# Connecting to the CotSS IRC Service

October 16, 2017

## 1 Introduction

IRC (Internet Relay Chat) is a protocol that allows users to communicate over the Internet; specifically, it allows a *client* (you) to communicate with a *server* (CotSS, located at frostsnow.net). By itself, IRC sends information in *plaintext*, meaning that anyone can view and even modify said information. In order to prevent this, the IRC protocol can be encapsulated by the TLS (Transport Layer Security) protocol, which turns the plaintext into *ciphertext* which can be neither read (giving *confidentiality*) nor modified (giving *integrity*). In addition, TLS provides *authenticity*, which verifies the server to the client (letting the client know that it is indeed communicating with CotSS and not an imposter) and optionally verifies the client to the server (letting the server know that it is talking with a specific client). Lastly, TLS provides *authorization*, which verifies that the client is allowed to connect to the server. This guide will show you how to use TLS to connect to the CotSS IRC server.

TLS uses a type of cryptography often referred to as "Public-Private Key" Cryptography (formally, *Asymmetric* Cryptography), because each party has a pair of keys. A *public* key is shared between parties while a *private* key is kept secret. In TLS, the public key is called a *certificate* (more specifically, an *X.509* certificate). Therefore, in order to communicate with CotSS, you will need three things: 1) the server's certificate, in order for you to verify that the computer you are talking to is in fact CotSS; 2) your certificate, in order for the server to verify that it is talking to you; and 3) your private key, in order to verify to the server that you are the proper owner of the certificate. Note that the server will use its private key in order to verify to you that it is the proper owner of its certificate.

## 2 Connecting

Connecting will involve two steps: 1) verifying the server to yourself, and 2) verifying yourself to the server.

## 2.1 Server Verification

Verifying the server to yourself will also consist of two steps: 1) obtaining a trusted copy of the server's certificate, and 2) configuring your client to use said certificate.

Ideally, you would obtain a trusted certificate by either obtaining and verifying its GPG (GNU Privacy Guard) signature with a previously-verified public key or by physically obtaining a trusted copy from me. Alas, the former is outside the scope of this document and the latter is rather cumbersome for both of us, so in most cases you'll simply connect to `CotSS` over the Internet, obtain the certificate that is presented to you, and use that, but do keep in mind that if your first connection was to an imposter, all subsequent connections will be vulnerable to the same imposter, thus you should make an attempt to verify the certificate if you are able to do so.

In order to get the certificate, run the commands:

```
$ mkdir ~/irc
$ openssl s_client -connect frostsnow.net:6697 2> \
    /dev/null | sed -n '/BEGIN CERT/,/END CERT/p' \
    > ~/irc/cotss.pem
```

The `$` denotes the beginning of a command. The first command create a directory named `irc` in your home directory. The second command should retrieve the certificate from `CotSS`. The `openssl` command itself consists of a series of subcommands; the `s_client` command acts as a client to a server, `-connect frostsnow.net:6697` tells the client to connect to `frostsnow.net` on port `6697`, which is where `CotSS` is located. `2> /dev/null` redirects extra crap into a void from which it does not return, `| sed -n '/BEGIN CERT/,/END CERT/p'` filters the output of the command to only include the lines encoding the certificate, and `> ~/irc/cotss.pem` takes the filtered output and places it in the file `cotss.pem` within the `irc` directory. The backslash denotes that the command continues onto the next line; you may omit them and place the entire command on a single line.

You'll want to verify that you have actually recieved a certificate (for example, there may be a network error). You can run:

```
$ cat ~/irc/cotss.pem
```

. . . which should show a text file beginning with `-----BEGIN CERTIFICATE-----`, followed by a bunch of text, and ending with `-----END CERTIFICATE-----`. In order to actually view the contents of the certificate, run the command:

```
$ openssl x509 -in ~/irc/cotss.pem -text -noout
```

. . . which should then show various information about the certificate. For extra fun, perform the same steps except use `google.com:8080` instead of `frostsnow.net:6697`.

Now that you have a copy of the server's certificate, you need to tell your IRC client to use it as a *Certificate Authority* (CA); Certificate Authorities determine which certificates are valid (roughly speaking). This step of configuring your

IRC client is client-specific, in other words, different clients will have different methods of configuration. This guide will use the irssi client; if you are using another client, you will need to look at its documentation in order to figure out how to perform this step. In order to connect using the certificate you've downloaded, launch the irssi program and connect with the command:

```
/connect -ssl_verify -ssl_cafile ~/irc/cotss.pem \
    frostsnow.net 6697
```

This will instruct irssi to connect to frostsnow.net, port 6697 (again, where CotSS is located) using the file ~/irc/cotss.pem as a Certificate Authority, after which you will propmpty be disconnected from the server as you have not yet provided a valid client certificate, which is the topic of the next section. Go ahead and type /rmreconns to prevent irssi from trying in vain to reconnect to the server, then type /exit to quit irssi.

## 2.2 Client Verification

Creating a client certificate is rather more involved than getting the server's certificate, in large part because you must both store a *secret* key and you must also have your certificate *signed* by the appropriate certificate authority. In total, there are five steps: 1) Generating a private key, 2) Generating a certificate signing request, 3) Sending the certificate signing request, 4) Recieving the signed certificate, and 5) Configuring your client to use the signed certificate and private key.

Generating the key is simple enough: the one trick is to make sure that no other user on the system has read permission on the key, as that would defeat the point of the key being *secret* (note that even if you are the only real person that logs into your system, most GNU\Linux systems are configured with multiple users for security purposes). In order to generate the secret key, run the following three commands:

```
$ umask 0077
$ openssl genrsa -out ~/irc/key.pem 4096
$ umask 0022
```

The first command, umask 0077 prevents any file created from being accessible in any way whatsoever to other users (besides the system administrator, of course). The second command generates an *RSA* key of length 4096 bits. The third command (optional) prevents any subsequent file created from being writable by other users (this command restores the umask defaults; the key will remain inaccessible). As you may have guessed, this command will create the secret key for you in the location ~/irc/key.pem; it is important to keep this file to yourself and never reveal it to anyone, as part of the magic of asymmetric cryptography relies on the fact that only you have access to this file (not even the server knows your secret key).

Second, use the secret key to generate a *certificate signing request* (CSR) that will be sent to the appropriate certificate authority (you should know who this is). The CSR includes a "Common Name" (CN) field, in which you should place your *Erisian Holy Name* (think: "nickname"). The command itself is:

```
$ openssl req -key ~/irc/key.pem -sha512 -new \
    -subj '/CN=yournickhere/' -out ~/irc/csr.pem
```

This command uses the `req` subcommand in order to generate the certificate signing request; `-key ~/irc/key.pem` uses the previously-generated key in generating the request, `-sha512` uses the SHA-512 hash in the requet, `-new` specifies that we are generating a new request, `-subj '/CN=yournickhere/'` allows you to specify your Common Name and `yournickhere` should (obviously) be replaced with your chosen nickname[1], and `-out ~/irc/csr.pem` specifies where the generated request is output.

Thirdly, now that you have your certificate signing request, you must get it signed (obviously). Ideally, this should be done in the same way as for validating the server certificate: either using GPG signatures with a previously-verified public key or by physically exchanging the files in person. In practice, you'll probably just e-mail the `~/irc/csr.pem` file to the person who invited you like a lazy fuck. You ought to know how to do that.

Fourthly, you will obtain the signed certificate using whatever method you've employed. It will probably be called `yournick.pem` or `cert.pem`. There isn't really anything that needs to be done here besides placing the signed certificate at `~/irc/yournick.pem`, but it's worth viewing the contents of the certificate with:

```
$ openssl x509 -in ~/irc/yournick.pem -text -noout
```

Of interest is the "Issuer" (the person who signed the certificate) and the "Subject" (you).

Lastly, you must configure your client to use both the signed certificate and your secret key. As with setting up the server certificate, these instructions are specific to the `irssi` client and if you are using a different client then you will need to adapt the instructions to your client. Also, the following command will assume that you have successfully obtained and verified the server's certificate as per the preceding section. Now, in order to use your newly-obtained client certificate, start `irssi` and run the following command (without backslashes and all on one line):

```
/connect -ssl_verify -ssl_cafile ~/irc/cotss.pem \
    -ssl_cert ~/irc/yournick.pem \
    -ssl_pkey ~/irc/key.pem \
    frostsnow.net 6697
```

The additional arguments, `-ssl_cert ~/irc/yournick.pem` and `-ssl_pkey ~/irc/key.pem` specify your client certificate and your private key, respectively. After running the command, you should now be connected to IRC![2]

[1] This is easier than specifying or skipping each of the wonky fields interactively, which happens if you do not specify a `-subj` argument.

[2] If not then curse, take a deep breath, and carefully re-read the instructions. Failing that, feel free to contact someone knowledgeable for help.

# 3 Using irssi

This section provides a quick introduction for using the irssi client. If you already know how to use irssi or some other IRC client then you may skip this section.

## 3.1 Autoconnect

Unless you are a fan of circumlocution, you probably do not wish to re-type the preceding /connect command every time you connect to CotSS, thus it makes sense to configure irssi to connect automatically on startup. In order to do this it must first be understood that IRC consists of a *network* of interconnected *servers*. Therefore, in order to set up autoconnect we must tell irssi both the network **and** the server that we are connecting to. For CotSS, which has only one server, this seems rather redundant, but remember that most IRC networks consist of multiple servers and thus irssi reflects this fact. Start by running the command:

```
/network add cotss
```

. . . which adds a network with the name cotss. Next add the server to the cotss network:

```
/server add -auto -network cotss -ssl_verify \
    -ssl_cafile ~/irc/cotss.pem \
    -ssl_cert ~/irc/yournick.pem \
    -ssl_pkey ~/irc/key.pem \
    frostsnow.net 6697
```

This mirrors the /connect command, except instead of connecting it is configuring irssi to automatically connect to the server with the given arguments on startup. Save the configuration with:

```
/save
```

. . . and then restart irssi to automatically connect!

## 3.2 Navigation

irssi is divided into *windows*, which can contain either *channels* or *private messages*. Windows are *numbered*, with window number 1 holding status messages and subsequent windows numbered sequentially after that (2, 3, 4, . . . ). You can join channels by running /join <channelname> where <channelname> is, obviously, the name of the channel that you wish to join; note that, in IRC, channels start with a # symbol (which is **not** a Twatter hashtag). Likewise, you may message users by typing /msg <username> where <username> is, obviously, the nickname of the user that you wish to message. Both of these actions will open up a new window (/join will actually **activate** that window), and you can then navigate between windows by either typing /window <number> or by pressing Alt+<number> (depending on your terminal) where <number> is the number of the window that you wish to activate. More information and options may be found by typing /help window.

# 4 NickServ

By default, anyone may use any nickname that is not currently in use after connecting to IRC. In order to claim ownership of a nickname, one must *register* it with *services* by messaging the bot named NickServ and then *identify* to that nickname on subsequent connections. The easiest way to do this on cotss is to 1) register with NickServ and then 2) identify using your client certificate. In order to register, run the following command while on the nick that you wish to register:

```
/msg NickServ register <your_password> \
    <your_bogus_email>
```

...where <your_password> should be a **unique** password for your account (don't send me a password that you've previously used elsewhere, that's just stupid) and <your_bogus_email> is some gibberish (all e-mail functionality has been disabled for cotss). After that, you may manually identify to NickServ on subsequent connections by running the command:

```
/msg NickServ identify <your_password>
```

...but that's a pain, so the easier option is to add your client certificate's *fingerprint*, which is a unique identifier for that certificate, to NickServ's list of valid fingerprints for your account. You can add your current fingerprint to NickServ's list of valid fingerprints for your account by running:

```
    /msg NickServ cert add
```

...and you should now be identified automatically when you connect!

More information about NickServ's services may be found by running /msg NickServ help.

# 5 Inviting Friends

Depending on your client certificate's *extensions*, you may be able to *sign* additional client certificates, thereby inviting your friends to cotss. The policy for invites is dubbed *friend-of-friend*, meaning that, if you are a friend of me, the server admin, you may also invite your friends to join (taking into consideration that they aren't dicks, obviously), thus those that may be invited are the "friend of a friend". Sadly, the method for signing certificates is a pain-in-the-ass, mainly because you need to set up a directory as a *Certificate Authority*, including an OpenSSL configuration file, before you are able to sign certificates from that directory. In addition, you must create and submit a *Certificate Revocation List* (CRL) and send it to me, the server admin, before any of your friend's certificates will be accepted, though you will not need to update it as you invite friends unless you *revoke*, a.k.a. ban, one of them. This section of the guide will walk you through these processes.

## 5.1 Viewing Certificate Extensions

First, you'll probably wish to confirm that you are able to validly sign certificates of your friends before embarking on this endeavor. In order to do this, you must view the *X509v3 extensions* of your client certificate, which may be done by running:

```
$ openssl x509 -in ~/irc/yournick.pem -text \
    -noout | grep -A1 "Basic Constraints"
```

You should be familiar with the first part of this command by now; the second part, `grep -A1 "Basic Constraints"`, acts as a filter to limit the input to what we're interested in[3]. The two fields (comma-separated) that we are interested in are the `CA` and `pathlen` fields: the first field, `CA`, is a *boolean* (either true or false) specifying whether or not one is a Certificate Authority and thus able to sign certificates (`TRUE` if one is able to sign certificates and `FALSE` if one is not able to sign certificates); the second field, `pathlen`, is an integer that specifies how many more Certificate Authorities you may create with your certificate, and thus ought to be either omitted (because you are not a Certificate Authority and thus unable to issue certificates) or `0` (you may sign additional certificates, but the certificates that you sign are not also able to sign additional certificates). In summary, the two valid combination are `CA:TRUE, pathlen:0`, thus you may invite people by signing additional certificates, and `CA:FALSE`, thus you may not invite people.

## 5.2 Creating a CA

You must set up a special directory for your Certificate Authority; we'll use `~/irc/ca`. Run the following series of commands:

```
$ mkdir ~/irc/ca
$ cd ~/irc/ca
$ mkdir certs crl csr newcerts
$ umask 0077
$ mkdir private
$ umask 0022
$ touch index.txt
$ echo 1000 > serial
$ echo 1000 > crlnumber
```

You will also need to create an OpenSSL configuration file for your CA, please copy the file from Appendix A into `~/irc/ca/openssl.cnf`. The file is divided

[3] If you get no output, make sure you spelled "Basic Constraints" precisely, or try removing that part of the command and searching the output yourself.

into square-bracket ([]) delimited *sections*, of which a brief description is as follows: the ca section declares the default Certificate Authority section; the ca_default section declares various attributes of a Certificate Authority; the policy_anything section defines which fields of a certificate signing request must be specified in order for a CA to sign said request; the v3_friend section defines extensions that a signed certificate will have.

Your CA directory will expect your client certificate and private key to be in certain locations, you may either move them there (and then update your IRC client's configuration appropriately) or create a symbolic link to their actual locations; the following command does the latter:

```
$ cd ~/irc/ca
$ ln ../yournick.pem certs/cert.pem
$ umask 0077
$ ln ../key.pem private/key.pem
$ umask 0022
```

Your Certificate Authority should now be set up, but keep in mind that you will need to remember to manually specify the configuration file when you are using it, as is done in the following subsection.

## 5.3 Creating a CRL

Although you have created your CA, you must first send the server a *Certificate Revocation List* (CRL) file before it will accept any certificate that you sign for your friends. Generating the CRL file can be done by running the following series of commands:

```
$ cd ~/irc/ca
$ openssl ca -gencrl -config openssl.cnf \
        -cert certs/cert.pem \
        -keyfile private/key.pem \
        -out crl/crl.pem
```

Then send the file crl/crl.pem to me, the server admin, and I will reconfigure the server to use it. Also, I must apologize for the convolution here, but OpenSSL requires each CA to have a valid CRL before it will verify certificates signed by that authority, even if said CA has not revoked any certificates. What a pain!

## 5.4 Signing Friend's Certificates

In order to sign your friend's certificate, they must first create a *certificate signing request* (csr), as explained earlier in this guide, and then send the signing request to you. When you receive the signing request and place it in ~/irc/ca/csr/yourfriend.pem (with yourfriend.pem replaced by the nick of your friend), you should first verify that it is of the appropriate key *length* and has the correct *Common Name* before signing it by running:

```
$ openssl req -in ~/irc/ca/csr/yourfriend.pem \
    -text -noout
```

...and making sure that the "Public-Key" reads `4096 bit` (and not, say, `2048 bit` or lower) and that the "Subject" line reads something akin to `Subject: CN=yourfriend`[4].

Now that you've verified the certificate signing request, you may sign the certificate using your Certificate Authority by running the following commands:

```
$ cd ~/irc/ca
$ openssl ca -config openssl.cnf \
    -extensions v3_friend \
    -in csr/yourfriend.pem \
    -out certs/yourfriend.pem
```

The `-extensions v3_friend` tells OpenSSL to use the X509v3 extensions with the label `v3_friend`, which is in the `openssl.cnf` file as specified by `-config openssl.cnf`. The arguments which specify the input certificate signing request and the output client certificate should be self-explanatory. You may wish to verify the output client certificate by running `openssl x509 -in certs/yourfriend.pem -text -noout`; the output certificate should have a long expiration time (for your and their convenience), and also have a 4096-bit length key and be signed with SHA-512 and not, say, SHA-256[5] (your configuration file should guarantee this).

Then all you need to do is send your friend their signed client certificate (at `~/irc/ca/certs/yourfriend.pem`), and they will be able to connect!

## 5.5 Revoking a Friend's Certificate

In the unfortunate event that you need to revoke a friend's certificate, it may be done by running the following commands, where `yourfriend` is replaced with the name of the friend whose certificate you are revoking:

```
$ cd ~/irc/ca
$ openssl ca -config openssl.cnf \
        -cert certs/cert.pem \
        -keyfile private/key.pem \
        -revoke certs/yourfriend.pem
```

Although you have revoked your friend's certificate, you must now inform the server by updating the CRL file and sending the updated CRL file to me. Update the CRL file and send it to me as in section 5.3. You will have then completed revoking your (possibly former) friend's certificate.

[4] Note that key lengths below `4096` *may* be accepted by the server as-is, but are not supported and may break at any time, as weak crypto will be deprecated today and not tomorrow.

[5] Check the `Signature Algorithm` line in the certificate, and note that, although it may work with SHA-256 initially, weak crypto will be deprecated today and not tomorrow.

# A OpenSSL CA Configuration File

You may also find this file included in the repository containing these instructions[6].

```
# OpenSSL configuration file for friends of frostsnow.net
# This is used for signing of friend's certificate requests.

[ ca ]
default_ca      = ca_default            # The default ca section

[ ca_default ]
# Directory and file locations
dir             = ./                    # Where everything is kept
certs           = $dir/certs            # Where the issued certs are kept
crl_dir         = $dir/crl              # Where the issued crl are kept
new_certs_dir   = $dir/newcerts         # default place for new certs.
database        = $dir/index.txt        # database index file.
serial          = $dir/serial           # The current serial number
RANDFILE        = $dir/private/.rand    # private random number file

# Root key and cert.
private_key     = $dir/private/key.pem  # The private key
certificate     = $dir/certs/cert.pem   # The CA certificate

# Use better hash.
default_md      = sha512

name_opt        = ca_default            # Subject Name options
cert_opt        = ca_default            # Certificate field options
default_days    = 7200
preserve        = no                    # keep passed DN ordering
policy          = policy_anything
x509_extensions = v3_friend             # The extentions to add to the cert

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional

[ v3_friend ]
# Extensions for friends of friends.
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical,CA:false
```

[6] https://github.com/clinew/frostsnow.net at the time of this writing.