Table of Contents

1 INTRODUCTION

This is a user-space implementation of SEcure Neighbor Discovery (SEND) for IPv6. For all the gory details on SEND, please see the IETF RFCs online or in the docs directory. This package also includes libraries for generating and verifying Cryptographically Generated Addresses(CGAs) and X.509 Extensions for IP Addresses.

The primary goal for this project is to create a SEND implementation that is easy to use and portable. Hence this implementation is completely in user-space and self-contained, requiring no patches to your kernel or any other programs.

It works like a firewall filter between network devices and the IPv6 stack. All incoming and outgoing neighbor discovery (ND) messages are intercepted and sent to user space for processing by sendd (the mechanism is OS-specific). Sendd will add SEND options to outgoing ND messages to secure them, and will verify SEND options on incoming messages, dropping those packets with invalid options, and passing valid packets on back to the kernel for processing.

This is a research prototype. We focused on protocol correctness; there are no doubt bugs, as well as much to be done in hardening the daemon itself against attack and making it more robust and stable. Do not expect commercial-grade reliability and security. We will try to provide support, but can do so only as time permits.

2 SUPPORTED PLATFORMS

Currently the following platforms are supported:

2.1 Linux

We have tested SEND on a number of 2.6.x kernels, on a number of major distributions: Fedora Core 2 – 4, Ubuntu 5.10, and SUSE 10.0. It should work on any distribution running a correctly configured 2.6.x kernel.

Your kernel must have the following enabled:

- CONFIG_NETFILTER
- CONFIG_IPV6
- CONFIG_IP6_NF_QUEUE
- CONFIG_IP6_NF_IPTABLES
- CONFIG_IP6_NF_FILTER
- CONFIG_PROC_FS

Additionally, you need to ensure that the netfilter iptables user space utilities are installed (check for the ip6tables command), and that you have the netfilter **libipq** development library and headers installed. Check your distribution's package repository, or download the source from http://www.netfilter.org.

SEND on Linux uses netfilter's IP queuing mechanism to capture and reinject packets. Before sendd runs, you need to ensure that the appropriate netfilter rules are in place. After installation, you will find the scripts “sendd” and “snd_upd_fw” in /etc/init.d. You should use /etc/init.d/sendd to start sendd by default. If you want to run sendd directly from the command line, add the necessary rules with /etc/init.d/snd_upd_fw add. When done, you can remove the rules with /etc/init.d/snd_upd_fw del.

2.2 FreeBSD

We have tested SEND on FreeBSD version 5.4. Other versions may work, but have not been tested.

Your kernel needs netgraph(4) support, with support for the BPF, SOCKET, and ETHER node types. These correspond to the following kernel configuration options:

- NETGRAPH
- NETGRAPH_BPF
- NETGRAPH_ETHER
- NETGRAPH_SOCKET

Additionally, you need to have libdnet installed (available from http://libdnet.sourceforge.net/).

3 DEPENDANCIES

In addition to platform-specific dependancies mentioned above, the following are also required for all platforms:

- libcrypto 0.9.7 or greater, library and development headers
- GNU make
- GCC (tested with 3.3.2 – 4.0.0)
- lex
- yacc
- optional: libreadline, ncurses libraries and development headers.

Most major distributions either already have these installed, or make them available from their package archives.

4 INSTALLATION

1. Edit the top level Makefile.config. You must set your OS type here, as well as some other OS-specific paramaters, and you can also set a number of optional paramaters.
2. make
3. make install

A successful build will result in the installation of three binaries:

```
sendd      The SEND daemon
cgatool    Tool for configuring CGAs
ipexttool  Tool for configuring PKIX IP certificate extensions
```

Additionally, os-specific start scripts may also be installed.

5 CONFIGURATION

There are two levels of configuration: a basic level for hosts that will not be participating in router discovery, and an additional level for those that will participate in router discovery.

5.1 BASIC CONFIGURATION

You must complete two steps:

1. Generate CGA parameters and a CGA
2. Configure sendd.

You must generate CGA parameters and at least one address. To do so, you need a RSA key pair. You can use a preexisting one from a PEM file or a certificate, or generate one with cgatool or openssl(1). For CGA generation, only the public key is needed (although you will need the corresponding private key for sendd).

Next you feed the key to cgatool along with a prefix to create your CGA parameters and a CGA. Probably the most straightforward way to do all this is with cgatool. The following example generates a new RSA key pair of 1024 bits, and then uses the key to generate a CGA and CGA parameters with a CGA sec value of 1 and a prefix of 2000::/64:

```
# cgatool --gen -R 1024 -k mykey.pem -p 2000:: -o myder -s 1
```

This puts the newly generated RSA key pair in mykey.pem and the DER-encoded CGA parameters in myder. The CGA is printed to stdout. For more information on cgatool, see cgatool/README. Once you have your new CGA, you should configure it on an interface (for example, using ifconfig(1)), and update your system configuration to configure the CGA upon boot.

You only need to generate CGA parameters once, before you run sendd for the first time. However, you can use cgatool to generate additional CGAs based on the initially generated CGA parameters.

Now you can configure sendd. Sendd reads its configuration from a file (the default location of which is /etc/sendd.conf). You can also specify an alternate configuration file with the -c command line argument. The file has a key-value format, i.e.

```
key=value
```

Any line beginning with a '#' is considered a comment. You can copy and edit the sample sendd.conf provided with this distribution.

The following setting is mandatory:

snd_cga_params	Full path name of a file containing CGA parameters specifications. See below for information on this file. *No default setting*

The following settings are optional:

snd_addr_autoconf	If “yes”, sendd will automatically generate CGAs based on prefixes received in router advertisements. *Default = yes*
snd_cga_minsec	The minimum CGA sec value this host will accept from peers. *Default = 0*
snd_pkixip_conf	Location of this host's IP Extensions configuration file. Only needed for router discovery; see below. *No default setting*

snd_full_secure
: If “yes”, sendd will drop all incoming ND messages that have not been secured with SEND. If “no”, sendd will allow unsecured ND messages. This setting is useful for transition to SEND. *Default = yes*

snd_replace_linklocals
: If “yes”, sendd will replace all non-CGA linklocals with CGAs on startup and during operation. *Default = yes*

snd_timestamp_cache_max
: Sets an upper limit on the number of entries sendd will keep in its timestamp cache. *Default = 1024 entries*

snd_timestamp_delta
: The amount of time (in seconds) that a peer's clock can differ from the local host's clock. See RFC3971, section 5.3.4.2. *Default = 300 seconds*

snd_timestamp_drift
: See RFC3971, section 5.3.4.2. *Default = 1%*

snd_timestamp_fuzz
: See RFC3971, section 5.3.4.2. *Default = 1 second*

snd_thrpool_max
: If compiled with multi-threading support, sets the maxiumum number of threads that sendd will spawn to handle cryptographic operations. Default is 2; hosts with multiple processors or multiple cores may see some scalability gains by increasing this value.

There are other undocumented configuration settings; you should have an understanding of the code to play with these.

It is necessary to keep your host's clock synchronized to within your snd_timestamp_delta setting.

Sendd can handle various levels of granularity for CGA parameters, from a single set of parameters for all addresses and interfaces on a host, down to different CGA parameters for each address or interface. The sendd CGA parameters configuration file allows you to assign CGA

parameters to addresses and interfaces. The file is comprised of sections. There are two different types of sections: “named” and “address”. “named” sections allow you to define CGA parameters that can be used by address sections as well as other named sections. “address” sections assign CGA parameters to an individual address. The file is formatted as follows:

```
named <name> {
     # CGA parameters specified here
}

address <address> {
     # CGA parameters specified here
}
```

A section contains key-value pairs terminated with a ';', separated by a space.

Each section can specify CGA parameters or simply use a set of named parameters. There MUST be at least one named section, “default”, specifying the default CGA parameters to use. To specify CGA parameters, a section must contain these values:

snd_cga_priv	Full path name of a file containing the RSA private key corresponding to the public key used to generate the CGA parameters.
snd_cga_params	Full path name of a file containing the DER-encoded parameters generated by cgatool.
snd_cga_sec	The CGA sec value used to generate the CGA parameters.

For example:

```
named default {
     snd_cga_params /etc/sendd/cga.params;
     snd_cga_priv   /etc/sendd/key.pem;
     snd_cga_sec    1;
}
```

To use named parameters instead of explicitly specifying a set of parameters, provide the “use” value. For example:

```
named foo {
     use default;
}
```

Address sections must contain, in addition to parameters specifications or a “use” directive, an interface directive naming

the interface on which the address is configured. For example:

```
address 2000::38cb:3d3d:14ad:cb08 {
     use foo;
     interface eth0;
}
```

There is one special type of named section: If the name corresponds with an actual interface, when sendd autoconfigures a new address on that interface it will use the parameters from that section. For example:

```
named eth0 {
     use foo;
}
```

5.2 ROUTER DISCOVERY CONFIGURATION

If you need this configuration, complete this section first, and then then basic section, using the generated keys to generate CGA parameters.

There are two aspects to this configuration:

1. Routers must be configured with a set of certificates that prove their authority to act as a router and advertise a set of subnet prefixes.
2. Hosts must be configured with one or more trust anchor certificates with which to verify router certificates.

5.2.1 ROUTER CONFIGURATION

You need to create a certificate path with at least one certificate, and use ipexttool to add IP Extensions to it. The OpenSSL toolkit is freely available, and can be used for this purpose (although it entails a somewhat tedious and confusing process). There is a sample script in example/ipext that can ease the process. Otherwise, you need to do something along the lines of the following example that shows how to do this with the OpenSSL toolkit.

First create a CA:

```
  # CA.pl -newca
```

(CA.pl(1) lives in the ssl installation's misc directory; you can also use CA.sh if you do not have perl installed).
The CA's certificate will be in demoCA/cacert.pem.

Now create the certificate for the next entity on the path.
Generate a key (this example uses 1024 bits; others will work too):

```
  # openssl genrsa -out <priv key file name> 1024
```

Generate a certificate request for the new entity:

```
  # openssl req -new -key <priv key file name> -out newreq.pem
```

Do not enter a password, unless you plan to always start SEND daemons interactively (since you will be prompted for a password when reading the private key).

Use CA.pl to sign the request.

```
  # CA.pl -sign
```

This creates a certificate path two deep.

If you want to create a deeper path, you need to replace the original CA's information directory (demoCA) with the second level entities certificate and keying material. To do this, create a new demoCA directory structure for each certificate in the chain, using 'CA.pl -newca'. For example, say you have a certificate in mid_cert, and you want to use it to sign a new certificate request just generated by openssl ... -newreq:

```
  # CA.pl -newca
  CA certificate filename (or enter to create)
  dir_cert

  # CA.pl -sign
  # mv newcert.pem <your cert name>
```

Next you must add a PKIX IP extension to each certificate and resign the certificate using ipexttool.

See the section below on ipexttool for directions on how to do this.

Once you have finished, edit your sendd.conf to add the location of the IP extensions configuration file, i.e.

```
  snd_pkixip_conf=/etc/sendd/ipext.conf
```

5.2.2 HOST CONFIGURATION

The host must be configured with at least one trust anchor certificate. A trust anchor can be any entity in the signing path of the router's certificate path (usually it can just be the CA certificate).

For each trust anchor certificate on the MN, add a trustedcert entry to the pkixip_conf file. For example:

```
files {
     trustedcert /usr/certs/certs/ca.pem;
     trustedcert /usr/certs/certs/lvl1.pem;
```

}

The trust anchor certificates should include the IP Extensions needed to authorize any routers the host may encounter.

Next edit sendd.conf, setting the snd_pkixip_conf key to the locatoin of the pkixip_conf file.

It is possible to configure a host to accept certificates unconstrained by IP Extensions. In this configuration, the host will accept any prefixes advertised by a router if the router's certificate does not contain any IP extensions. For security reasons, this configuration is disabled by default. To enable it, set the “snd_accept_unconstrained_ra” option to “yes” in sendd.conf.

It is recommended that kernel address auto configuration be disabled, since sendd will auto configure CGAs based on received prefixes.

6 RUNNING SENDD

Once your configuration is ready to go, use your system start script to run sendd. For Linux, this will /etc/init.d/sendd, and for FreeBSD, this will be /etc/rc.d/sendd (you will also need to enable sendd in rc.conf). By default it will run in the background as a system daemon. If you run sendd by hand, it takes the following command line arguments:

-c <conf>	Use an alternate configuration file
-f	Run in the foreground.
-i <iface>	Restrict SEND to running on this interface. This can be repeated for additional interfaces.
-l <method>	Specify where to output logging messages. Choices are “stderr”, “syslog” and “none”.
-V	Display version information and exit.

7 DEBUGGING

You can get an interactive console on sendd and cgatool if you set USE_CONSOLE=y in Makefile.config. Run sendd with the -f flag, and cgatool with the -i flag. The consoles can display lots of internal state, and with cgatool, you can interactively generate and verify CGAs.

For lots more debugging info, set DEBUG_POLICY=DEBUG in Makefile.config. Now ipexttool, cgatool, and sendd provide a -d command line flag to turn on debugging output. You can repeat the -d up to three times with sendd to get even more debugging output.

With debug enabled, you can fine-tune which sendd sub components produce debugging output. On the console, you can get a list of

available debugging levels with the “debug_levels” command. You can turn individual levels on or off with the “debug_on” and “debug_off” commands. For example, to enable debugging output for sendd certificate processing, do

```
sendd> debug_on sendd cert
```

You can also use “all” as an argument to “debug_on” and “debug_off” to turn on or off all debugging.

It is also possible to specify specific debug levels in sendd.conf, with the snd_debugs key. To enable sendd cert and proto:

```
snd_debugs=sendd:cert,sendd:proto
```

Note: this only works with debug levels within the sendd context.

8 CGATOOL

cgatool is a CLI front-end to the CGA library included in the distribution. It allows you to generate and verify CGAs.

8.1 GENERATION

When generating a CGA, use the -g or –gen command line argument. To generate, you must provide a key, an IPv6 prefix, and a CGA sec value. There are four ways to provide a key:

1. Provide a certificate with -C or --certfile.
2. Provide a PEM-encoded RSA key pair with -k or --keyfile.
3. Generate a RSA key on the fly with -R or --rsa <bits>. You must also provide a keyfile with -k to which to write the new key.
4. Provide DER-encoded CGA parameters with -D or --derfile.

Provide an IPv6 prefix with -p or --prefix <prefix>.

Provide a CGA sec value with -s or --sec <sec value>.

When generating, you must also provide a derfile with -D to which to write the new DER-encoded CGA parameters.
Some examples:

Provide the key from mykey.pem:

```
# cgatool -g -k mykey.pem -o myder -p 2000:: -s 1
```

Provide the key from myder:

```
# cgatool -g -D myder -o myder -p 2000:: -s 1
```

Generate from the example parameters provided in rfc3972:

```
# cgatool --gen -D rfc_example.params -o myder -p fe80:: -s 1
fe80::3c4a:5bf6:ffb4:ca6c
```

The amount of time needed for CGA generation depends on the speed of your hardware and the sec value. You should choose the largest sec value your hardware and patience can reasonably handle. On a 2GHz Pentium 4, sec=1 usually takes just a few milliseconds, while sec=2 takes at least a few hours. The faster your hardware (and the more patient you are), the larger the sec value you can use. The largest possible sec value is 7.

If you provide the key from a derfile, cgatool will use the modifier in the CGA parameters, and will not search for a new modifier.

Once finished generating, cgatool will print the new CGA to stdout, and write the CGA parameters to the provided derfile.

8.2 VERIFICATION

You will ordinarily not need to manually verify CGAs. This functionality is provided for experimentation and sanity checks.

When verifying, use the -v or --ver command line argument. To verify, you must provide the CGA to be verified, and the CGA's DER-encoded parameters. Provide the address with -a or --address, and the derfile with the -D or --derfile argument. For example:

```
# cgatool --ver -a 2000::2073:8e00:6d:aa09 -D myder
```

8.3 CGATOOL CONSOLE

Run cgatool with the -i or --interactive command line argument. You can set all the arguments one-by-one, and use the “show” command to display current CGA context state.

If you set USE_THREADS=y in Makefile.config, you can also use multiple threads to search for the CGA modifier in parallel. (Of course, this is only useful if you have a multi-processor and / or multi-core system). Set the number of threads to use with 'thrcnt <num>'. While generating, cgatool will search a certain number of modifiers, and then check for interrupts (i.e. You can halt generation with ^C). The number of modifiers searched between interrupt checks is called the batchsize. You can change this value with the 'batchsize <num>' command. The default batchsize is 500000.

8.4 DYNAMIC CGA PARAMETER ADMINISTRATION

Cgatool can be used to dynamically add or remove CGA parameters on sendd.

To add new parameters, use -A or --add. To add new named parameters, provide the new name with -N. To add new address parameters, provide the address with -a and the interface with -I. To specify new parameters, use -D to provide the DER-encoded parameters, -k to provide the key file, and -s to provide the sec value. To use preexisting named parameters, provide the name with -U.

For example, to add new named parameters “foo” that use the “default” parameters:

```
# cgatool --add -N foo -U default
```

Add new address parameters:

```
# cgatool --add -a 2000::3c44:d77d:3db1:9696 -s 1 -I eth0
     -D /etc/sendd/cga.params -k /etc/sendd/key.pem
```

If you look closely at the command line help synopsis, you might notice the --sigmeth (-S) switch to select a signature method. There is only one signature method in use, the one defined in RFC 3971, so currently this switch does not need to be used. In the future, if new signature methods are defined, this switch will come into use.

cgatool can also delete parameters with the --erase or -E switch. To erase address parameters, specify the address with -a:

```
# cgatool --erase -a 2000::3c44:d77d:3db1:9696
```

To erase named parameters, provide the name with -N:

```
# cgatool --erase -N foo
```

9 IPEXTTOOL

ipexttool is a CLI front-end to the IP extensions library. It allows you to add IP extensions to certificates, and verify chains of certificates with IP extensions.

To add or verify IP extensions in certificates, you must first specify a PKIX IP extension using ipexttool's configuration file. This file contains one section where you describe the PKIX IP extension, and another where you specify the locations of input, signing, and output certificates.

Following is a simple example of such a configuration file:

```
addresses {
     ipv6 {
          SAFI unicast;
```

```
          prefix fec0:0:0:1::/64;
          prefix fec0:0:0:2::/64;
          prefix fec0:0:0:3::/64;
          prefix fec0:0:0:4::/64;
     }
}

files {
     certfile /usr/src/fmip/send/pkixipext/certs/ca.pem;
     cacert /usr/src/fmip/send/pkixipext/certs/ca.pem;
     capriv /usr/src/fmip/send/pkixipext/certs/ca_priv.pem;
     outfile /usr/src/fmip/send/pkixipext/certs/ca_ipext.pem;
}
```

The first section, "addresses", is where you put lists of prefixes, ranges, or "inherit" directives. The first subsection contains address family blocks for either IPv4 or IPv6 addresses. The first directive in an address family block must be "SAFI", which stands for "subsequent address family identifier". Allowed SAFI values are "unicast", "multicast", and "both". "Both" means both unicast and multicast.

After SAFI comes one or more prefixes or ranges. A prefix is described by an address followed by '/' followed by a prefix length. A range is described by "range" followed by a minimun address and then a maximum address. The minumum address is followed by '/' and the number of zero bits in the address. The maximum address is followed by '/' and the number of bits that are set to 1. See rfc3779 section 2.2.3.9 for more information on this.

The files section tells ipexttool where to get and put certificates.
  "certfile" is the input certificate.
  "cacert" is the input signer's certificate.
  "capriv" is the input signer's private key.
  "outfile" is where to put the newly signed certificate.

Now use ipexttool to add the IP extensions and resign the certificate:

```
% ipexttool -w -i <conf file>
```

Now you have should have a new certificate in outfile, correctly signed and containing the PKIX IP extension you specified. Note that the keying material has not changed.

You can use ipexttool to verify that the IP extension has been written to the outfile:

```
% ipexttool -p -c ca_ipext.pem
     ...
```

```
        X509v3 extensions:
     ...
            PKIX IP Addr Extension: critical
                IPv6 (Unicast)
                    Prefix or Range
                        Prefix:fec0:0:0:1::/64
                        Prefix:fec0:0:0:2::/64
                        Prefix:fec0:0:0:3::/64
                        Prefix:fec0:0:0:4::/64
```

You must repeat the process for each certificate in the delegation chain. Once you have done this, you can use ipexttool to verify that you have created the chain correctly. For instance, say you have created the following delegation chain (DNs abbrieviated): ca -> lvl1 -> lvl2 -> lvl3, and that lvl3 has been delgated authority to route the fec0:0:0:4::/64 prefix. Assuming that the certificates are in ca.pem, lvl1.pem, lvl2.pem, and lvl3.pem, create a new configuration file setting ca, lvl1, and lvl2 as trusted certificates and lvl3 as the certificate to be verified:

```
addresses {
     ipv6 {
          SAFI unicast;
          prefix fec0:0:0:4::/64;
     }
}

files {
     trustedcert /usr/certs/ca.pem;
     trustedcert /usr/certs/lvl1.pem;
     trustedcert /usr/certs/lvl2.pem;
     certfile /usr/certs/lvl3.pem;
}

% ipexttool -v -i <conf file>
```

If the verification succeeds, ipexttool will complete silently.

Once you are satisfied that your configuration is correct, you need to modify sendd's configuration on the AR and client nodes. On the router, create a configuration file specifying all the certificates needed to complete the delegation chain (including the AR's certificate). (At this time there is no support for dynamic certificate lookups, so all certificates must be statically configured). The host certificate is indicated with "certfile", all others in the chain by "trustedcert". For example:

```
files {
     trustedcert /usr/certs/ca.pem;
     trustedcert /usr/certs/lvl1.pem;
```

```
     trustedcert /usr/certs/lvl2.pem
     certfile /usr/certs/lvl3.pem
}
```

10 LIMITATIONS

Since this implementation is completely independent of the kernel, there are a few architectural limitations worth mentioning.

RFC3971 states that if a ND packet is received and its time stamp is outside the permitted delta, the packet SHOULD still be processed, but it should cause no change to the neighbor cache. Since this implementation has no direct control over the neighbor cache, it cannot follow this advice. Instead, it takes the “better safe than sorry” approach and drops the packet. The work around for this is to keep your hosts' clocks reasonably in sync.

This implementation does not handle DAD collisions, since it would be too difficult to make this robust in the face of the kernel DAD process.