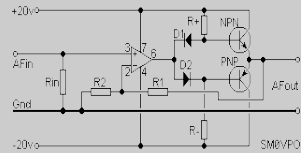
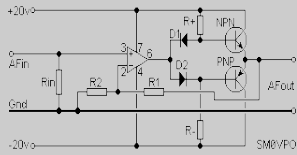
# Understanding SSL/TLS




## or What is an SSL Certificate and What Does It Do for Me?

J.K. Harris
Electrical and Computer Engineering
Virginia Tech
Oct 2008

# Understanding SSL/TLS




What is It?

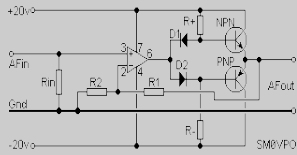
How Does It Work?

Why is It Important?  (What does it do?)

But, Most Importantly

–> ***Things the average user should know!***

# What is It?



Something about encryption of Web pages

https://...

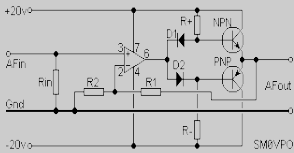
The “lock icon” at the bottom of your browser

–> Can Safely Type in Your Credit Card Number!

(...are you *sure* its safe?)

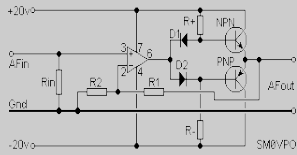
In short, very few people know what SSL/TLS is!

# How Does It Work?




- Based on the RSA algorithm
- Is a public key cryptography system
- It takes a little math to understand this (I'll keep the math to *very* little!)

# How Does It Work?



A little math:

For *properly chosen* (*e*, *d*, *n*)

–> Functions are inverses of each other!

$$c = m^e \bmod n$$

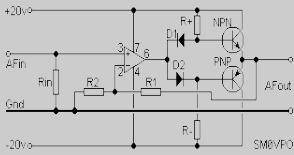
$$m = c^d \bmod n$$

Reference:

http://en.wikipedia.org/wiki/Rsa

I.e., wikipedia “RSA”

# How Does It Work?



(Some hand waving: *e* is not critical, almost all RSA use *e* = 65537)

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

Think:

*m* = message

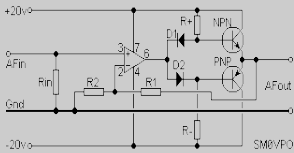
*c* = cypher (encrypted message)

We call:

*n* –> the Public key

*d* –> the Private key

# How Does It Work?



For *properly chosen* (*e*, *d*, *n*)

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

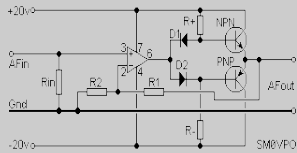
We call:

*n* –> the Public key

*d* –> the Private key

–> Given *n* can **not** (easily) find *d*!

# Encryption



Standard usage:

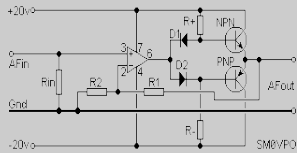
Alice –> Bob

- Bob *properly chooses* (*e*, *d*, *n*)
- Bob sends **public** key *(n)* to Alice (***How?***)
- Alice encrypts her message *(m)*
- Alice sends cypher *(c)* to Bob
- Bob uses his **private** key *(d)* to decrypt

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

# Encryption



Standard usage:

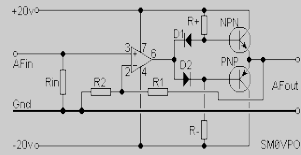
$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

**Very Important and Subtle Point:**

- Bob sends **public** key *(n)* to Alice (***How?***)

# Digital Signatures



Signing Digital Documents:

Digital Signatures

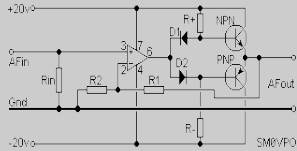
Bob –> Alice

- Bob *properly chooses* (*e*, *d*, *n*)
- Bob sends **public** key *(n)* to Alice (***How?***)
- Bob encrypts his document *(c)* using his **private** key *(d)* giving cypher *(m)*
- Bob sends *(m)* to Alice
- Alice uses Bob's **public** key *(n)* to decrypt

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

# Digital Signatures



Signing Digital Documents:

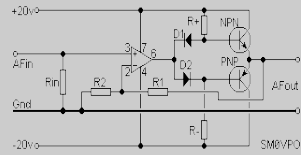
$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

**Very Important and Subtle Point:**

- Bob sends **public** key *(n)* to Alice (***How?***)

# Digital Signatures



Signing Digital Documents:

$$c = m^e \bmod n$$

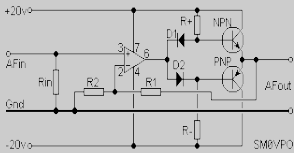
$$m = c^d \bmod n$$

Work equations in “reverse”

Alice knows that Bob sent the message because his public key decrypted a message that could *only* be created using Bob's private key.

(This assumes that Alice somehow has Bob's public key.)

# How Does It Work?



Points to remember:

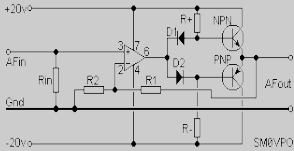
Equations in the “forward” direction –>
encryption

Equations in the “reverse” direction –>
message signing (digital signature)

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

# How Does It Work?



**Very Important and Subtle Point:**

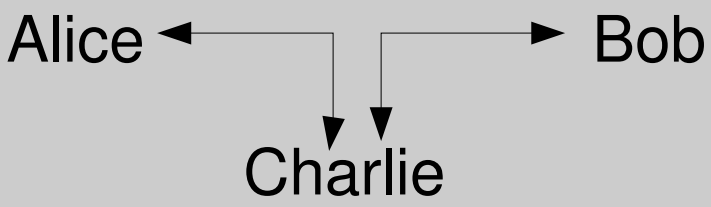
- Bob sends **public** key *(n)* to Alice (***How?***)

Why? "Man in the Middle Attack"

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

# Man in the Middle Attack

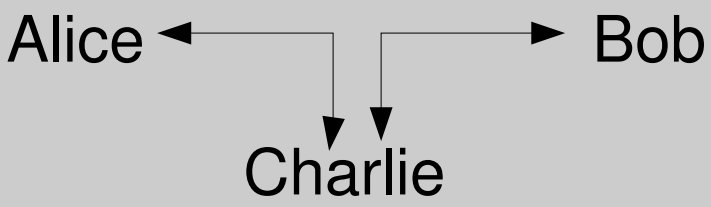



$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

- Charlie has full control of the “wire”
- Charlie sends Alice Charlie's Public key in place of Bob's Public key
- Charlie decrypts Alice's message using his own Private key
- Charlie uses Bob's Public to send him any message he wants

# Man in the Middle Attack



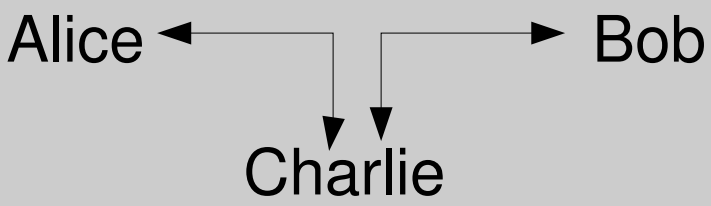

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

Because Alice does not have Bob's public key

- Alice has no way of knowing that she is not communicating with Bob
- Bob has no way of knowing that the message did not come from Alice
- Charlie can do anything he wants

# Man in the Middle

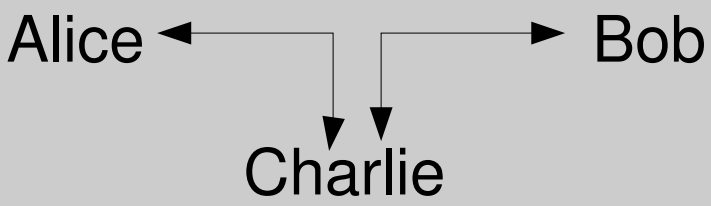



$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

**How to get Bob's public key safely to Alice?**

# Man in the Middle



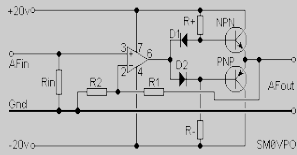

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

**How to get Bob's public key safely to Alice?**

–> “Trusted Third Party” (+ lots of confusion)

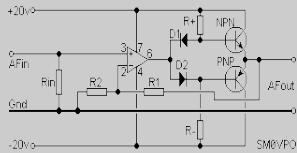
# Trusted Third Party



Introducing “Trusted Third Party”, Vera:

- Vera has her own public and private key
- Vera has her public key widely distributed in a fashion that everyone believes *(**How?**)*
  - Generally, everyone's web browser has them “built in” (Internet Explorer, FireFox, Safari)
- (Think Vera –> Verisign)

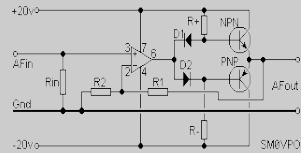
# Trusted Third Party



- Bob sends his public key to Vera
  - Certificate Signing Request (.csr)
- Vera verifies that Bob

  *“is who he says he is” (**How?**)*
  - This is what you are paying for!
- Vera “digitally signs” and returns this to Bob
  - SSL/TLS Certificate (.crt)

# Trusted Third Party



Vera

Generates
public key
private key
(e, d, n)

public key *(n)*
(somehow
widely published)

Alice
"built into" the
web browser
aka root certificates

Bob

Generates
public key
private key
(e, d, n)

public key *(n)*
certificate
signing request
.csr

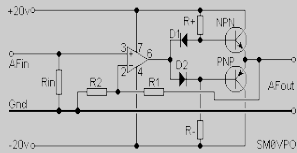
Vera

verifies Bob's identity
(calls on the phone?)

"digitally signs"
SSL/TLS certificate
.crt

Bob

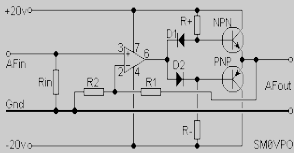
# Trusted Third Party



Now the communications between Alice and Bob:

- Alice asks Bob for his SSL/TLS certificate
- Alice checks to see if she can verify the digital signature using Vera's public key
- If the digital signature verifies, *and* Alice trusts Vera, then Alice believes that the SSL/TLS certificate came from Bob – No one else could have generated Vera's digital signature
- “Inside” of this SSL/TLS certificate is Bob's public key!
- Alice now has Bob's public key and can proceed as before

# Trusted Third Party



Alice → Bob

*(m)*

*(m)*

Alice

Bob

hello

SSL/TLS certificate .crt

check digital signature using Vera's "built in" public key

use Bob's public key found in certificate

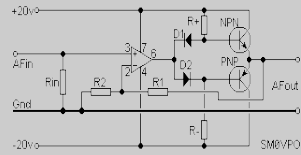
$c = m^e \bmod n$

RSA encrypted message *(c)*

$m = c^d \bmod n$

*(m)*

# That's How SSL/TLS Works



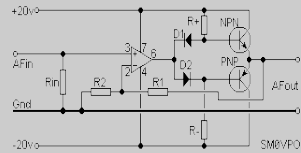

That's it! That's how SSL/TLS works!

... Simple, right?

Depends upon:

- Trusting Vera:
  - Vera actually verifies that Bob “is who he says he is”
  - Distribution of Vera's public keys (root certificates)

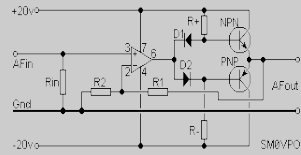
# That's How SSL/TLS Works




But, think about this a little:

- In some sense, we have traded the problem of getting Bob's public key to Alice, for the problem of getting Vera's public key to Alice.

# That's How SSL/TLS Works

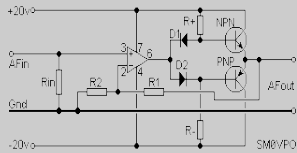



But, think about this a little:

- In some sense, we have traded the problem of getting Bob's public key to Alice, for the problem of getting Vera's public key to Alice.
- But, there is only one Vera, and lots of Bobs!

So, we still have the problem, but we have made the problem much smaller, and possibly tractable.

# What is It?

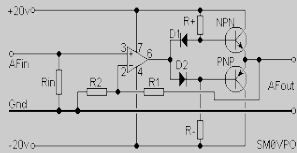



- Connection is Encrypted – but that's easy
- **Verification of "the other end"**
  - (via the trusted third party)
  - ***This is the real reason for SSL/TLS!!!***

... Is it any thing else?

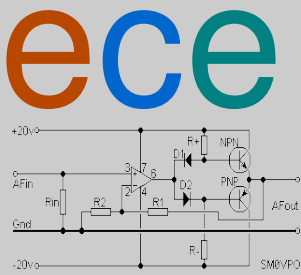
  - ***NO!***

# Lingering Issues




- Trust ends where the credit card begins
- Who's certificate is this?
- Revocation Lists
- Root certificate poisoning
- Your own government

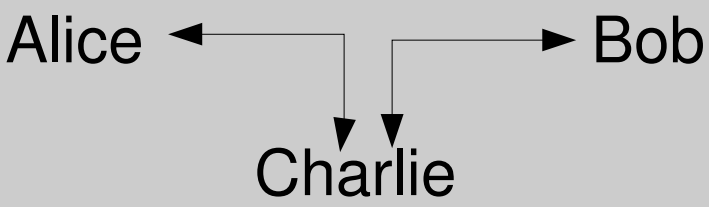
# Trust Ends Where the Credit Card Begins




- All SSL/TLS tells you is that you have an encrypted connection to whomever was issued that certificate
- Do you *really* trust the person at the other end of the connection?
- Rules governing “verification” (you are who you say you are) are being weakened
  - Due to high volume of certificates issued
  - No human in the loop!!!

# Who's Certificate is This?

A.K.A. DNS / URL spoofing

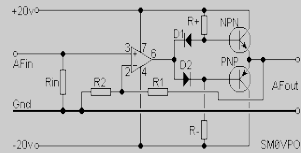


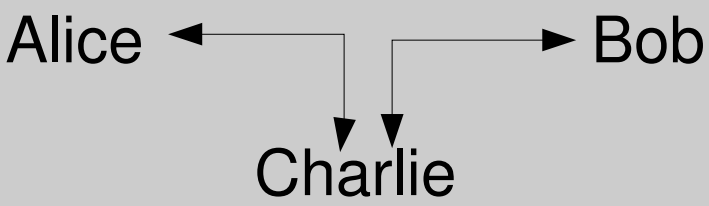

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

- Charlie has his own certificate signed by Vera
- Charlie hands *his* certificate to Alice
- How is Alice to know its not Bob's certificate?

# Who's Certificate is This?





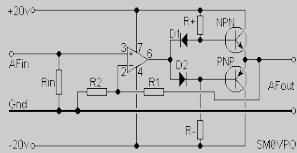

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

- Charlie has his own certificate signed by Vera
- Charlie hands *his* certificate to Alice
- How is Alice to know its not Bob's certificate?

**–> Certificate has Bob's "name" on it**

# Who's Certificate is This?

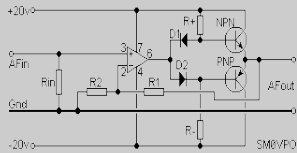


Certificate has Bob's “name” on it?

- What *is* Bob's “name”?
- Bob's “name” is his DNS name

–> Always check the URL!

# Always Check the URL?

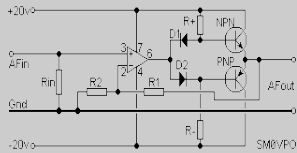


Is checking the URL sufficient?

What about “similar names”

- www.amazon.com –> www.amazone.com
- www.capitalone.com –> www.capitolone.com
- www.there.com –> www.their.com
- www.amazon.com –> www.amazon.çom

# Always Check the URL?

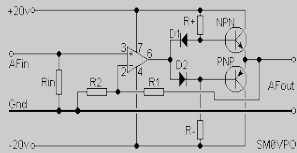


Is checking the URL sufficient?

What about “similar names”

- www.amazon.com –> www.amazone.com
- www.capitalone.com –> www.capitolone.com
- www.there.com –> www.their.com
- www.amazon.com –> www.amazon.çom

# Revocation Lists



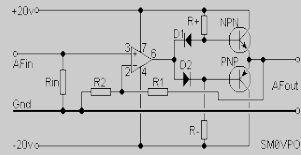
Revocation Lists and Short “valid” times

- Certificates usually valid only for 1 – 2 years
- Widely published list of certificates that have been revoked
  - Minimize the amount of time a “bad guy” can use his certificate
  - Quickly revoke bad guy's certificate

Largely unused!

- I defy you to find these “widely published” revocation list!!!

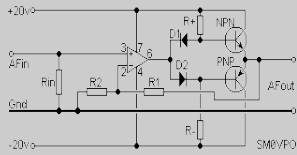
# Root Certificate Poisoning




This is a big deal!  Most people just ignore this.

- Microsoft people clicking “*ok*” might *install* a bad guy's root certificate
- Leave your desk, someone could easily, in a couple of clicks, install his own root certificate
- Product update channels get poisoned along the way

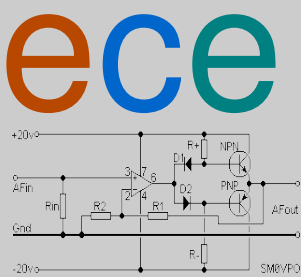
# Your Own Government



(Get out your tin-foil hats!)

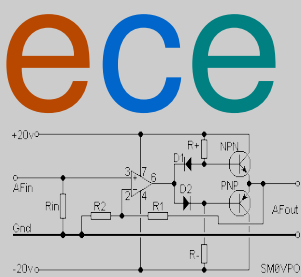
- The US Government's encryption policy: “Strong enough so that citizens can't listen to other citizens, but not so strong that the Government can't”
- You think the Government does not already has Verisign's private key?
- DES – Government pushed encryption, now known to have “Government exploitable” weaknesses
- Recent A.Q. Kahn allegations – implies that it took ~3 years to break encryption on his laptop

# Recommendations for the Average User



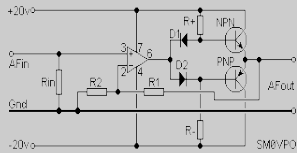

- Don't just click *ok* to any “certificate error” popup unless you really know what your doing!!!
- Check the URL carefully, is it who you think it should be?
- Your level of “trust” at the other end of the connection – Don't deal with unknown web sites

# Recommendations for the Sys Admins and Developers




- Some way to “shore up” the distribution of root certificates
- Someway to easily verify if any of your users have suspicious root certificates
- Someway to actually use those revocation lists
- Education of your users!
- Other???

# Understanding SSL/TLS

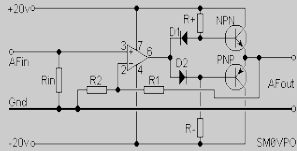



End of first half of talk

Second half, will be a technical “howto”

(If you're a pointy haired boss, now would be a good time to make for the exit.)

# OpenSSL HowTo

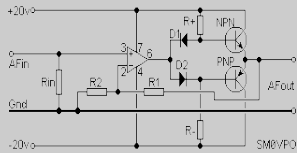


There are two things we would like to cover

- Standard SSL use, have “real” signer sign your certificate – what most people want to do
- “Self signed certificates” – Be your own signing authority

(This will be a “Linux” point of view howto)

# OpenSSL HowTo



A key concept that was left out of the first half:

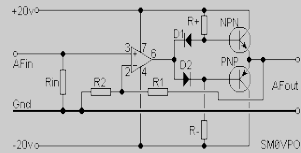
SSL/TLS is usually “one sided”

- Anonymous client wants to connect to a verified server
- Typical web situation

SSL/TLS can be mutual (two sided), just need a certificate for both ends

- There have been suggestions that all mail servers should use and *require* mutual SSL/TLS

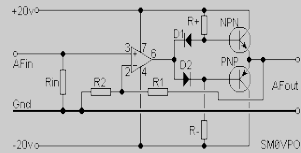
# Standard SSL Use



Have “real” signer sign your certificate – what most people want to do

- Generate your public / private key pair
- Create a “certificate signing request”
- Send it to Verisign
- Receive certificate
- Put files in correct place, and do config files
- Debug

# SSL/TLS “Setup”



Vera

Generates
public key
private key
(e, d, n)

public key *(n)*

Alice
“built into” the
web browser
aka root certificates

Verisign Does This

Bob

Generates
public key
private key
(e, d, n)

public key *(n)*
certificate
signing request
.csr

Vera

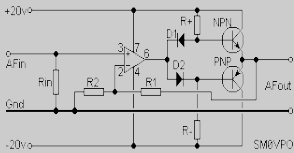
verifies Bob's identity
(calls on the phone?)

“digitally signs”
SSL/TLS certificate
.crt

Bob

We Do This

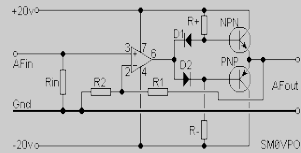
# OpenSSL HowTo




Before we get started, some questions that need answers:

- Are there different types of certificates?
- Where (what directory) do I do this?
- To Passphrase or not to passphrase?

# OpenSSL HowTo



## Are there different types of certificates?

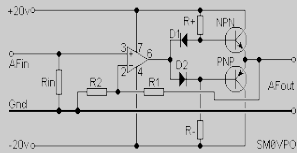
Not really.

Sometime you see instructions for apache mod_ssl, apache strong hold, etc. These differences are for config/setup differences.

I use the same certificate for SMTP, LDAP, web.

Caveat: Windows users, I don't really know.

# OpenSSL HowTo

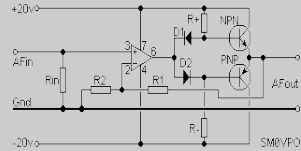



Where (what directory) do I do this?

You often find instructions that so go to such-n-such directory to generate your keys.

It does not matter. I just make a directory someplace and use that. Later on, you move all the files to their proper place.

# OpenSSL HowTo



To Passphrase or not to passphrase?

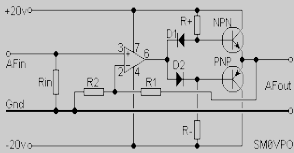
- The short answer is no. Why? Because every time you reboot your web server you have to type in the passphrase.
- Furthermore, you can “remove” the passphrase anytime you want.
- Just handle your keys and certificates wisely.

# Generate Public & Private Keys

Create a directory somewhere, go there and type:

openssl genrsa -out server.key 1024

# Public & Private Keys



Now, lets look inside this file:

openssl rsa -in server.key -text -noout

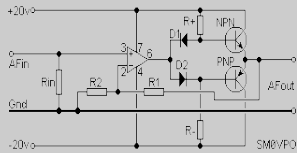
jkh@jkh:~/SSL/test

File Edit View Terminal Tabs Help

```
[jkh@jkh test]$ ls
server.key
[jkh@jkh test]$ openssl rsa -in server.key -text -noout
Private-Key: (1024 bit)
modulus:
    00:b1:62:59:ba:22:3a:f4:8d:4c:04:84:11:6f:54:
    ae:2b:00:54:1a:a9:e1:2e:27:d5:bb:0f:77:ac:fd:
    74:94:55:1c:80:d8:b8:05:06:b5:3d:9d:5e:95:57:
    12:d4:83:5c:ee:3c:27:0e:61:4a:21:d2:86:0e:53:
    07:cd:2f:a0:cf:5a:ef:15:89:a9:b8:ad:be:20:81:
    99:25:68:f3:83:74:f8:f5:7a:05:f6:f3:64:65:17:
    7b:23:0b:ec:71:b4:04:d1:30:1d:eb:6f:f8:bf:1d:
    a0:58:8c:25:51:a8:56:44:17:1d:16:1a:f4:39:ed:
    f7:54:c0:b4:69:d0:8f:85:e7
publicExponent: 65537 (0x10001)
privateExponent:
    3e:97:eb:fd:a8:92:91:35:42:fb:dc:f6:a2:08:89:
    a5:16:ea:96:b6:17:85:c7:fc:bd:93:89:b1:7b:86:
    a5:c9:1a:d3:c4:a5:ba:f0:51:08:3a:fd:e3:48:e4:
    96:24:fc:a7:fc:85:f3:a7:15:17:b6:48:23:c9:dc:
    d8:e4:d3:a3:81:18:8e:60:39:56:67:fc:55:66:df:
    db:8c:72:ef:e7:6e:7e:0b:29:b0:ad:bc:71:46:f7:
    a9:1b:f1:3f:98:d9:ec:cd:18:95:3b:33:6f:29:a0:
    66:4b:50:8f:5b:f1:cf:25:43:94:2b:e0:6e:fc:11:
    18:90:6b:55:01:05:d9:01
prime1:
    00:e7:30:61:a6:dc:29:4f:a8:3f:cf:c7:ef:76:1b:
    ea:58:28:44:70:56:3a:33:e6:93:53:0f:75:f7:0d:
    27:a9:88:7b:85:10:18:93:0d:25:83:0c:a3:c8:2d:
    f0:3c:0a:a4:e3:c7:ff:3c:14:f7:d8:99:09:d4:fe:
    e2:fc:c1:0a:41
prime2:
    00:c4:6b:c0:62:dc:8a:a3:1b:c6:0c:1c:53:5e:64:
    11:e1:41:5d:9e:a5:00:58:b5:06:38:f3:f7:6e:3e:
    ea:49:41:b6:7e:d3:70:1f:71:40:ad:f4:36:a3:b3:
    12:8e:66:49:ba:d7:3b:68:2c:d3:61:48:b8:c2:b5:
    2d:c7:2d:76:27
exponent1:
    7d:69:e1:8f:83:1f:65:f6:ff:a6:d7:ef:2f:43:35:
    7d:5d:5a:c3:16:03:bc:e3:f8:47:b6:bb:67:a9:db:
    3a:3c:af:c5:1f:56:35:6f:8c:49:eb:31:6e:06:7d:
    ac:14:9a:1e:77:00:1e:b7:c1:3b:de:c9:56:a2:6e:
    c0:6e:78:01
exponent2:
    00:a2:3d:f2:92:76:62:99:69:d6:83:a8:5e:00:88:
    9d:f0:36:e2:d2:a8:ae:09:97:b5:45:13:d7:5c:1b:
    e1:bf:92:e4:e0:0b:5c:00:cc:b6:45:39:84:6c:60:
    fc:c1:dc:a1:a8:10:22:44:e1:49:d4:80:92:04:5a:
    4d:2e:c9:51:5f
coefficient:
    0b:7f:02:2b:a8:24:17:41:65:f3:3d:3f:2f:4b:9e:
    5f:03:97:fb:24:5f:3e:c9:96:fc:57:10:69:fe:f4:
    5a:90:69:e1:93:62:c4:3c:9b:e7:c1:8e:1f:ed:e2:
    bf:5b:4c:46:58:12:df:2a:3d:0c:c0:cf:65:04:17:
    fe:00:0a:d6
[jkh@jkh test]$
```

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

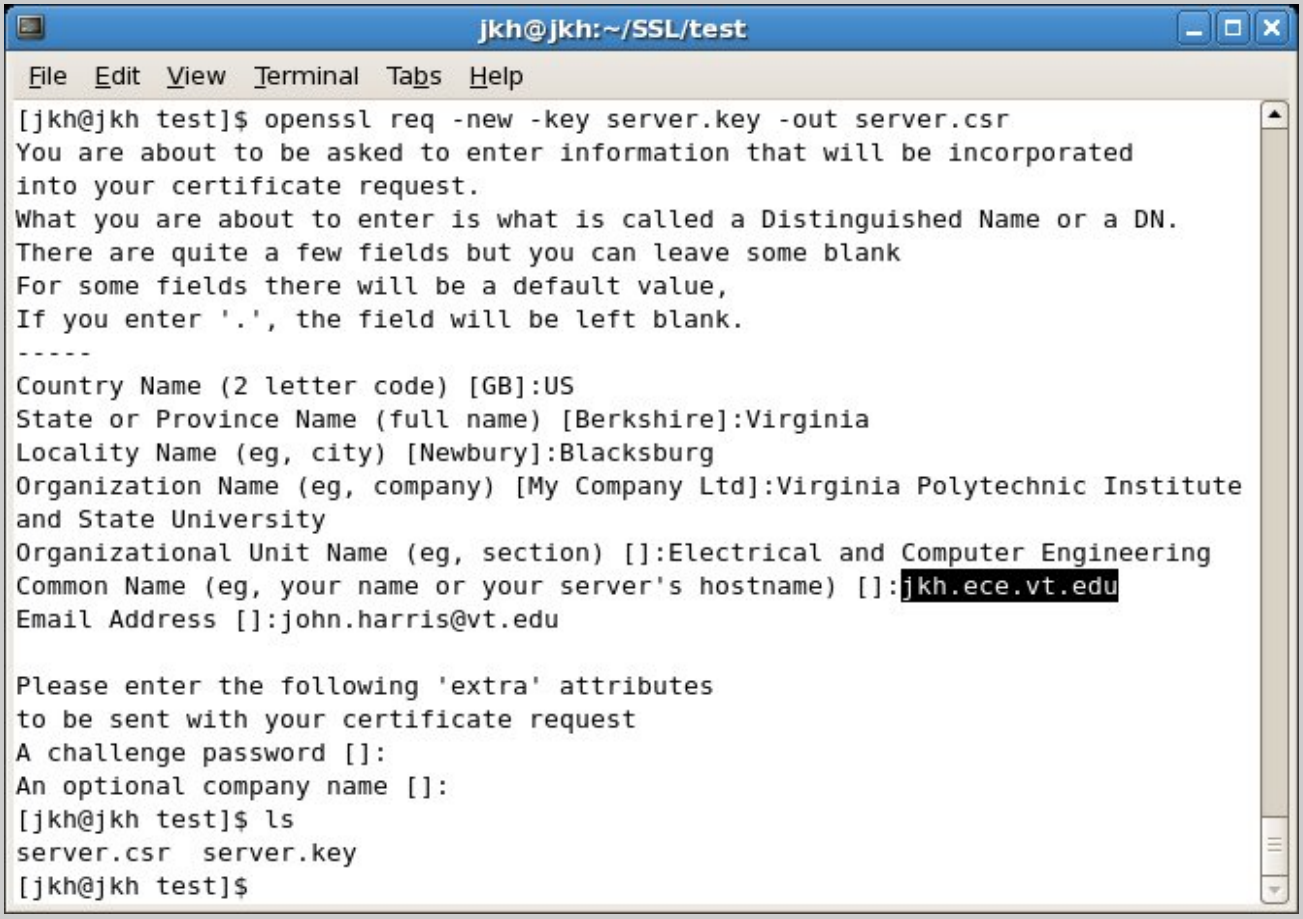


# Create a Certificate Signing Request

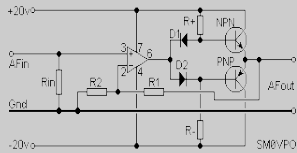
openssl req -new -key server.key -out server.csr

***Asks for your "name"!!***

This MUST be correct.



# Certificate Signing Request




To see what is inside the .csr:

```
openssl req -in server.csr -text -noout
```

# Certificate Signing Request

Has only public key

and “name”

$$c = m^e \bmod n$$

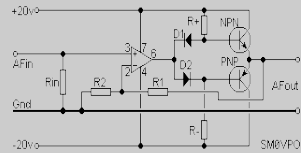
$$m = c^d \bmod n$$

jkh@jkh:~/SSL/test

File Edit View Terminal Tabs Help

```
[jkh@jkh test]$ openssl req -in server.csr -text -noout
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=US, ST=Virginia, L=Blacksburg, O=Virginia Polytechnic Institu
te and State University, OU=Electrical and Computer Engineering, CN=jkh.ece.vt.e
du/emailAddress=john.harris@vt.edu
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:b1:62:59:ba:22:3a:f4:8d:4c:04:84:11:6f:54:
                    ae:2b:00:54:1a:a9:e1:2e:27:d5:bb:0f:77:ac:fd:
                    74:94:55:1c:80:d8:b8:05:06:b5:3d:9d:5e:95:57:
                    12:d4:83:5c:ee:3c:27:0e:61:4a:21:d2:86:0e:53:
                    07:cd:2f:a0:cf:5a:ef:15:89:a9:b8:ad:be:20:81:
                    99:25:68:f3:83:74:f8:f5:7a:05:f6:f3:64:65:17:
                    7b:23:0b:ec:71:b4:04:d1:30:1d:eb:6f:f8:bf:1d:
                    a0:58:8c:25:51:a8:56:44:17:1d:16:1a:f4:39:ed:
                    f7:54:c0:b4:69:d0:8f:85:e7
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha1WithRSAEncryption
        74:38:6d:82:6d:2f:0f:10:07:ca:0b:cb:39:56:a1:5e:54:69:
        94:bb:96:32:f8:09:96:24:01:17:de:b5:85:17:cd:a2:df:36:
        88:ab:a7:55:81:f2:cb:6e:b0:e0:9b:d1:28:74:93:8b:26:52:
        6a:f8:fc:25:8f:e6:db:12:c2:31:5a:0e:ff:d1:67:0c:03:97:
        cd:5a:52:4f:f4:49:dd:00:35:3e:74:4c:6a:0f:cf:0d:90:d6:
        52:9e:dd:32:95:c7:6a:8f:33:93:c6:26:32:9e:1d:5b:99:9f:
        2a:d1:dc:95:8e:b6:34:9e:6d:0f:54:85:75:5e:71:70:83:34:
        09:3c
[jkh@jkh test]$
```

# Certificate Signing Request



The .csr is PEM encoded – text format:

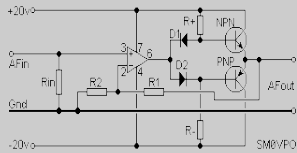
jkh@jkh:~/SSL/test

File Edit View Terminal Tabs Help

```
[jkh@jkh test]$ more server.csr
-----BEGIN CERTIFICATE REQUEST-----
MIICHjCCAYcCAQAwgd0xCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhWaXJnaW5pYTET
MBEGA1UEBxMKQmxhY2tzYnVyZzE8MDoGA1UEChMzVmlyZ2luaWEgUG9seXRlY2hu
aWMgSW5zdGl0dXRlIGFuZCBTdGF0ZSBVbml2ZXJzaXR5MSwwKgYDVQQLEyNFbGVj
dHJpY2FsIGFuZCBDb21wdXRlciBFbmdpbmVlcmluZzEXMBUGA1UEAxMOamtoLmVj
ZS52dC5lZHUxITAfBgkqhkiG9w0BCQEWEmpvaG4uaGFycmlzQHZ0LmVkdTCBnzAN
BgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAsWJZuiI69I1MBIQRb1SuKwBUGqnhLifV
uw93rP10lFUcgNi4BQa1PZ1elVcS1INc7jwnDmFKIdKGDlMHzS+gz1rvFYmpuK2+
IIGZJWjzg3T49XoF9vNkZRd7IwvscbQE0TAd62/4vx2gWIwlUahWRBcdFhr0Oe33
VMC0adCPhecCAwEAAaAAMA0GCSqGSIb3DQEBBQUAA4GBAHQ4bYJtLw8QB8oLyzlW
oV5UaZS7ljL4CZYkARfetYUXzaLfNoirp1WB8stusOCb0Sh0k4smUmr4/CWP5tsS
wjFaDv/RZwwDl81aUk/0Sd0ANT50TGoPzw2Q1lKe3TKVx2qPM5PGJjKeHVuZnyrR
3JWOtjSebQ9UhXVecXCDNAk8
-----END CERTIFICATE REQUEST-----
[jkh@jkh test]$
```

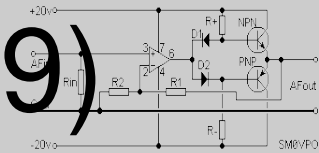
It is typical that you “cut-n-paste” this text into a web page to submit your .csr to the signing authority.

# Send Your CSR




- You send your money and your .csr to Verisign
- Verisign somehow verify that “you are you”
- Verisign will sign and send you your X.509 certificate!

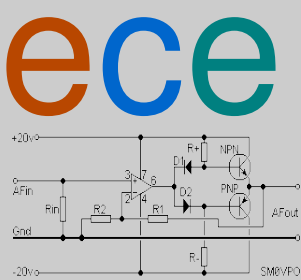
# Receive Your Certificate (X.509)



Verisign sends you your X.509 certificate!

And for the obligatory look inside:

```
openssl x509 -in server.crt -noout -text
```



jkh@jkh:~/SSL/test

File Edit View Terminal Tabs Help

```
[jkh@jkh test]$ openssl x509 -in server.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 38514 (0x9672)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=ES, ST=Barcelona, L=Barcelona, O=IPS Certification Authority s.l., O=general@ipsca.com C.I.F.  B-B62
210695, OU=ipsCA CLASEA1 Certification Authority, CN=ipsCA CLASEA1 Certification Authority/emailAddress=general@ipsca.
com
        Validity
            Not Before: Aug  4 22:32:32 2006 GMT
            Not After : Aug  3 22:32:32 2008 GMT
        Subject: C=US, ST=Virginia, L=Blacksburg, O=Virginia Polytechnic Institute and State University, OU=The Bradle
y Department of Electrical and Computer Engineering, CN=jkh.ece.vt.edu/emailAddress=john.harris@vt.edu
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:a0:cc:50:28:a5:f1:70:0d:9d:6f:02:94:9a:0a:
                    a8:79:ba:ff:80:9f:51:ba:d7:61:85:ba:fe:43:60:
                    2f:80:a1:82:8c:f2:dd:ac:d6:ce:0a:94:05:0e:0f:
                    80:08:67:5a:78:aa:ff:35:2b:6d:2a:c2:13:bd:50:
                    b1:78:b8:11:af:02:38:7c:9e:7a:3c:fa:c9:a1:43:
                    e2:db:5f:2d:ef:d7:c6:51:9a:ac:fd:82:90:82:ff:
                    2a:19:46:4b:c2:42:be:6f:31:74:54:98:28:74:be:
                    26:48:4f:20:1a:89:38:38:45:f4:b5:99:43:21:34:
                    06:0d:7c:1d:9e:c3:fa:bd:07
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Cert Type:
                SSL Server
            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Key Encipherment, Data Encipherment, Key Agreement
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Subject Alternative Name:
                email:john.harris@vt.edu
            X509v3 Issuer Alternative Name:
                email:general@ipsca.com
            Netscape Comment:
                Organization Information NOT VALIDATED. CLASEA1 Server Certificate issued by https://www.ipsca.com/
            Netscape Base Url:
                https://www.ipsca.com/ipsca2002/
            Netscape CA Revocation Url:
                https://www.ipsca.com/ipsca2002/ipsca2002CLASEA1.crl
            Netscape Revocation Url:
                https://www.ipsca.com/ipsca2002/revocationCLASEA1.html?

    Signature Algorithm: sha1WithRSAEncryption
        6d:e5:77:f4:45:d0:37:b4:70:f4:9c:22:cc:16:9b:28:39:8f:
        d1:78:c3:0b:f8:12:53:88:7d:f4:ea:75:29:d6:b2:14:15:28:
        ce:a7:3a:4e:d0:e6:12:2b:43:63:c2:95:c7:ef:06:a7:09:a6:
        6d:95:17:fd:ca:66:6a:c4:6a:42:62:4d:9a:af:8b:c3:87:7b:
        db:bd:60:30:da:7f:72:a1:07:cd:a3:d6:8d:ac:40:53:91:70:
        2a:cf:12:20:b6:26:23:02:d3:78:06:4e:eb:c3:2b:52:4f:8d:
        bf:33:f6:e5:ab:3b:46:16:dd:df:b2:c9:6e:65:aa:fb:8d:6e:
        6a:cf
```

# Configuring Apache

O.k., so you got your shiny new certificate, lets see it go.

Apache setup:

vi /etc/httpd/conf.d/ssl.conf

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

jkh@monitor:~

File Edit View Terminal Tabs Help

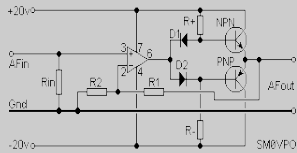
```
#   Server Certificate:
# Point SSLCertificateFile at a PEM encoded certificate.  If
# the certificate is encrypted, then you will be prompted for a
# pass phrase.  Note that a kill -HUP will prompt again.  A new
# certificate can be generated using the genkey(1) command.
#jkh SSLCertificateFile /etc/pki/tls/certs/localhost.crt
SSLCertificateFile /etc/pki/tls/certs/server.crt

#   Server Private Key:
#   If the key is not combined with the certificate, use this
#   directive to point at the key file.  Keep in mind that if
#   you've both a RSA and a DSA private key you can configure
#   both in parallel (to also allow the use of DSA ciphers, etc.)
#jkh SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
SSLCertificateKeyFile /etc/pki/tls/private/server.key

#   Server Certificate Chain:
#   Point SSLCertificateChainFile at a file containing the
#   concatenation of PEM encoded CA certificates which form the
#   certificate chain for the server certificate. Alternatively
#   the referenced file can be the same as SSLCertificateFile
#   when the CA certificates are directly appended to the server
#   certificate for convinience.
#SSLCertificateChainFile /etc/pki/tls/certs/server-chain.crt
#jkh
SSLCertificateChainFile /etc/pki/tls/certs/IPS-IPSCABUNDLE.crt
```

133,0-1 51%

# Testing Apache

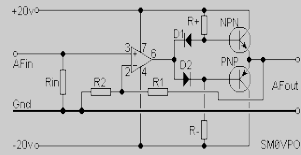



Restart you web server, and try

https://<machine.domain>

If all goes well, you see the little “lock” appear with no “certificate error” popups.

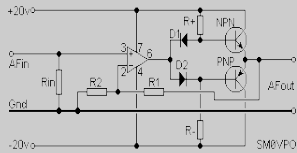
# Debugging



We all know that in the real world, everything works perfectly, the first time, every time... But should that exceedingly rare event actually occur where things aren't working...

- Take a long look at the conf file, make *sure* the files are where you say they are
- Look at the log files (note ssl has separate log files ssl_access_log, ssl_error_log)

# Debugging



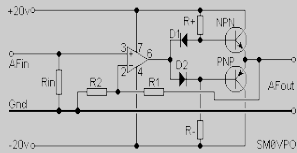
And lastly, try

openssl s_client -connect <machine>:<port> -debug -state

Keep in mind that the port number changes for SSL. For web servers, it is not port 80, but port 443.

What this does is gives you **a lot of output**, but most importantly, a **bidirectional connection** to your web server through SSL.

# Debugging



openssl s_client -connect filebox.ece.vt.edu:443 -debug -state

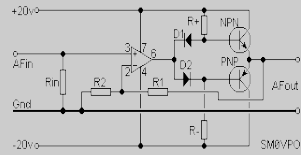
So, you must be able to “speak HTTP”. Type the above and admire the voluminous output. It will wait for input. There is no prompt, just type:

GET / HTTP1.0<enter>

<enter>

More voluminous output, but you should see some HTML looking stuff.

# Standard Use



Thats it!  Its that simple, only two commands:

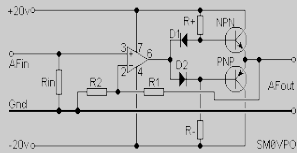
```
openssl genrsa -out server.key 1024

openssl req -new -in server.key -out server.csr
```

Note(s) to self:

- Protect your keys, especially the private key. Change file/directory permissions.
- Make a backup of your keys!  If you lose your certificate, they are *not* suppose to issue you another one (but they do).

# Self Signed Certificates

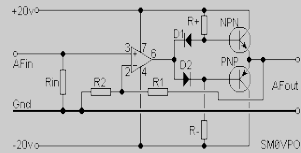


"Self signed certificates"

- Be your own signing authority

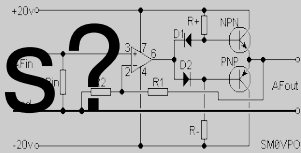
- What do you mean "Self signed certificate"?
- Why would you want to do this?
- What happens when you use your self signed certificate?
- And of course, how do you do this?

# What Do You Mean by "Self Signed Certificate"?



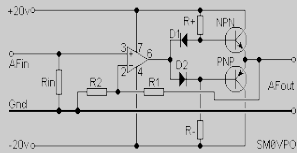

- The term "Self signed certificate" is incorrect, the proper phrase is "Being your own Certificate Authority", or CA
- You have the "root key"
- And you can "sign" other certificates

# Why Would You Want to do This?



- Cost – free.  Any linux box that has openssl installed (all) has everything you need
- Provides encryption, but no “verification”
- Closed systems.  Sometimes you want to keep others out.  Ex. LDAP /w “require ssl”
- Keep “Big Brother” from snooping!

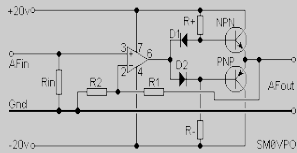
# Self Signed Certificates



What happens when you use your self signed certificate?

- Some applications won't proceed, e.g., LDAP requre ssl
- Well, you get the “certificate error” popup
- You can “install” the public key
- Or you can “install” the root certificate

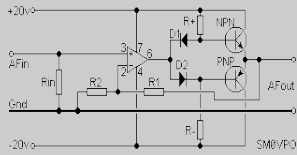
# Self Signed Certificates




There is a lot of confusion on the net about this. If you google “self signed certificate” will get lots of hits. They'll give you some commands to type, but almost all of the instructions lack explanation (that I can understand).

... But all these web pages give different instructions!

# Self Signed Certificates

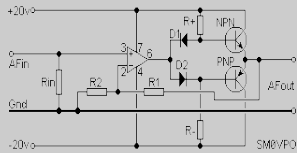


Before we get started, one important point:

- Private keys do *not* have the “name” in them
- Public keys (AKA X.509) Certificates have the “name”

When you create an X.509 certificate, you will be asked for the “name”. The “name” is actually more than just the DNS, it is also your “location”, “affiliation” and perhaps a “responsible party”. (Other?)

# Self Signed Certificates

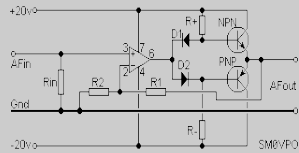


O.k., enough adieu, lets get started.

We need:

- Two sets of keys:
  - The CA's keys (Certificate Authority)
  - The certificate you are going to “sign”

# SSL/TLS “Setup”



CA

Generates
public key
private key
(e, d, n)

public key *(n)*

CA
Distributed
Somehow

Server

Generates
public key
private key
(e, d, n)

public key *(n)*
certificate
signing request
.csr

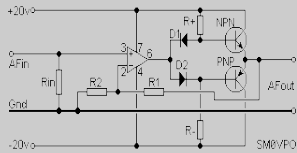
CA

Identity
Verification?

“digitally signs”
SSL/TLS certificate
.crt

Server

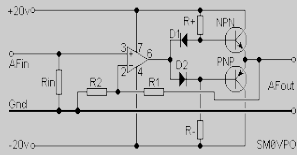
# Self Signed Certificates



## Step 1: Create the CA's key pair

openssl genrsa -out CA.key 1024

# Self Signed Certificates



Step 2: The CA needs its own “certificate”

Why?

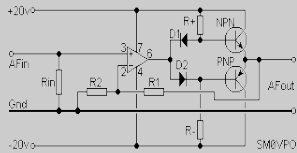
–> This is the “widely published” “root certificate”

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

–> Ask for “name”

**This “name” is the CA's name!!! Not a valid DNS name.**

# Self Signed Certificates



Create CA's root

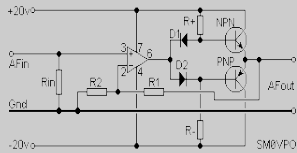
Create root cert (asks for "Name")

jkh@jkh:~/SSL/test4

File Edit View Terminal Tabs Help

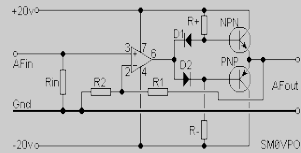
```
[jkh@jkh test4]$ openssl genrsa -out CA.key 1024
Generating RSA private key, 1024 bit long modulus
.........++++++
............++++++
unable to write 'random state'
e is 65537 (0x10001)
[jkh@jkh test4]$ ls
CA.key
[jkh@jkh test4]$ openssl req -new -x509 -days 3560 -key CA.key -out CA.crt
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [GB]:US
State or Province Name (full name) [Berkshire]:Virginia
Locality Name (eg, city) [Newbury]:Blacksburg
Organization Name (eg, company) [My Company Ltd]:Virginia Polytechnic Institute and State University
Organizational Unit Name (eg, section) []:Electrical and Computer Engineering
Common Name (eg, your name or your server's hostname) []:Virgina Tech, ECE Certificate Authority
Email Address []:john.harris@vt.edu
[jkh@jkh test4]$ ls
CA.crt  CA.key
[jkh@jkh test4]$
```

# Self Signed Certificates




Note: For the pedantic minded people, the “root certificate” is the only “self signed” certificate.

# Self Signed Certificates



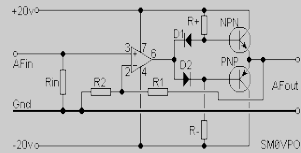
Step 3: Create the private key for the server.

(The “server” in this case, is you web server.)

Just like any other public/private key generation:

```
openssl genrsa -out server.key 1024
```

# Self Signed Certificates



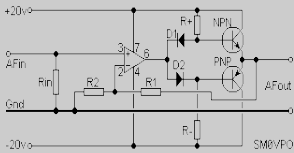
Step 4: Create a “Certificate Signing Request”

openssl req -new -key server.key -out server.csr

This will ask you for the “name” of the machine.

In this case you *must* use the DNS name!!!

# Self Signed Certificates



Step 5: “Sign” the certificate.

openssl x509 -req

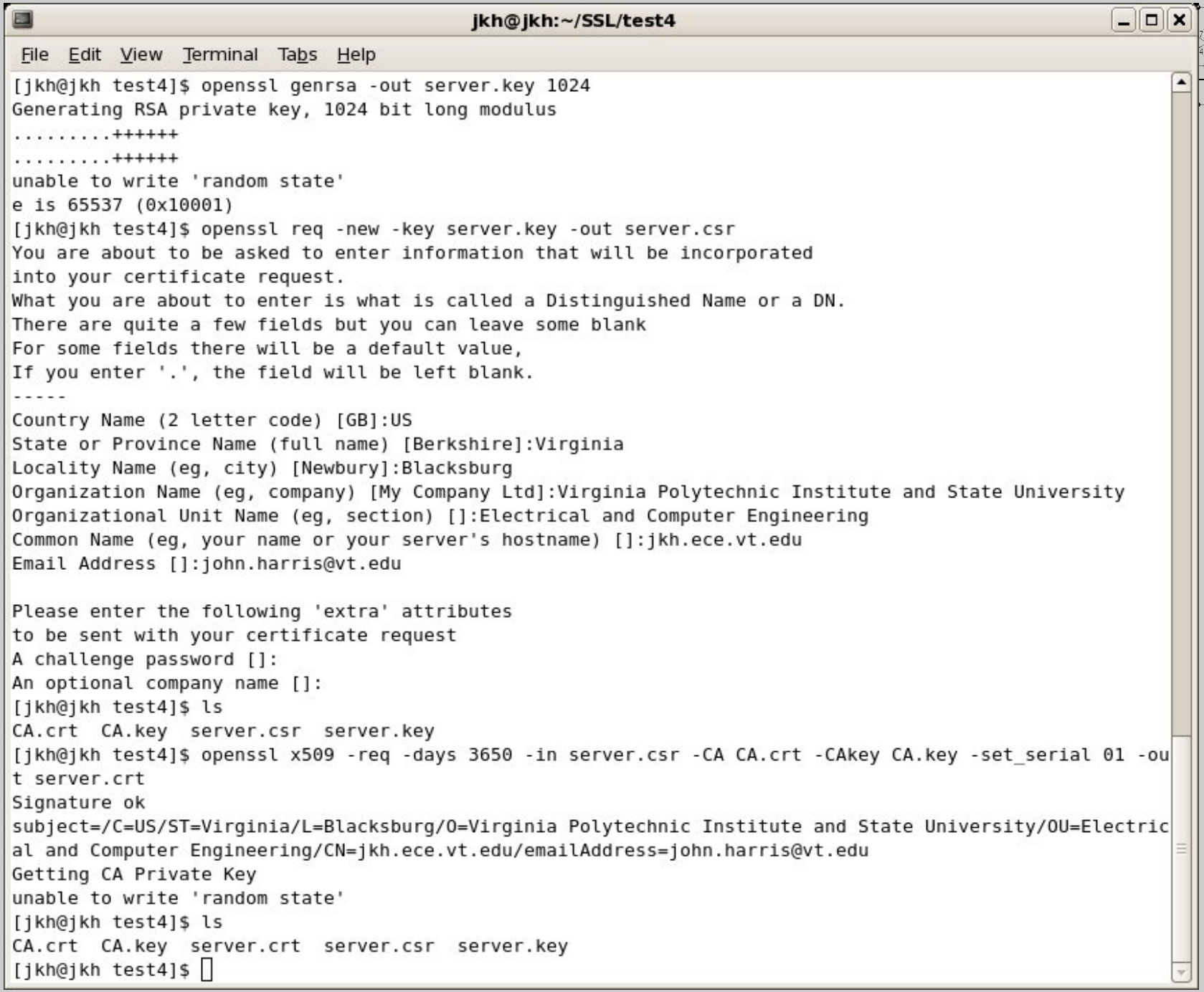
-days 3650

-CA CA.crt -CAkey CA.key

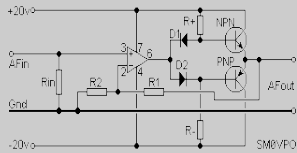
-set_serial 01

-in server.csr

-out server.crt

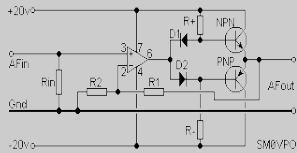

jkh@jkh:~/SSL/test4
File Edit View Terminal Tabs Help
[jkh@jkh test4]$ openssl genrsa -out server.key 1024
Generating RSA private key, 1024 bit long modulus
.........++++++
.........++++++
unable to write 'random state'
e is 65537 (0x10001)
[jkh@jkh test4]$ openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [GB]:US
State or Province Name (full name) [Berkshire]:Virginia
Locality Name (eg, city) [Newbury]:Blacksburg
Organization Name (eg, company) [My Company Ltd]:Virginia Polytechnic Institute and State University
Organizational Unit Name (eg, section) []:Electrical and Computer Engineering
Common Name (eg, your name or your server's hostname) []:jkh.ece.vt.edu
Email Address []:john.harris@vt.edu
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
[jkh@jkh test4]$ ls
CA.crt CA.key server.csr server.key
[jkh@jkh test4]$ openssl x509 -req -days 3650 -in server.csr -CA CA.crt -CAkey CA.key -set_serial 01 -ou
t server.crt
Signature ok
subject=/C=US/ST=Virginia/L=Blacksburg/O=Virginia Polytechnic Institute and State University/OU=Electric
al and Computer Engineering/CN=jkh.ece.vt.edu/emailAddress=john.harris@vt.edu
Getting CA Private Key
unable to write 'random state'
[jkh@jkh test4]$ ls
CA.crt CA.key server.crt server.csr server.key
[jkh@jkh test4]$

# Self Signed Certificates




- You can “look at” the certificates the same as above.
- You install the certificates the same as above.
- And of course, debug as before.

# Self Signed Certificates

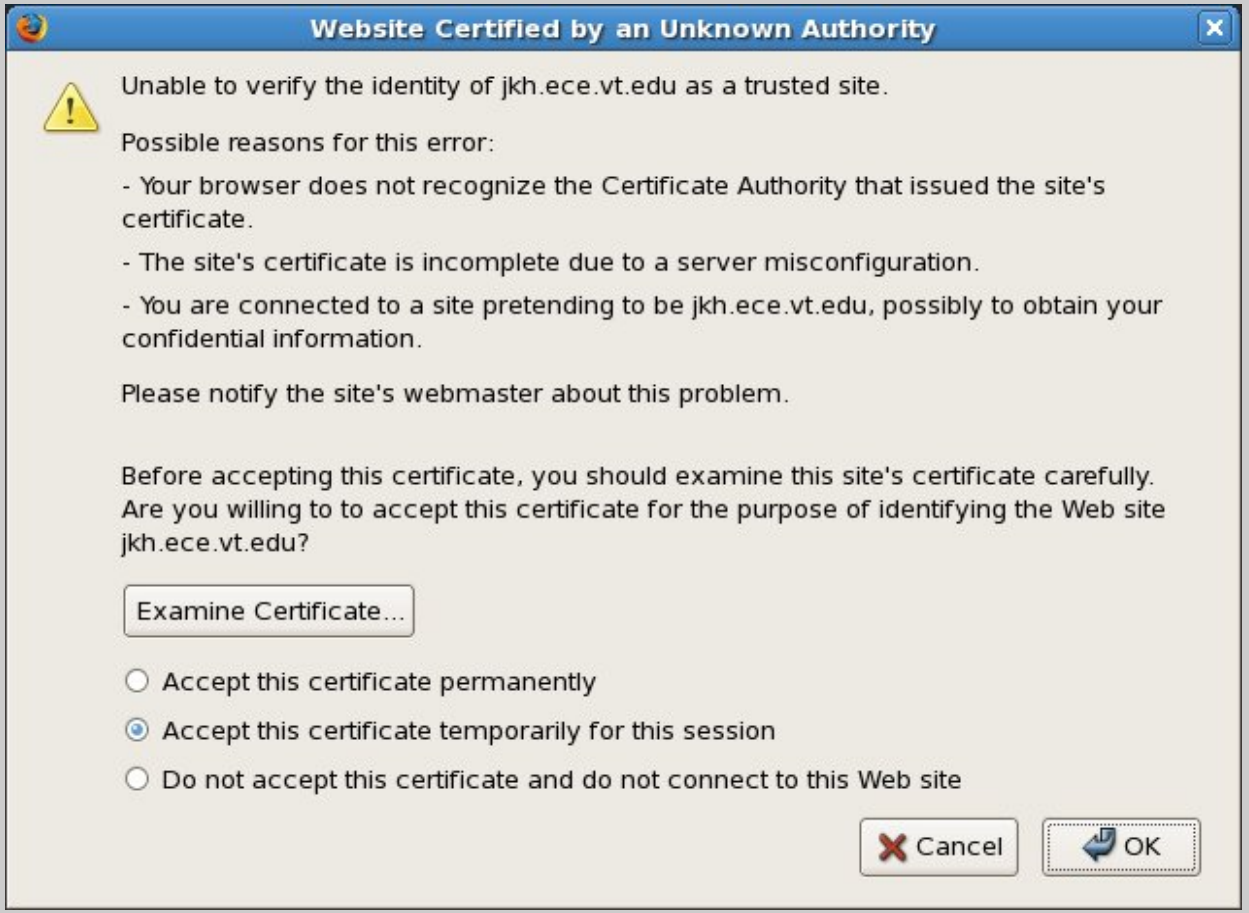


O.k., so you've created your own self signed certificate, now what?

- Always get the certificate error popup – just click OK
- Accept the certificate forever – no more popup.
- Install the “root certificate” – no popups for any certificate signed by this CA.

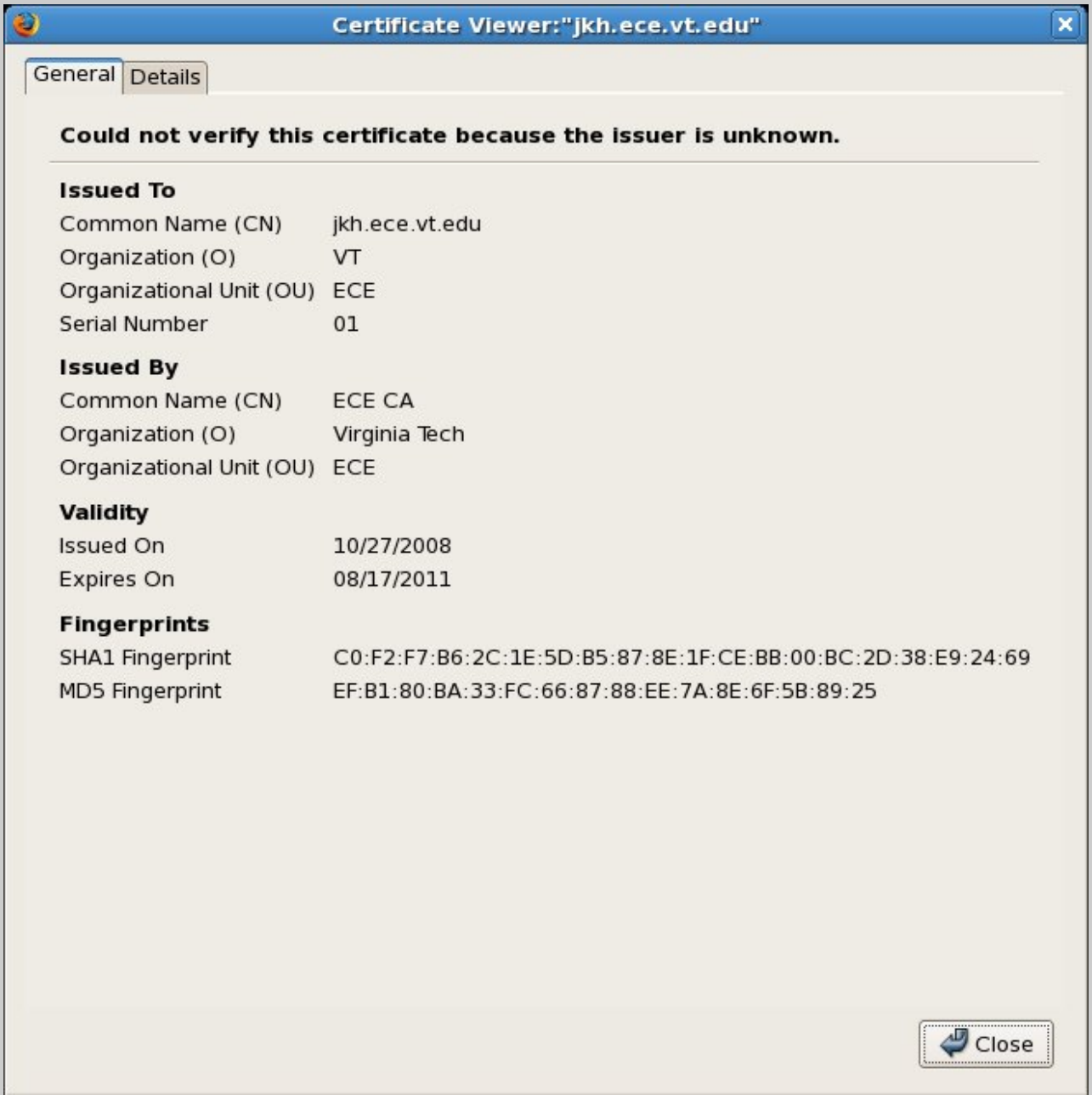
# Certificate Error Popup

Always get the certificate error popup – just click OK



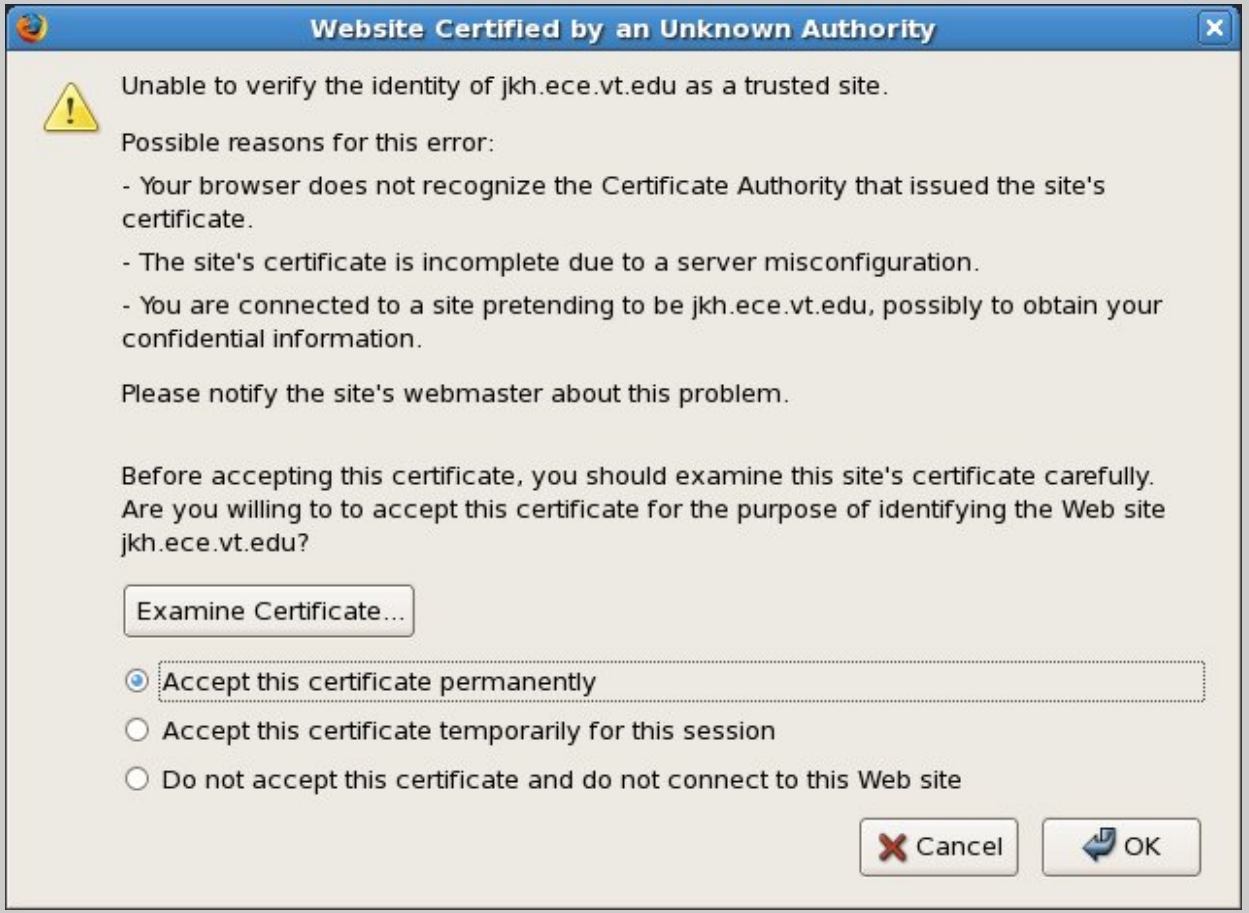
# Examine the Certificate

Sure enough, we see our certificate.



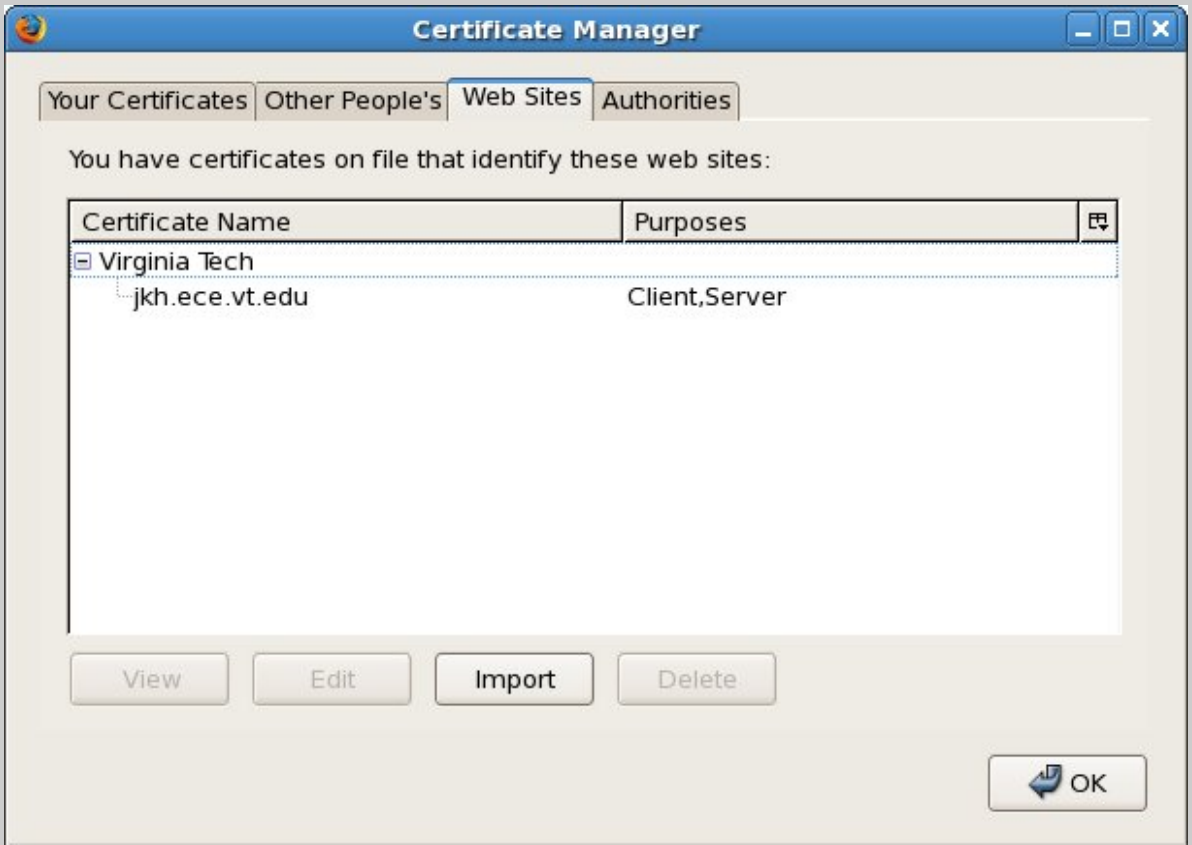
# Accept the Certificate

You can “bless” this certificate and get rid of the popup.

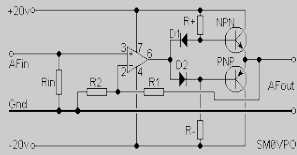


# Accept the Certificate

Let's take a look at what “accepting the certificate” did. Click: Edit, Preferences, tab Advanced, View Certificates. Now click: tab Web Sites.



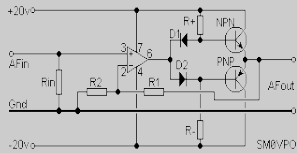
# Install “Root Certificate”



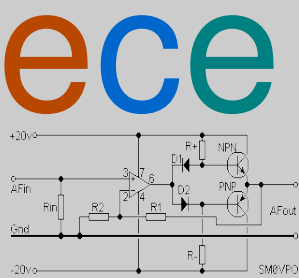
Suppose you have a number of places that you might use a certificate that is signed by your CA. Rather than accepting the certificate for a given web site, you could install the CA's root certificate.

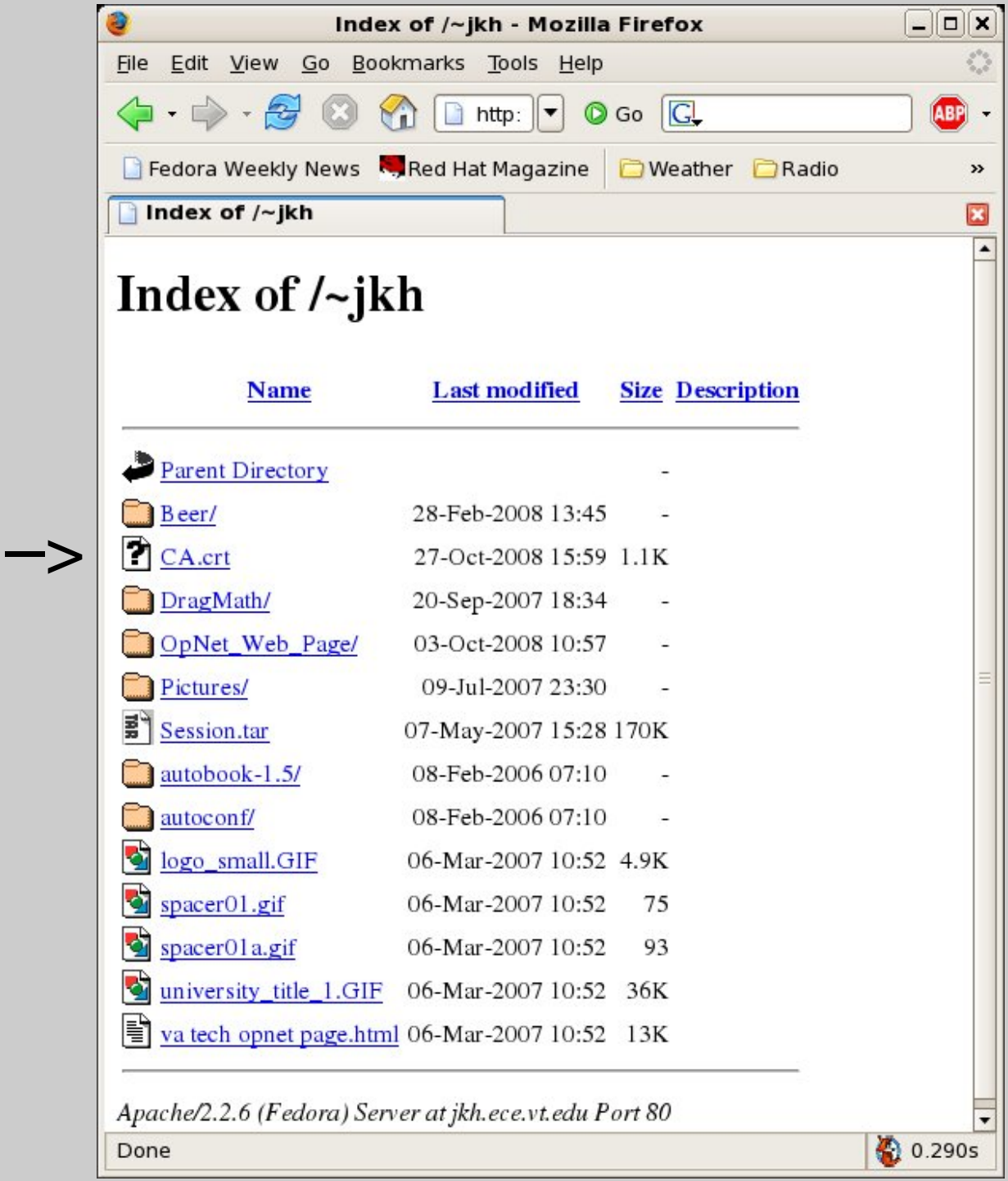
This has the affect of saying: “I trust all certificates signed by this CA.”

# Install “Root Certificate”




A quick way of doing this is to put the file CA.crt on a web page. Then, from your browser, just click on the file, the browser will know what a .crt file is.

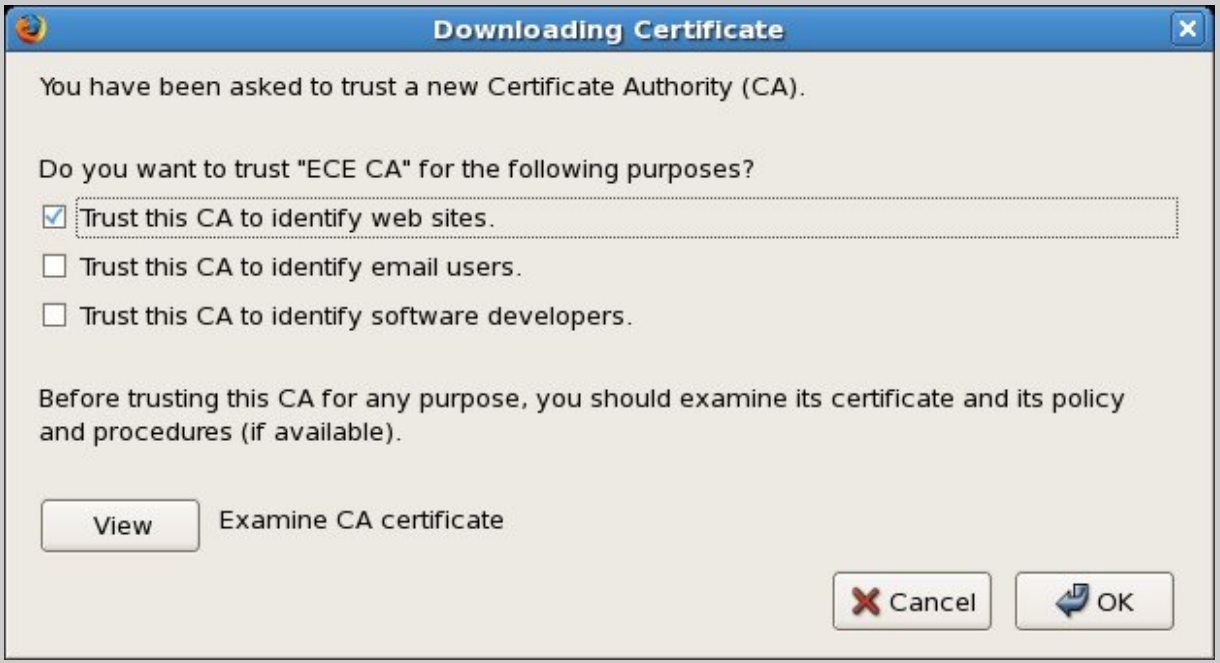



Index of /~jkh - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http:
Go
Fedora Weekly News
Red Hat Magazine
Weather
Radio
Index of /~jkh
Index of /~jkh
Name
Last modified
Size
Description
Parent Directory -
Beer/ 28-Feb-2008 13:45 -
CA.crt 27-Oct-2008 15:59 1.1K
DragMath/ 20-Sep-2007 18:34 -
OpNet_Web_Page/ 03-Oct-2008 10:57 -
Pictures/ 09-Jul-2007 23:30 -
Session.tar 07-May-2007 15:28 170K
autobook-1.5/ 08-Feb-2006 07:10 -
autoconf/ 08-Feb-2006 07:10 -
logo_small.GIF 06-Mar-2007 10:52 4.9K
spacer01.gif 06-Mar-2007 10:52 75
spacer01a.gif 06-Mar-2007 10:52 93
university_title_1.GIF 06-Mar-2007 10:52 36K
va tech opnet page.html 06-Mar-2007 10:52 13K
Apache/2.2.6 (Fedora) Server at jkh.ece.vt.edu Port 80
Done
0.290s

# Install “Root Certificate”

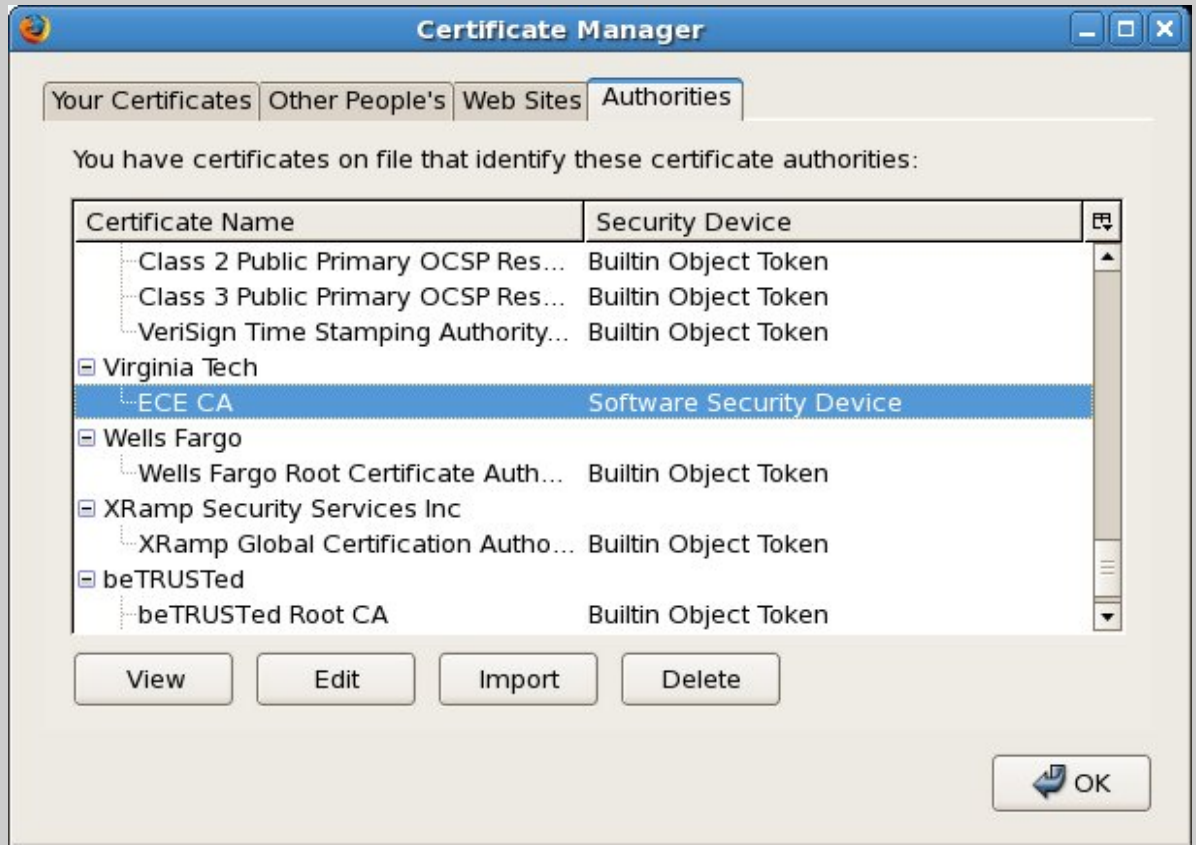
You must say “for what” you will trust this root certificate:

1. Web
2. email
3. Java applet

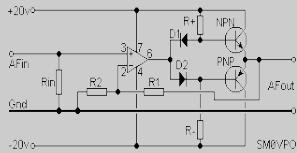


# Install “Root Certificate”

Let's take a look at what “installing the root certificate” did. Click: Edit, Preferences, tab Advanced, View Certificates. Now click: tab Authorities.



# Self Signed Summery



- Generate CA (.key and .crt):

  openssl genrsa -out CA.key 1024

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

- Generate server (.key and .csr):

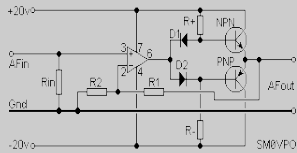
  openssl genrsa -out server.key 1024

  openssl req -new -key server.key -out server.csr

- “Sign” the certificate (.crt):

  openssl x509 -req -days 3650 -set_serial 01
  -CA CA.crt -CAkey CA.key
  -in server.csr -out server.crt

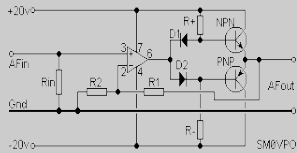
# Understanding SSL/TLS




Wrapping up, last comments

- We talked about SSL, what about TLS?
- Something about DSA
- Protocols that support SSL/TLS
- Things not covered in this talk

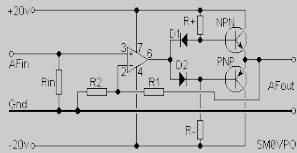
# What About TLS?




Trying to SSL existing applications required a second “TCP port”. Eg. www: 80 (non-SSL) & 443 (SSL port), SMTP: 25 & 465, IMAP: 143 & 585. “Port proliferation”.

TLS is basically SSL, but you start the connection unencrypted, and ask: “Can you do TLS?” Then enter into the SSL protocol.

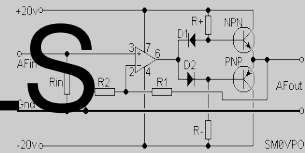
# Something About DSA




In addition to the RSA algorithm, there are several other algorithms, but to the best of this authors research, the only one widely implemented is RSA.

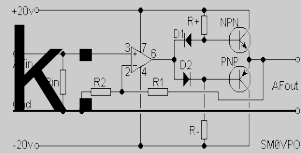
In short, someday we might have something other than RSA, but for right now, just use RSA.

# Protocols That Support SSL/TLS



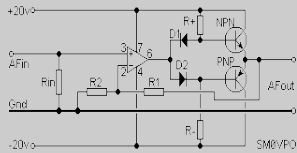
- www
- nntp
- imap
- pop
- SMTP
- ftps
- LDAP
- CORBA
- Oracle
- MS Global Catalog

# Things Not Covered in This Talk:



- Being a “real” CA – more complicated than you'd ever expect; revocation list, serial numbers, etc.
- Mutual trust: certificate at both ends of the connection.  Eg. LDAP, SMTP?
- Programming using SSL/TLS
- Understanding the rhyme and reason behind openssl's command line...
- How to do this using Windows

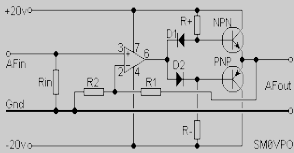
# Understanding SSL/TLS



References:

- www.openssl.org
- www.openldap.org
- www.courier-mta.org
- http://www.madboa.com/geek/openssl/
  (Recommended!)

# Understanding SSL/TLS




The End (Finally!)