installs in C:\Nburn\pcbin)
- The default configuration file is named openssl.cnf, and is located C:\Nburn\ssl\config. You may edit the default values for your business if you wish.
- Make sure that your system environment has the variable: OPENSSL_CONF= <your path >\openssl.cnf. For example, in a command prompt window you can type: "SET OPENSSL_CONF=C:\NBURN\SSL\CONFIG\OPENSSL.CNF if that is your path.

Creating a Certificate Authority (CA)

This step creates a CA you can use to sign SSL server certificates. Important: You should only have to do this step once. The key file created in this step should be protected as the security of all your SSL Certificates depends on it. You have two choices to protect this key file.

You can use a pass phrase to encode it. You can leave it unencoded, but would need to secure the computer it is stored on.

Note**:** If you want the key to not be protected by a pass phrase then leave the –des off the genrsa command.

- Open a command prompt/DOS window
- Navigate to the directory you want to house your CA files
- To make a CA Key file, execute the command (and press the Enter key when finished):

  ```
  openssl genrsa –out CA.key –des
  ```

- Create a CA Certificate, by executing the command (and press the Enter key when finished):

  ```
  openssl req -new -key CA.key -x509 -days 3650 -out CA.crt
  ```

You will be prompted to answer some identification questions. How you answer them is up to you, but when creating a NetBurner CA we answered these questions as follows:

Country Name: US
State or Province: California
Locality: San Diego
Organization Name: NetBurner
Organizational Unit Name: Certificate Authority
Common Name: NetBurner CA

If you are going to be accessing the embedded SSL device via a web browser you will need to add this Certificate to your web browser's list of trusted certificate authorities. To do this for Internet Explorer:

- Open up your Internet Explorer web browser
- From the Tools menu go into the Internet Options section
- Select the Content tab
- Press the Certificates button
- Select the Trusted Root Certification Authorities tab
- Press the Import button
- Select the CA.crt file to import. (Note: It will not show up via the browse button unless you change the file type combo box at the bottom of the window to look for X509 certificate files.)

Creating a Server Key

You will need to create a server key for each SSL Server you intend to deploy. If you are deploying many SSL servers, the bookkeeping associated with this will not be trivial.

- Open a command prompt/DOS window
- Navigate to the directory you want to house your device files
- To make a Device Key file, execute the command (and press the Enter key when finished):

```
openssl genrsa –out devicename.key
```

You will need to keep track of this key file while you make a server certificate, as the two have to be matched. If you are creating your own certificates, you can create a big batch file that does all of the steps in a single execution. See the batch file in Appendix I.

Creating a Server Certificate with your CA

You will need to create a server certificate for each SSL Server you intend to deploy. If you are deploying many SSL Servers, the bookkeeping associated with this will not be trivial. The common name you enter in this step must match the deployed DNS name or the IP Address of the Server it will be used on.

- Open a command prompt/DOS window
- Navigate to the directory that you want to house your device files
- To make a Device Certificate Request file, execute the command (and press the Enter key when finished):

  ```
  openssl req -new -key devicename.key -out devicename.csr
  ```

- To make a Device Certificate, execute the command (**all on one line**) and press the "Enter" key when finished:

  ```
  openssl x509 -req -days 365 -in devicename.csr –CA CA.crt
               -CAkey CA.key -CAcreateserial -out devicename.crt
  ```

You can combine the creation of server keys, certificates, and code by running the batch file shown in Appendix I (at the end of this section).

Converting a Certificate and Key to Code

This step takes both the private Server Key and the Server Certificate and converts them into a CPP source code module that can be linked into your application. This implies that you need to generate a different application image for each of your servers.

- Open a command prompt/DOS window
- Navigate to the directory that you want to house your device files
- To make a Device CPP file with the key in it, execute the command (and press the Enter key when finished):

  ```
  openssl rsa -in devicename.key -nburn –out devicename.cpp
  ```

- To add the Certificate to the CPP file, execute the command (and press the Enter key when finished):

  ```
  openssl x509 -nburn -in devicename.crt –append devicename.cpp
  ```

Adding the Module to your Code Set

Take the devicename.cpp file previously created and import it into your project directory. If you are using command line tools, copy it to your project directory and add it to your makefile.

Create a Server Certificate for External CA

If you are going to have your certificates signed by an external entity, they will need a Certificate Request file. Note**:** The common name you enter in this step must match the deployed DNS name or IP Address of the Server it will be used on.

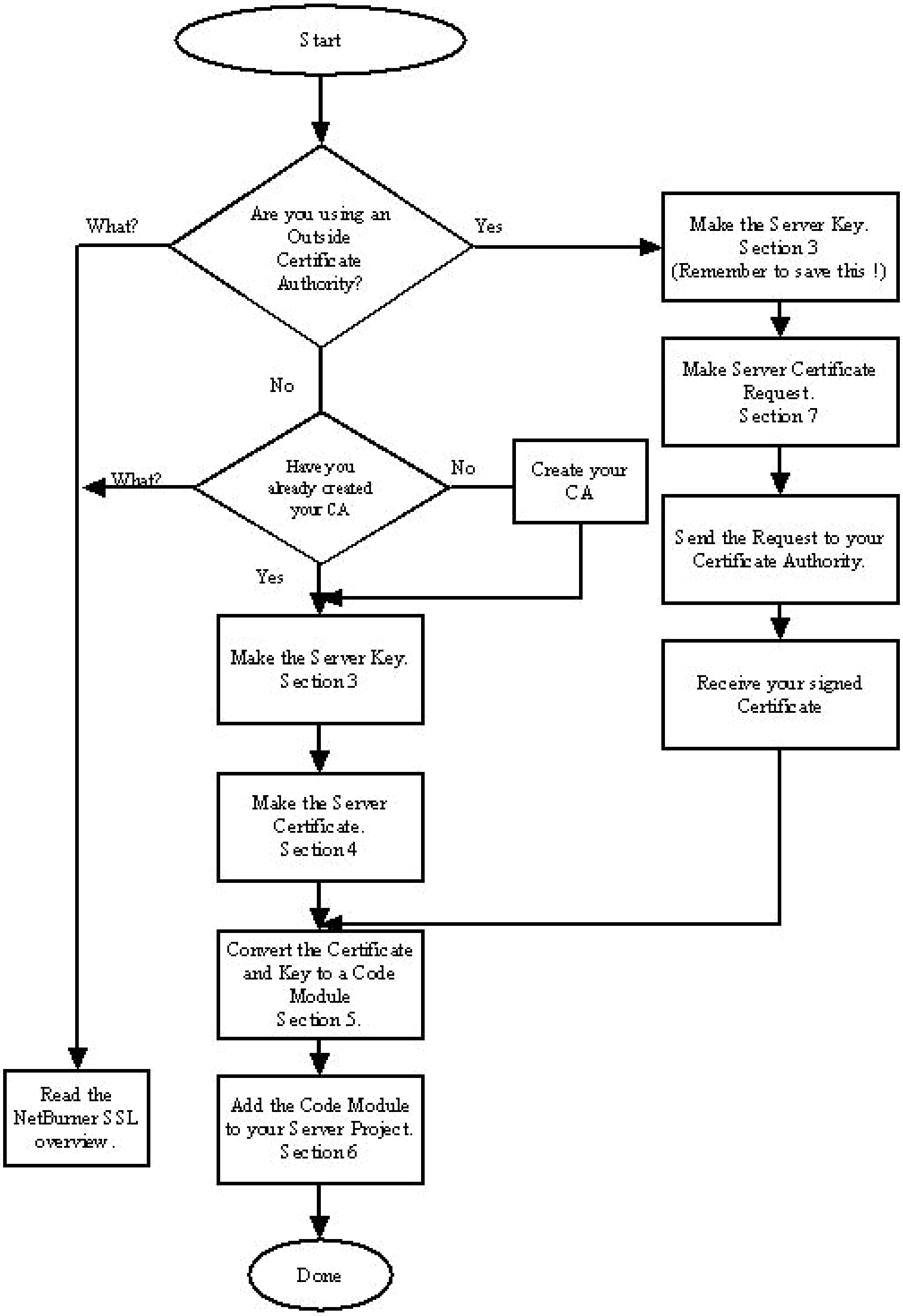
- Open a command prompt/DOS window
- Navigate to the directory that you want to house your device files
- To make a Device Certificate Request file, execute the command (and press the Enter key when finished):

```
openssl req -new -key devicename.key -out devicename.csr
```

- Send this devicename.csr to the CA that will create your certificate.

Warning: If you lose the devicename.key file associated with this particular device, then you will not be able to use the certificate file they send back.

4.4. Creating a Code Module - SSL Server Key & Certificate - Diagram


Start
Are you using an Outside Certificate Authority?
What?
Yes
No
Make the Server Key. Section 3 (Remember to save this !)
Make Server Certificate Request. Section 7
Have you already created your CA
What?
No
Yes
Create your CA
Send the Request to your Certificate Authority.
Receive your signed Certificate
Make the Server Key. Section 3
Make the Server Certificate. Section 4
Convert the Certificate and Key to a Code Module Section 5.
Add the Code Module to your Server Project. Section 6
Read the NetBurner SSL overview.
Done

4.5. Creating a Code Module for SSL Client Certificates

Introduction

The NetBurner SSL library provides some open source tools for the generation and maintenance of SSL Keys and Certificates. These key and certificate management tools are based on the fine openssl package available from http://www.openssl.org. These tools are subject to the openssl License.

The embedded SSL Library code is derived from other sources and is subject to the standard NetBurner license (a copy is in your C:\Nburn\docs directory) that holds a list of certificate authorities that will be acceptable signers for SSL client connections.

Determining what Certificates you need

If you do not know what certificates are needed you should read the Easy SSL Overview document in this section. You want to include the certificates from the CA(s) that will be signing the server certificates that your SSL client will be connecting to.

Testing your Certificates

For each CA you want to trust you need to obtain a certificate. These certificates should be in the **X509 format** with the extension .crt (see RFC 3280). General purpose openssl documents can be found at: http://www.openssl.org. To test your certificate:

- Make sure the **openssl.exe** provided with the NetBurner SSL library is in your path (by default it installs in C:\Nburn\pcbin)
- Open a command prompt/DOS window in your project directory
- Execute the command (and press the Enter key when finished):

  ```
  openssl x509 –in yourcert.crt –text
  ```

- This command should dump the cert contents
- If this command fails then you will need to convert the format. Note: One way to do this is to use the openssl tools.
- Certificates can be in DER, NET and/or PEM formats. Warning: The Certificates Must be in the PEM format. To convert a Certificate from DER to PEM, just execute the following command (and press the Enter key when finished):

```
openssl x509 -inform der -in server_cert.crt –out server_cert_in_pem.crt
```

Creating a CA List file

This step creates a cpp file that holds all of the CA certificates you will accept. To create a CA List file:

- Open a command prompt/DOS window in your project directory
- Execute the command (and press the Enter key when finished):

```
openssl x509 –out ccerts.cpp -nburnccerts cerfile1 cerfile2
             ...lastcertfile
```

- This command should build the file ccerts.cpp (Important: Make sure that this file (ccerts.cpp) has all of your certs in it.)
- If using NBEclipse, import this file into your project directory. If using command line tools, copy to your project directory and add this file to your project's makefile

4.6. StartHTTPs

Synopsis:

```
void StartHTTPs( WORD ssl_port=443, WORD http_port=80 );
```

Description:

This function starts the secure web server. By default it will listen on port 80 for unencrypted connections, and port 443 for encrypted connections.

Parameters:

Type	Name	Description
Word	ssl_port=443	Port 443 is the standard HTTPS port.
Word	http_port=80	Port 80 is the standard HTTP Server port.

Returns:

Nothing --- This is a void function

4.7. SSL_accept

Synopsis:

```
int SSL_accept( int fdListen, IPADDR * address, WORD * port, WORD timeout );
```

Description:

Similar to TCP accept() but used for SSL connections.

Parameters:

Type	Name	Description
int	fdListen	The file descriptor of the TCP listening socket.
IPADDR	*address	The IPADDR variable to hold the address of the connecting computer.
WORD	*port	The WORD variable to receive the remote port of this connection.
WORD	timeout	The number of ticks to wait for a connection.

Return Values:

> 0 --- The file descriptor of the connected SSL socket
TCP_ERR_TIMEOUT --- Underlying TCP system timed out
TCP_ERR_NOCON --- The underlying TCP connection failed to negotiate
TCP_ERR_CLOSING --- The underlying TCP fd was closing
TCP_ERR_NOSUCH_SOCKET --- The fd listen socket was invalid
TCP_ERR_NONE_AVAIL --- No free sockets to return
TCP_ERR_CON_RESET --- The connection was reset by the remote device
TCP_ERR_CON_ABORT --- The connection was aborted by the remote device
SSL_ERROR_FAILED_NEGOTIATION --- The SSL system failed to successfully negotiate a connection
SSL_ERROR_HASH_FAILED --- The connection failed the startup hash test
SSL_ERROR_WRITE_FAIL --- The connection failed to write out a full record

4.8. IsSSLfd

Synopsis:

```
BOOL IsSSLfd( int fd );
```

Description:

This Boolean function is used to determine if the fd (file descriptor) is an SSL connection (i.e. Is the file descriptor an SSL FD or some other kind?). It can be used by things like the callback functions of the web server to determine how secure the fd is before sending sensitive information over it.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.

Return Values:

TRUE --- If it is an SSL file descriptor
FALSE --- If it is not an SSL file descriptor, or on error

4.9. SSL_GetSocketRemoteAddr

Synopsis:

```
IPADDR SSL_GetSocketRemoteAddr( int fd );
```

Description:

This function returns the remote address of this connected socket. This function is used to retrieve the remote address of an SSL fd. This function will also work correctly if you pass in a fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.

Return Values:

remote ---The IP Address of the TCP or SSL connection
0 --- Otherwise

4.10. SSL_GetSocketRemotePort

Synopsis:

```
WORD SSL_GetSocketRemotePort( int fd );
```

Description:

This function returns the remote port of this connected socket. This function is used to retrieve the remote port of an SSL fd. This function will also work correctly if you pass in a fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.

Return Values:

remote --- The port number of the TCP or SSL connection
0 --- Otherwise

4.11. SSL_GetSocketLocalAddr

Synopsis:

```
IPADDR SSL_GetSocketLocalAddr( int fd );
```

Description:

This function returns the local address of this connected socket. This function is used to retrieve the local address of an SSL fd. This function will also work correctly if you pass in a fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.

Return Values:

local --- The IP address of the TCP or SSL connection
0 --- Otherwise

4.12. SSL_GetSocketLocalPort

Synopsis:

```
WORD SSL_GetSocketLocalPort( int fd );
```

Description:

This function returns the local port of this connected socket. This function is used to retrieve the local port of an SSL fd. This function will also work correctly if you pass in an fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.

Return Values:

local ---The port number of the TCP or SSL connection
0 --- Otherwise

4.13. SSL_setsockoption

Synopsis:

```
int SSL_setsockoption( int fd, int option );
```

Description:

This function will set the socket option.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.
int	option	The socket option.

Returns:

> 0 --- The file descriptor of the connected SSL socket
TCP_ERR_TIMEOUT --- Underlying TCP system timed out
TCP_ERR_NOCON --- The underlying TCP connection failed to negotiate
TCP_ERR_CLOSING --- The underlying TCP fd was closing
TCP_ERR_NOSUCH_SOCKET --- The fd listen socket was invalid
TCP_ERR_NONE_AVAIL --- No free sockets to return
TCP_ERR_CON_RESET --- The connection was reset by the remote device
TCP_ERR_CON_ABORT --- The connection was aborted by the remote device
SSL_ERROR_FAILED_NEGOTIATION --- The SSL system failed to successfully negotiate a connection
SSL_ERROR_HASH_FAILED --- The connection failed the startup hash test
SSL_ERROR_WRITE_FAIL --- The connection failed to write out a full record
SSL_ERROR_CERTIFICATE_UNKNOWN SSL --- Received a certificate it could not decode
SSL_ERROR_CERTIFICATE_NAME_FAILED The connected name did not match common_name
SSL_ERROR_CERTIFICATE_VERIFY_FAILED --- The server returned a certificate that we did not trust

4.14. SSL_clrsockoption

Synopsis:

```
int SSL_clrsockoption( int fd, int option );
```

Description:

This function will clear the socket option.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.
int	option	The socket option.

Returns:

> 0 ---The file descriptor of the connected SSL socket
TCP_ERR_TIMEOUT --- Underlying TCP system timed out
TCP_ERR_NOCON --- The underlying TCP connection failed to negotiate
TCP_ERR_CLOSING --- The underlying TCP fd was closing
TCP_ERR_NOSUCH_SOCKET --- The fd listen socket was invalid
TCP_ERR_NONE_AVAIL --- No free sockets to return
TCP_ERR_CON_RESET --- The connection was reset by the remote device
TCP_ERR_CON_ABORT --- The connection was aborted by the remote device
SSL_ERROR_FAILED_NEGOTIATION --- The SSL system failed to successfully negotiate a connection
SSL_ERROR_HASH_FAILED The connection failed the startup hash test
SSL_ERROR_WRITE_FAIL The connection failed to write out a full record
SSL_ERROR_CERTIFICATE_UNKNOWN SSL --- Received a certificate it could not decode
SSL_ERROR_CERTIFICATE_NAME_FAILED --- The connected name did not match common_name
SSL_ERROR_CERTIFICATE_VERIFY_FAILED --- The server returned a certificate that we did not trust.

4.15. SSL_getsockoption

Synopsis:

```
int SSL_getsockoption( int fd );
```

Description:

This function will get the socket option.

Parameters:

Type	Name	Description
int	fd	The file descriptor to test.

Returns:

> 0 --- The file descriptor of the connected SSL socket
TCP_ERR_TIMEOUT --- Underlying TCP system timed out
TCP_ERR_NOCON --- The underlying TCP connection failed to negotiate
TCP_ERR_CLOSING --- The underlying TCP fd was closing
TCP_ERR_NOSUCH_SOCKET --- The fd listen socket was invalid
TCP_ERR_NONE_AVAIL --- No free sockets to return
TCP_ERR_CON_RESET --- The connection was reset by the remote device
TCP_ERR_CON_ABORT --- The connection was aborted by the remote device
SSL_ERROR_FAILED_NEGOTIATION --- The SSL system failed to successfully negotiate a connection
SSL_ERROR_HASH_FAILED --- The connection failed the startup hash test
SSL_ERROR_WRITE_FAIL --- The connection failed to write out a full record
SSL_ERROR_CERTIFICATE_UNKNOWN SSL --- Received a certificate it could not decode
SSL_ERROR_CERTIFICATE_NAME_FAILED --- The connected name did not match common_name
SSL_ERROR_CERTIFICATE_VERIFY_FAILED --- The server returned a certificate that we did not trust

4.16. SSL_connect

Synopsis:

```
int SSL_connect( IPADDR ip, WORD local_port, WORD remote_port,
                 DWORD timeout, const char * common_name );
```

Description:

Make an outgoing SSL connection to a SSL server or peer.

Parameters:

Type	Name	Description
IPADDR	ip	The address to connect to.
WORD	local_port	The local port to use. **Note:** 0 will pick a local port.
WORD	remote_port	The port to connect to.
DWORD	timeout	The number of ticks to wait for a connection.
const char	*common_name	The common name to use for checking certificate validity. **Note:** Passing in NULL will accept any connection.

Return Values:

```
> 0 ---The file descriptor of the connected SSL socket
TCP_ERR_TIMEOUT --- Underlying TCP system timed out
TCP_ERR_NOCON --- The underlying TCP connection failed to negotiate
TCP_ERR_CLOSING --- The underlying TCP fd was closing
TCP_ERR_NOSUCH_SOCKET --- The fd listen socket was invalid
TCP_ERR_NONE_AVAIL --- No free sockets to return
TCP_ERR_CON_RESET --- The connection was reset by the remote device
TCP_ERR_CON_ABORT --- The connection was aborted by the remote device
SSL_ERROR_FAILED_NEGOTIATION --- The SSL system failed to successfully
                                 negotiate a connection
SSL_ERROR_HASH_FAILED --- The connection failed the startup hash test
SSL_ERROR_WRITE_FAIL --- The connection failed to write out a full record
SSL_ERROR_CERTIFICATE_UNKNOWN SSL --- Received a certificate it could not
                                      decode
SSL_ERROR_CERTIFICATE_NAME_FAILED --- The connected name did not match
                                      common_name
SSL_ERROR_CERTIFICATE_VERIFY_FAILED --- The server returned a certificate
                                        that we did not trust
```

4.17. SSL_SendMail

Synopsis:

```
int SSL_SendMail( IPADDR smtp_server,
        PCSTR userid,
        PCSTR pass,
        PCSTR from_addr,
        PCSTR to_addr,
        PCSTR subject,
        PCSTR textbody,
        BOOL STARTTLS = FALSE );
```

Description:

Send an email message using SSL encryption. This function is very similar to the unencrypted SendMail() function.

The initial connection to the SMTP server can happen in one of two ways:
1. An SSL connection is negotiated immediately.
2. A TCP connection is opened first, and a SSL connection is negotiated
   when the SMTP server sends a STARTTLS command to the NetBurner device.

Setting STARTTLS = TRUE enables the second mode of operation. The default is FALSE. Most SMTP serves will support both modes, but there are no rules as to whether or not both modes must be supported.

Type	Name	Description
IPADDR	smtp_server	Name or IP Address of the SMTP Server
PCSTR	userid	SMTP account user id string
PCSTR	pass	SMTP account password
PCSTR	from_addr	From email address.
PCSTR	to_addr	Send to email address
PCSTR	subject	email subject
PCSTR	textbody	email body content
BOOL	STARTTLS	Enable/disable STARTTLS functionality. Default is disabled.

Return Values:

0 --- Send failed
1 --- Send was successful

5. SSH Library (Secure Shell)

The NetBurner SSH package is sold as a licensed option only, and is not part of the standard development kit package. Please contact our Sales Department for purchase information.

5.1. Features

SSH supports the following :

- SSH Server Mode
- SSHV1 and SSHV2
- Encryption ciphers of DES, 3DES, IDEA and Arcfour.
- Public-key cryptography: RSA, Diffie-Hellman, DSA.
- One-way hash functions: MD2, MD4, MD5 or SHA1.
- Permanent RSA and DSA 512 bit private keys installed by default.

Compile Options:

- Encryption ciphers Twofish (128/256), Blowfish not compiled in by default. Enabling this option will increase the size of the library.
- Conditional compilation tags to use 512, 1024 or 2048 bit keys separately for RSA and DSA private keys.
- User supplied private key accessor.
- User supplied password authentication.

5.2. Performance

The fastest performance will be on NetBurner devices based on the 5270 and 5234 processors. When the server is first contacted it requires an exchange of session keys. The generation of this session key uses various amounts of time for various size keys and module types. These are typical average times in seconds:

	Modules		
Key Size (Width)	MOD5270 MOD5234 CB34EX PK70	MOD5282	MOD5272 SB72(EX)
512	7	15	30
1024	10	21	41
2048	31	64	134

Including a single SSH socket increases the size of the application by at least 110K bytes for SSH protocol, encryption and supporting mathematics routines

5.3. System Library Requirements

Important: Before you compile any programs, edit predef.h (located in C:\Nburn\include) with any text editor, and uncomment the lines for SSH support, SSL too if used. Any changes to system files require that the system libraries be rebuilt. From NBEclipse, select NBEclipse -> Rebuild System Files.

```
/*
 * SSH Supported
 *    Should be defined when SSH is included in library
 */

/* #define NB_SSH_SUPPORTED   ( 1 ) */
```

Header File

```
#include <ssh\ssh.h>      // Found in C:\Nburn\include\ssh
```

SSH Server Functions

`SshAccept` SSH equivalent of the TCP accept function

`SshSetUserAuthenticate` Assigns the callback function used by the SSH library to authenticate a user login.

`SshSetUserGetKey` Specify the callback function to use to get the user supplied key. The callback will be executed anytime SshAccept() or SshNegotiateSession() is called. If you do not provide your own callback function, the NetBurner SSH Library keys will be used by default.

`SshValidateKey` Validates a PEM encoded key.

`SshWritePublicKey` Write the current SSH public key to a file descriptor. Can be used to display a public key that can be copied to a SSH client program.

`SshPrintStatistics` Prints amounts and processing times for each SSH packet type.

`SshNegotiateSession` Negotiates SSH session on accepted socket. Designed for use with the command processor provided with the library.

Sockets are released using the file descriptor close function: `close()`.

5.4. Protection

Secure Shell protects against :

- IP spoofing: A remote host sends out packets which pretend to come from another, trusted host. SSH even protects against a spoofer on the local network, who can pretend he is your router to the outside.
- IP source routing: A host can pretend that an IP packet comes from another, trusted, host.
- DNS spoofing: An attacker forges domain name server records, then intercepts plain text passwords and other data.
- Attacks based on listening to X authentication data and spoofed connection to the X11 server.

In other words, SSH never trusts the net; somebody hostile who has taken over the network can only force SSH to disconnect, but cannot decrypt or play back the traffic, or hijack the connection.

5.5. SSH Keys

The SSH support library, sshLibrary.a, contains by default a RSA and DSA private key. The example in directory \nburn\examples\ssh\SSHFactoryApp uses the SSH library function calls, the web server interface, and Embedded Flash File System (EFFS) to acquire, validate and store a user created private key.

5.5.1. NetBurner Default Keys

The SSH key and size is determined by the #define value set in \nburn\include\ssh\ssh.h. An excerpt from the header file is shown below:

```
/* Default static key size, no choice is 1024 */
#define SSH_RSA_KEY_DEFAULT_512                 (  512 )
/* #define SSH_RSA_KEY_DEFAULT_1024             ( 1024 ) */
/* #define SSH_RSA_KEY_DEFAULT_2048             ( 2048 ) */

#define SSH_DSS_KEY_DEFAULT_512                 (  512 )
/* #define SSH_DSS_KEY_DEFAULT_1024             ( 1024 ) */
/* #define SSH_DSS_KEY_DEFAULT_2048             ( 2048 ) */
```

Any change to the ssh.h file requires that you rebuild the system libraries. This can be done in NBEclipse by selecting NBEclipse -> Rebuild All Libraries.

5.5.2. Using a Custom Key

An application must use the `SshSetUserGetKey()` function to specify a callback function that the SSH library can use during runtime to access the user created key. The SSH library must have access to the key before an application can call SshAccept() or SshNegotiateSession(). See the section on the `SshSetUserGetKey()` function for more details.

5.6. Creating a SSH Server

Calling any of the SSH functions will include the SSH library. When the first connection is established, SSH server session key task will be created at priority SSH_TASK_PRIORITY, which is set to 56 by default. The additional libraries needed are sshLibrary and debugLibrary. Debug libraries have a DB prepended to their names.

5.7. Implementing a Login User Name and Password

If a SSH User Authentication callback function is defined, the SSH library will call it before allowing an incoming SSH connection to be established. To activate this feature, create a function of type `SshSetUserAuthenticate()` in your application, then instruct the SSH library to call this function with `SshSetUserAuthenticate().` See the section on `SshSetUserAuthenticate()` for additional information.

5.8. Example Applications

SSH example applications are located in the \nburn\examples\ssh directory.

5.9. Recommended Reading

For an excellent overview of SSH:
SSH, The Secure Shell: The Definitive Guide, Second Edition, by Daniel J. Barrett, Richard E. Silverman, Robert G. Byrnes, May 2005, O'Reilly Media, Inc.

For quick review of SSH:
URL: http://en.wikipedia.org/wiki/Secure_shell

5.10. SSH_accept

Synopsis:

```
int SSH_accept( int fdListen, IPADDR * address, WORD * port, WORD timeout )
```

Description:
This function accepts a SSH connection from a listening TCP socket.

Parameters:

Type	**Name**	**Description**
int	fdListen	The file descriptor of the listening socket
IPADDR	*address	Pointer to the variable that will be written with the IP address of the connecting host.
WORD	*port	Pointer to the variable that will be written with the TCP port number of the connecting host.
WORD	timeout	The number of timer ticks to wait for a connection.

Return Values:
Returns the file descriptor of the connected SSH socket, or a negative number on error:

TCP_ERR_TIMEOUT	Underlying TCP system timed out
TCP_ERR_NOCON	The underlying TCP connection failed to negotiate
TCP_ERR_CLOSING	The underlying TCP fd was closing
TCP_ERR_NONE_AVAIL	No free sockets to return
TCP_ERR_CON_RESET	The connection was reset by the remote device
TCP_ERR_CON_ABORT	The connection was aborted by the remote device
TCP_ERR_NOSUCH_SOCKET	The fd listen socket was invalid
SSH_ERROR_FAILED_NEGOTIATION	SSH failed to successfully negotiate a connection

5.11. SshSetUserAuthenticate

Synopsis:

```
void SshSetUserAuthenticate( sshUserAuthenticateFn sshUserAuthenticateFnPtr )
```

Description:
Assigns a user created callback function that will be called by the SSH library to authenticate a username and password during login.

Parameters:

Type	Name	Description
Pointer to a function	sshUserAuthenticateFnPtr	Pointer to a function

Returns: Nothing

Type definition for a function pointer to a user-provided SSH authentication function:

```
typedef int ( *sshUserAuthenticateFn )
            ( const char* usernamePtr, const char* passwordPtr );
```

Type	Name	Description
const char *	usernamePtr	User name in plain text
const char *	passwordPtr	Password in plain text

Returns:
1 = Authentication passed, any other value is a failure.

SshSetUserGetKey

Synopsis:

```
void SshSetUserGetKey( sshUserGetKeyFn sshUserGetKeyFnPtr )
```

Description:
This function is used to specify a user created/defined key to be used in SSH connections.

Parameters:

Type	Name	Description
Pointer to a function	sshUserGetKeyFnPtr	User provided key provision routine called as a pointer to a function. See typedef explanation below.

Returns: Nothing

Type definition for a function pointer to a user-provided SSH key function:

To use a user-provided SSH key an application must create and execute a function of the following type. The buffer must contain an opens(L|H) key pair, PEM encoded, unencrypted, with no pass phrase. The buffer containing the key must not be deallocated.

```
typedef int ( *sshUserGetKeyFn )( int keyRequested,
                                  const unsigned char** keyBufferPtr,
                                  int* keyLengthPtr );
```

Type	Name	Description
int	keyRequested	Type of key requested: SSH_KEY_RSA or SSH_KEY_DSS.
const unsigned char	**keyBufferPtr	Buffer containing the key.
int	*keyLengthPtr	Size of key in 8-bit bytes.

Returns:
0 if key and length is valid, or a -1 if the key requested is not available

Example:
In this example we create a function named `SshUserGetKey()` that will be called in main.cpp. It returns the key requested, which is part of the NV_Settings structure that is read in on startup from the flash memory file system. Note DSS is the security algorithm abbreviation that uses the DSA key.

This function is an excerpt from \nburn\examples\ssh\SSHFactoryApp\sshuser.cpp. The example also uses the Web server interface to read in a key, `SshValidateKey()` to validate the key, and stores it in the flash file system which at startup is copied into the structure NV_Settings.

```
int SshUserGetKey( int keyRequested, const unsigned char** keyBufferPtr,
                   int* keyLengthPtr )
{
   int keyValid = -1;

   if ( ( keyBufferPtr != NULL ) && ( keyLengthPtr != NULL ) )
   {
      if ( ( keyRequested == SSH_KEY_RSA ) &&
            ( NV_Settings.SshKeyRsaSource != 0 ) &&
            ( NV_Settings.SshKeyRsaLength > 0 ) )
      {
            *keyLengthPtr = NV_Settings.SshKeyRsaLength;
            *keyBufferPtr = (const unsigned char*)gSshRsaKeyPemEncoded;
            keyValid = 0;
      }
      else if ( ( keyRequested == SSH_KEY_DSS ) &&
            ( NV_Settings.SshKeyDsaSource != 0 ) &&
            ( NV_Settings.SshKeyDsaLength > 0 ) )
      {
            *keyLengthPtr = NV_Settings.SshKeyDsaLength;
            *keyBufferPtr = (const unsigned char*)gSshDsaKeyPemEncoded;
            keyValid = 0;
      }
   }  /* End if ( ( keyBufferPtr != NULL ) && ( keyLengthPtr != NULL ) ) */

   return keyValid;
}
```

5.12. SshValidateKey

Synopsis:

```
BOOL SshValidateKey( const char* candidateKey, int candidateKeySize,
                     int* keyTypePtr );
```

Description:
Validate a PEM encoded openSS(L|H) key

Parameters:

Type	Name	Description
const char ptr	*candidateKey	Pointer to a PEM encoded key buffer.
int	candidateKeySize	Size of key in 8-bit bytes.
int	*keyTypePtr	Pointer to variable to store the type of key detected: 0 = None, 1 = RSA, 2 = DSA.

Return Values:

Returns TRUE if valid RSA or DSA key is detected, otherwise returns FALSE.

5.13. SshPrintStatistics

Synopsis:

```
void SshPrintStatistics ( int secureFd )
```

Description:
Used for application debugging and troubleshooting. Calling this functions displays the SSH statistics to the debug serial port using the stdio iprintf() function.

Parameters:

Type	**Name**	**Description**
int	secureFd	Secure SSH file descriptor

Return Values:

None

5.14. SshNegotiateSession

Synopsis:

```
void* SshNegotiateSession(  FILE* acceptedSocketFILEptr  )
```

Description:

Negotiates a SSH connection using the FILE object which has already been accepted. The yet unsecured file descriptor is replaced in the object with the secure file descriptor if the function succeeds.

Parameters:

Type	Name	Description
FILE	*acceptedSocketFilePtr	Not yet secure socket FILE pointer

Return Values:

Returns a pointer to a SshSession object, or NULL on failed negotiation.