Network Working Group   Craig Metz
Internet Draft   The Inner Net
draft-metz-net-security-api-01.txt   16 January 1998

Network Security API for Sockets

Status of this Memo

This document is an Internet Draft. Internet Drafts are working documents.

Internet Drafts are draft documents valid for a maximum of 6 months. Internet Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet Drafts as reference material or to cite them other than as "work in progress".

A future version of this draft will be submitted to the RFC Editor for publication as an Informational document.

Abstract

This API is a means for sockets applications to request network security services from an operating system. It is designed to move most of the work and intelligence of security policy processing into the operating system so that the burden on application authors is light enough to encourage the use of network security.

It is documented here for the benefit of others who might also adopt and use the API, thus providing increased portability of applications that use network security services (e.g., the IP Security ESP and AH protocols).

1. Introduction

Many network protocols now provide security services such as encryption and authentication at the network layer. For example, IPv4 supports and IPv6 requires the IP Security protocols, ESP and AH. While various flow-based policy schemes can frequently identify the security requirements of a particular packet, applications and the end user should be able to provide input to the policy process and request security services from the system. That is the main goal of this application programming interface: to provide a means for applications (and, through them, the end user) to request security services and properties from the system.

Secondary goals of this API include moving most of the burden to the system, thus making it easier for the application programmer to use security services, supporting complex policy decisions with reasonable performance, and giving the application more input to and feedback from the policy process than is provided for in similar APIs.

This API is built as an extension to the POSIX p1003.1g sockets interface. That interface is REQUIRED for this API. This API assumes that network security services follow a conceptual model similar to that of IP Security. This interface may need to be changed in the future to support protocol families that differ radically from that model.

While not required to use this API, it is intended that the PF_KEY key management API be used in systems that implement this network security API for sockets. Readers of this document who have not read the PF_KEY specification are encouraged to do so in order to understand the context for some of the capabilities of this API.

This API is intended to be usable with any network protocol supported by the POSIX p1003.1g sockets interface. However, because it leads to extra code complexity and it is almost never desirable, this API MUST NOT be used with protocol families that are only capable of system-local communication. Such protocol families include PF_LOCAL (i.e., PF_UNIX), PF_ROUTE, and PF_KEY).

1.1. Terminology

Even though this document is not intended to be a standard, the words that are used to define the significance of particular features of this interface are usually capitalized. Specific behavior compliance requirements are itemized using the requirements terminology (specifically, the words MUST, SHOULD, and MAY) defined in RFC 2119. In addition, the following terms should be noted:

- CONFORMANCE and COMPLIANCE

Conformance to this specification has the same meaning as compliance to this specification. In either case, the mandatory-to-implement, or MUST, items MUST be fully implemented as specified here. If any mandatory item is not implemented as specified here, that implementation is not conforming and not compliant with this

specification.

- IMPLEMENTORS

Those who are building a software implementation that uses this API. If not otherwise specified, this term refers both to application implementors and to system implementors. Many of the concepts and caveats of this API need to be carefully noted by both.

- ULP

An upper-layer protocol (ULP) is an opaque payload for purposes of security processing. This can be transport protocol (e.g., TCP or UDP), a control protocol (e.g., ICMP or IGMP), or another network protocol (e.g., IP or IPv6).

1.2. Conceptual Model

This section describes the conceptual model of a system that implements this API. It is intended to provide background material useful to understand the rest of this document. Presentation of this conceptual model does not constrain an implementation to strictly adhere to the conceptual components discussed in this section.

Systems implementing this API are expected to have a "policy engine". This term is used to refer to whatever components of a system that have programmed with rules that control what security operations and parameters are allowed and which are preferred for given requirements. In many cases, flow information determined from the contents of network packets and the rules in this policy engine will completely satisfy the security needs of an application without the need for this API. For example, if a policy engine is programmed with a rule that tells it to require that all TCP packets with a foreign port of 23 be encrypted, outbound telnet connections will be encrypted without the need for the telnet client itself to request that encryption. The problem with this example is that all outbound telnet connections are encrypted whether or not the user wants it. In order to make the choice of whether or not to encrypt available to the user, the telnet client needs to specify an input to the policy engine that reflects the user's desire to encrypt or not. That is one of the things that this API does.

This API is built on a model where the policy engine makes the decisions for the system. It has complete and final control over what, if any, security processing is done on a packet and the parameters for that processing. The application makes requests (presumably representing the needs of some user) to the policy engine that reflect what it would like the policy engine to do with its

packets. The policy engine can, if so programmed, completely or selectively ignore this request. In some environments, ignoring some or all of an application's request is critical to maintaining system security; in others, it is inappropriate and frustrating. This API is not concerned with the programming of the policy engine and how the policy engine acts on an application's request, only how the application makes that request and receives feedback from the policy engine on what actually happened.

A complete network security implementation requires many components beyond those just described, but these are all hidden from the application behind the system's policy engine. Figure 1 illustrates a possible system organization to show where this API applies.

```
                     User
                       |
              Network Application        Key Management Daemon
               |                 |        |                 |
====Network Security API=====PF_INET Sockets=======PF_KEY Socket====
                                     |                      |
                                  TCP/UDP           PF_KEY Messaging
                                     |                      |
                             Policy Engine-----------------SADB
                                   |      |
                               IPsec---IP
                                          |
                                        Link
```

Figure 1 - A Possible System Organization

This API uses abstract security properties instead of specific values. That is, an application might request encryption and ask that it be "stronger," but it does not specify the exact security transforms or cryptographic algorithms to be used. This design choice was made for several reasons.

First, this allows applications to take advantage of network security services with the least amount of involvement in the inner workings of the system. Security associations, policy rules, available transforms, and available algorithms may change during an application's lifetime. These changes could make an application's request invalid or less desirable. Using abstract values, the policy engine can simply remap the abstract values to a new set of actual operations and parameters without the intervention of the application. Other approaches would place a more significant burden on the authors of applications that wish to use network security services.

Second, this allows the policy engine greater flexibility in combining the user's request with system policy rules. A request for a specific algorithm or transform does not tell the policy engine what the application's and/or user's requirements are or what properties are expected of that specific choice.

Third, some applications will come in binary-only form and will try to select their security properties without user intervention. By abstracting the algorithms in use, a system administrator has the ability to change what actual algorithms and parameters are in use without the need for changing every such binary on the system.

2. Requests

The model chosen for the network security requests themselves could be compared to a set of recipes. Each recipe is a step-by-step listing of the steps that should be taken to achieve the result, but those steps might not get executed exactly as requested if the executor "knows better". Of the possible recipes, a few will be available for frequent use, and only one can be executed at a time. Typically, only one will be chosen and executed, but there are situations in which the ability to execute one of several quickly (and cache preparatory steps) is desirable.

2.1. Requests

Each request is a list of operations that specifies what security properties an application would like performed on its packets. There are four currently defined operations:

Authenticate (A): Verify that the sender is as claimed and that the packet has not been changed in transit. This operation provides the properties of authentication and integrity.

Encrypt (E): Protect the data from receipt by unauthorized parties. This operation provides the property of confidentiality.

Encapsulate (N): Prepend a new network header to the packet. This allows applications to create half-tunnels "on-the-fly".

For example, an application might make a request of:

```
A N E
```

And the system policy engine might translate that and build a

packet that looks like:

```
IP ESP [ IP AH ULP ]
```

But the system policy engine could also build a packet that looks like:

```
IP ESP [ ULP ]
```

On the surface, this looks like the policy engine is being unreasonable or denying service. But consider the "optimization" policy rules that could turn that request into that packet:

1. Combine A and E into an ESP combined encryption/authentication transform and put the combination in the place of E.

2. Combine N with adjacent network headers to prevent encapsulation with nothing between the two network headers.

Implementors must always remember that requests are just that. The policy engine controls what services are delivered.

Requests are always ordered such that the first operation is the one the application would like to be performed closest to the upper-layer protocol. Another way to look at this is that the first operation in the request is the one the application would like to be the first operation performed in output security processing and the last operation performed in input security processing. For example, a mapping scheme that used the application's ordering might map:

```
A E          to   IP ESP [ AH ULP ]
E A          to   IP AH ESP [ ULP ]
A N E        to   IP ESP [ IP AH ULP ]
```

2.2. Preferences

Associated with each operation is a value that specifies the application's preference towards that operation actually taking place. There are four currently defined preferences:

Default (d): The application has no preference about this operation being performed or not. The operation is included to specify properties of the operation should it take place and/or as a place-holder. In absence of any other policy information suggesting otherwise, a system SHOULD default to not performing the operation.

Use (u): The application prefers, but does not require, that the operation take place.

Require (r): The application requires that the operation take place and requests that processing of the request halt if the operation cannot be performed.

Never (n): The application requires that the operation not take place and requests that processing of the request halt if the operation cannot be performed. For example, key management applications would need to use this request to prevent the system from attempting to provide security services for them and creating a catch-22.

Policy engines SHOULD grant the application's request to cause processing of the request to halt if require or never preferences cannot be satisfied. For input processing, this would result in the request not being matched and, if the request is the last in the group, the packet being dropped. For output processing, this would result in the packet being dropped and an error being returned to the application. It is a serious security problem for processing to not fail if the application has requested it. For example, confidential data could then be sent out as cleartext if key management fails.

Consider as an example the request:

Au Er

Suppose that the system's policy engine mapped this into an attempt to use AH and ESP. If key management failed to obtain a SA for AH but was able to obtain a SA for ESP, communication could continue and the actual packet would look like:

IP ESP [ULP]

However, if the ESP SA could not be obtained, regardless of whether key management could obtain the AH SA, the packet SHOULD be dropped and an error returned to the application.

2.3. Barriers

Barriers (b) are the most difficult concept in this API. They are a flag on some operations that asks the policy engine to maintain a separation located to the right of that operator. The policy engine SHOULD NOT re-order or combine operations across or through this barrier. Thus, barriers ask the policy engine to prevent certain kinds of optimizations from taking place. For example, consider the case of an application that deliberately wants to superencrypt its packets. A policy engine might have a rule that combines consecutive encryption operations. A request of:

E E

Would map to:

```
IP ESP [ ULP ]
```

But a request of

```
Eb E
```

Would map to:

```
IP ESP [ ESP [ ULP ] ]
```

If that policy engine is aggressive in its attempts to optimize security operations (because fewer operations performed means better performance), it might try to combine encryption operations even across an encapsulation operation. This would lead to internal steps like:

```
E N E        (Re-order: move the second E right)
N E E        (Combine: E + E = E)
N E          (Combine: remove leading N since there's a header
                       immediately preceding it)
E
```

This results in something that probably isn't quite what the application expected when it made the original request. Insertion of a barrier solves this problem:

```
E Nb E
```

The barrier is "located" between the N and E requests, though the actual flag is on the N request. Now the two E requests cannot be moved and/or combined, so the desired behavior is delivered.

Barriers also affect how a policy engine using certain kinds of designs would make decisions based on the application's request. More detailed discussion on this topic is outside the scope of this document.

As a general rule of thumb, most encapsulation requests SHOULD be flagged as barriers and most other requests SHOULD NOT be flagged as barriers. Except as prevented by barriers, implementations SHOULD perform optimization steps such as re-ordering and combining compatible operations to attempt to decrease the amount of processing necessary for a packet.

2.4. Multiple Requests

A request specifies one possible arrangement of operations to protect an application's packet. An application might provide the kernel with multiple requests. These are used differently for output and input security processing.

For output, providing multiple requests allows an application to quickly flip between a small set of them and therefore change the security properties it desires for its packets. This could also be done by loading the requests into the kernel as they are needed, but providing multiple requests to the kernel has less overhead (instead of having to specify the entire request when the application wishes to change, it only has to specify an index; a system call can also be saved by using control messages) and it allows the system to cache state information associated with each request. Note that only one request may be used with any given network packet on output.

For input, providing multiple requests allows an application to provide several possible acceptable input policies (in preference order) to try matching an incoming packet against. This can be used in applications that might allow communication with different security properties but might behave differently based on what properties are present. For example, one might use this capability to connect an incoming connection to a different daemon depending on whether or not its packets were encrypted. Also, this can be used in conjunction with output switching to build datagram servers that can "match" the properties of incoming packets -- packets that were received encrypted could then be sent encrypted responses, while packets that were received as cleartext would be sent cleartext responses.

Consider as an example a UDP datagram server. It might make the following request:

```
0. Au Nr Er
1. Ar Er
2. Er
3. Ed
```

A packet comes in:

```
IP ESP [ IP ULP ]
```

The application receives the packet and a notification that request zero was matched. It then sends its reply and notifies the kernel that it wishes to use request zero for that packet. All SAs are successful, so the reply looks like:


```
IP ESP [ IP AH ULP ]
```

Note that the reply doesn't look exactly the same as the original packet. This is a feature of using the "use" preference. If exact matching of the input and output specifications is required, the use and default preferences MUST NOT be used.

2.5. Latching

Stream sockets present a special problem because there is generally not a correlation between output boundaries at the application layer and at the network layer. Consider this sequence of events:

* Open a stream socket.
* Load a request of:
  1. Er
  2. En
* Select request 1.
* connect() to a remote end.
* write() a byte of secret data.
* Select request 2.
* write() a byte of non-secret data.

Typically, the two bytes written would be combined by a stream transport protocol into one packet. But should that packet be encrypted or not? Either encrypting or not encrypting that packet violates one of the requests. Stream protocols like TCP can also retransmit packets and slice/combine packets while retransmitting, which complicates things more. Some might try to modify the implementations of stream protocols to "tag" ranges of data with security properties and prevent incompatible combinations and ensure that the correct properties are present on retransmitted packets. Doing so is complex and tends to turn the stream protocol into a reliable datagram protocol, which has very different properties.

Therefore, all implementations of this API MUST implement a "latching" behavior for stream protocols that "latches in" the first request that is used to successfully process a packet for a connection and does not allow any other request to be used for the lifetime of that connection. Consider this example:

```
Both client and server open a TCP stream socket.
Both client and server load a request of:
  1. Er Ar
  2. Er
  3. En
Client selects request 2 and issues a connect(). The first
  packet in the handshake "latches in" request 2 for the
```



```
  lifetime of the TCP connection.
Server receives the first packet in the handshake from
  the client. It first tries to match request 1, fails,
  and then tries to match request 2, which succeeds. The
  server then creates its connection state for the new
  connection and "latches in" request 2 for that new
  connection. Note that the accepting socket is "latched",
  but NOT the listening socket.
The handshake completes and data flows. All packets beyond
  the initial exchange are required to meet the criteria
  in request 2.
```

Latching also applies to identity information. The innermost local and remote identity used for the security associations used to process the first packet input and output MUST be used for all subsequent security associations allocated to a stream socket (including and especially on "rekey" operations). This precludes attacks where a connection could be "hijacked" by rekeying with a different identity without the application's knowledge.

Implementations MUST NOT extend the latching behavior from the lifetime of the connection to the lifetime of the socket. It is a legitimate behavior for a stream socket to be connected and latched, disconnected, and connected and latched again with possibly different security properties. Sockets implementations have historically not correctly handled disconnecting a stream socket and connecting it to a new endpoint; now that these problems are finally being fixed, implementations of this API MUST NOT reintroduce this problem.

3. Detailed Interface

This section discusses the actual symbols, structures, and function calls used with this API. These are all based on the concepts discussed in the previous section.

3.1. Name Space

This network security API defines preprocessor symbols that start with the prefix "NET_SECURITY" and other names that start with the prefixes "net_security" and "__net_security". These are all defined as a result of including the header file <net/security.h>.

Inclusion of the file <net/security.h> MUST NOT define symbols or structures in this name space that are not described in this document without the explicit prior permission of the author. An implementation that fails to obey this rule IS NOT COMPLIANT WITH THIS SPECIFICATION and MUST NOT make any claim to be. This rule also applies to any files that might be included as a result of including

the file <net/security.h>. This rule provides implementors with some assurance that they will not encounter name space-related surprises.

3.2. Request Format

An application using this API gives the kernel zero or more requests that describe the set of security operations that the application requests for its packets.

Two constants define the limits on these requests. The first, NET_SECURITY_REQUEST_MAX, is the maximum number of requests that an application can load for each socket. Note that the requests themselves are numbered starting at zero. Therefore, the last request is numbered (NET_SECURITY_REQUEST_MAX-1). The second, NET_SECURITY_OPERATION_MAX, is the maximum number of operations that can be in a request. Both of these constants MUST have a value of greater than four and NET_SECURITY_REQUEST_MAX MUST be less than FD_SETSIZE. The recommended value of these constants is sixteen.

Each request consists of zero or more operations, in order. Each operation looks like:

```
struct net_security_sockaddr_union {
  struct sockaddr sa;
#if AF_INET
```

```
  struct sockaddr_in sin;
#endif /* AF_INET */
#if AF_INET6
  struct sockaddr_in6 sin6;
#endif /* AF_INET6 */
};

struct net_security_operation {
  uint8_t net_security_operation_type;
  uint8_t net_security_operation_preference;
  uint8_t net_security_operation_barrier;
  uint8_t net_security_operation_reserved;
  /* compiler-inserted pad if 64 bit */
  union {
    struct {
      uint8_t __alloctype;
      uint8_t __hints;
      uint8_t __algid;
      uint8_t __reserved;
    } __forsa;
    union net_security_sockaddr_union __sockaddr;
  } __union;
};
```



```
#define net_security_operation_alloctype __union.__forsa.__alloctype
#define net_security_operation_hints __union.__forsa.__hints
#define net_security_operation_algid __union.__forsa.__algid
#define net_security_operation_sockaddr __union.__sockaddr.sa
```

type — The type of operation to be executed. Defined operations are described in section 2.1.

preference — The preference of the application toward this operation being executed. Defined preferences are described in section 2.2.

barrier — If set to one, indicates that a barrier should be placed after this operation. If set to zero, no barrier is placed there. All other values are reserved.

reserved — MUST be set to zero.

alloctype — For E and A requests, indicates the allocation type requested for SAs obtained for this operation. Defined values for this field are described in section 3.3.

hints — For E and A requests, a set of bit-mapped values that give the policy engine hints as to what algorithm and parameters should be used for this operation. Defined values for this field are described in section 3.5.

algid — For E and A requests, if nonzero, an algorithm identifier that requests that a specific cryptographic algorithm be used. Values for this field are defined in [MMP97]. This MUST be used as a means of last resort only. The use of this field is a privileged operation and subject to system policy; if it is nonzero and the application is not privileged, the system MUST return EPERM when the request is loaded. If an system's policy rejects the use of the algorithm specified in this field, the request SHOULD fail. Applications MUST NOT require this capability for normal operation. Systems MAY universally refuse to honor this field. [cmetz: This field is a concept-breaking blemish, but it's here by popular demand.]

sockaddr — For N requests, if sa_family is nonzero, the destination address of the requested encapsulation. This MAY be an address in a protocol family other than that of the socket. Specification of a destination MAY be a privileged operation, the details of which are specific to a particular system implementation. If sa_family is zero, the destination is the same as the destination address of the inside packet. This field MUST NOT contain addresses for system-local protocol families (e.g., PF_UNIX, PF_LOCAL, PF_ROUTE, and PF_KEY).

net_security_sockaddr_union is defined as being large enough to

hold any sockaddr on the system that can be used with a socket protocol family that supports this API.

3.3. Allocation Types

This API gives applications the ability to request the granularity with which the system shares (or doesn't share) its SAs. The different granularities are called allocation types because they control the SA database's allocation functions. Six allocation types are currently defined:

System: The application requests that SAs allocated to this socket be shared with any other socket on the system.

GID: The application requests that SAs allocated to this socket be shared only with other sockets with the same group ID.

UID: The application requests that SAs allocated to this socket be shared only with other sockets with the same user ID.

PGID: The application requests that SAs allocated to this socket be shared only with other sockets with the same process group (sometimes called session) ID.

Family: The application requests that SAs allocated to this socket be allocated only to this socket and its descendants. Descendants are sockets created by through calls such as accept() as well as those copies of a socket created for child processes.

Socket: The application requests that SAs allocated to this socket be allocated only to this socket. In the case of a passively created stream socket, control of the SAs created for connection setup will be transferred to the child socket returned at accept() time. After that, the listening socket MUST NOT have access to those SAs.

Note that the GID, UID, and PGID MUST for a socket MUST be recorded at the time of socket creation and that stored copy is the GID, UID, and/or PGID used for SA allocation. This means that, if a program changes any of these after a socket has been opened, the ID used for allocation of SAs to a socket does not change. Also note that the actual UID used is the REAL UID.

Application programmers should note that this behavior may not be what they would expect. For example, if an application opens a socket, requests an allocation type of PGID, then calls setpgid(), fork(), and exec()s a new process that also opens a socket and requests an allocation type of PGID, the SAs WILL NOT be shared

between the two sockets.

3.4. Identity Types

This API gives applications the ability to request the type of identity information sent to remote key management. The following identity types are currently defined:

- Address: The application requests that no identity information beyond the addresses present on a SA be specified.
- Prefix: The application requests that additional identity information for the system's prefix be specified. (For IP systems, a prefix is the same thing as a subnet)
- FQDN: The application requests that additional identity information for the system's fully qualified domain name be specified.
- UserFQDN: The application requests that additional identity information in the form of a user's name and a system's fully qualified domain name be specified.

Please note that some combinations of identity type and allocation type may or may not make sense for a given system. For example, most systems will probably not want to allow system or GID allocation with UserFQDN identities. System implementations SHOULD make the allowable combinations a policy control available to system administrators.

More information on identity types may be found in the IP Security DOI specification [Piper97].

3.5. Hints

This API gives applications the ability to give certain hints to the policy engine about its expected security needs. These hints SHOULD affect the selection of specific transforms and algorithms by the policy engine.

There are three parameters that an application can give hints about: the sensitivity of its data, the expected volume of data, and the latency needs of the application. Note that these parameters are not quite independent or dependent. How these parameters affect algorithm selection SHOULD be a policy decision that is configurable by the system administrator.

The following sensitivity levels are currently defined. Note that there are sixteen numeric values currently available but only five named values; these extra intermediate values MAY be used when extra granularity is needed.

```
Unknown:       The application does not know in advance how
               sensitive its data will be.
Lowest:        The application expects its data to have the lowest
               sensitivity of any on the system and requires the
               weakest security.
Low:           The application expects its data to have low
               sensitivity; a weak algorithm is acceptable.
Medium:        The application expects its data to have medium
               sensitivity.
High:          The application expects its data to have high
               sensitivity; a strong algorithm should be used.
Highest:       The application expects its data to have the highest
               sensitivity of any on the system and requires the
               strongest security available. Applications MUST NOT
               use this level unless absolutely necessary.
```

The following volume levels are currently defined:

```
Unknown:       The application does not know in advance what volume
               of data it will communicate.
Low:           The application expects to communicate a low volume
               of data. For example, diagnostic applications like
               ping(8) might use this.
Medium:        The application expects to communicate a moderate
               volume of data.
High:          The application expects to communicate a high volume
               of data. For example, bulk data transfers such as
               FTP might use this.
```

The following latency levels are currently defined:

```
Unknown:       The application does not know in advance what
               latency requirements it has for its data.
Low:           The application expects its data to need low
               latencies. For example, certain real-time traffic
               might need this.
Medium:        The application expects its data to tolerate
               moderate latencies.
High:          The application expects its data to tolerate high
               latencies. For example, bulk data transfers such as
               FTP might use this.
```

Applications SHOULD use the hints field to describe their security needs in abstract properties if possible. This API tries to prevent applications from directly selecting security algorithms and

parameters. This is a deliberate and useful feature, though some may consider this a bug.

3.6. Defined Values

The following values have been defined for various fields. All other values are reserved.

Operation types (net_security_operation_type):

```
/* Authenticate (A) */
#define NET_SECURITY_TYPE_AUTHENTICATE         1
/* Encrypt (E) */
#define NET_SECURITY_TYPE_ENCRYPT              2
/* Encapsulate (N) */
#define NET_SECURITY_TYPE_ENCAPSULATE          3
```

Operation preferences (net_security_operation_preference):

```
/* Default (d) */
#define NET_SECURITY_PREFERENCE_DEFAULT        0
/* Use (u) */
#define NET_SECURITY_PREFERENCE_USE            1
/* Require (r) */
#define NET_SECURITY_PREFERENCE_REQUIRE        2
/* Never (n) */
#define NET_SECURITY_PREFERENCE_NEVER          3
```

Operation SA allocation types (net_security_operation_alloctype):

```
/* System */
#define NET_SECURITY_ALLOCTYPE_SYSTEM          1
/* GID */
#define NET_SECURITY_ALLOCTYPE_GID             2
/* UID */
#define NET_SECURITY_ALLOCTYPE_UID             3
/* PGID */
#define NET_SECURITY_ALLOCTYPE_PGID            4
/* Family */
#define NET_SECURITY_ALLOCTYPE_FAMILY          5
```

```
/* Socket */
#define NET_SECURITY_ALLOCTYPE_SOCKET          6

  Operation SA hints (net_security_operation_hints):

/* Mask for sensitivity hints */
#define NET_SECURITY_HINTS_SENSITIVITY          0x0f
/* Mask for volume hints */
```



```
#define NET_SECURITY_HINTS_VOLUME               0x30
/* Mask for latency hints */
#define NET_SECURITY_HINTS_LATENCY              0xc0

/* Unknown sensitivity */
#define NET_SECURITY_SENSITIVITY_UNKNOWN        0x00
/* Lowest sensitivity */
#define NET_SECURITY_SENSITIVITY_LOWEST         0x01
/* Low sensitivity */
#define NET_SECURITY_SENSITIVITY_LOW            0x04
/* Medium sensitivity */
#define NET_SECURITY_SENSITIVITY_MEDIUM         0x07
/* High sensitivity */
#define NET_SECURITY_SENSITIVITY_HIGH           0x0c
/* Highest sensitivity */
#define NET_SECURITY_SENSITIVITY_HIGHEST        0x0f

/* Unknown volume */
#define NET_SECURITY_SENSITIVITY_UNKNOWN        0x00
/* Low volume */
#define NET_SECURITY_VOLUME_LOW                 0x10
/* Medium volume */
#define NET_SECURITY_VOLUME_MEDIUM              0x20
/* High volume */
#define NET_SECURITY_VOLUME_HIGH                0x30

/* Unknown latency */
#define NET_SECURITY_LATENCY_UNKNOWN             0x00
/* Low latency */
#define NET_SECURITY_LATENCY_LOW                0x40
/* Medium latency */
#define NET_SECURITY_LATENCY_MEDIUM             0x80
/* High latency */
```

```
#define NET_SECURITY_LATENCY_HIGH                0xc0
```

Identity types:

```
/* Address */
#define NET_SECURITY_IDENTTYPE_ADDRESS          1
/* Prefix */
#define NET_SECURITY_IDENTTYPE_PREFIX           2
/* FQDN */
#define NET_SECURITY_IDENTTYPE_FQDN             3
/* USERFQDN */
#define NET_SECURITY_IDENTTYPE_USERFQDN         4
```

Identity size:

```
/* Maximum buffer needed to hold an identity string */
```



```
#define NET_SECURITY_IDENTITY_MAX               1024
```

Receive identity types (for net_security_receiveident() et al.):

```
/* Receive local identity information */
#define NET_SECURITY_RECEIVEIDENT_LOCAL         1
/* Receive remote identity information */
#define NET_SECURITY_RECEIVEIDENT_REMOTE        2
```

3.7. Function Calls

This API specifies several functions that will typically be implemented as a simple setsockopt() or getsockopt() call to the operating system. However, systems MAY choose to implement this using a different low-level interface.

Other functions provide user interface or helper functions that SHOULD be implemented completely in user space.

Unless otherwise specified, all functions return zero on success, negative one on failure, and return any further indication of the reason for failure in the global variable errno.

3.7.1. Set and Get Request

```
int net_security_setrequest(int s, int number,
```

```
                              struct net_security_operation *ops, int numops);
  int net_security_getrequest(int s, int number,
                             struct net_security_operation *ops, int *numops);
```

These calls set and get, respectively, the application's current request for network security services from the kernel.

The net_security_setrequest() call requests that the system replace the currently registered network security request with the specified number for the socket s with the new request pointed to by ops of numops operations. This function MUST verify that the request is correctly formed and that all values are within range; that verification SHOULD be done in the kernel. If this is not the case, the function MUST fail with errno=EINVAL. If the socket is a stream socket that is currently latched, this call MUST fail with errno=EPERM. If this call fails for any reason, the loaded request MUST remain unchanged.

System implementors must be careful about attempting to check the request against system policy when net_security_setrequest() is called. Values that might be acceptable to the policy at registration time might not be acceptable when it's time to send or

receive data, and the converse is also true. Therefore, the system MUST NOT check the request against any part of system policy that could change during the life of the socket.

The net_security_getrequest() call returns the currently registered request for a socket. This call is useful because it allows applications that receive an existing socket from a parent, such as children of inetd(8), to determine what security properties were originally requested. Note that numops MUST be initialized to the maximum number of operations that may be stored in the buffer ops before this call and the actual number of operations returned is stored in numops upon success.

Both of these calls SHOULD be implemented as much as possible in kernel space and SHOULD be implemented as a setsockopt() and getsockopt() call, respectively.

3.7.2. Request Bitmap Functions

```
int net_security_activerequests(int s, fd_set *which);
int net_security_inputrequests(int s, fd_set *which);
```

The net_security_activerequests() function returns a bitmap that identifies which request numbers point to a request that has been loaded into the kernel. This can be used to determine if a request number is in use or to find an unused request number. A set bit indicates that the request number corresponding to the bit number is in use by a loaded request.

The net_security_inputrequests() function loads into the kernel a bitmap that identifies which request numbers identify requests that the application would like to attempt to match against an incoming packet if the request number has been set to -1. A set bit indicates that the request number corresponding to the bit number corresponds to a request that the application would like to attempt to match. The kernel SHOULD silently ignore set bits in this bitmap if they point to unused request numbers. The default for this bitmap MUST be all set bits.

Both of these calls SHOULD be implemented as much as possible in kernel space and SHOULD be implemented as a setsockopt() and getsockopt() call, respectively.

3.7.3. Request Selection Functions

```
int net_security_setnum(int s, int num);
int net_security_getnum(int s, int *num);
```


The net_security_setnum() call sets the currently selected request number for I/O operations. If the socket is a stream socket and a request has been latched in, this call MUST fail with errno=EPERM. If a value between and including zero and (NET_SECURITY_REQUEST_MAX-1) is specified, that request will be used for all inputs or outputs unless otherwise specified in a control message. Implementations MUST NOT return an error if a valid request number is specified for which a request has not yet been set.

If a value of negative one is specified, output operations will fail. Input operations will attempt to match requests starting at zero if the corresponding bit in the input mask has been set and will loop through all loaded requests until one matches or all eligible requests have been tried. If the socket is a stream socket, the index of the succeeding request will then be latched in and will replace the value specified in this request. Thus, subsequent outputs will succeed.

The net_security_getnum() call gets the currently selected request number for I/O operations. This is useful both for sockets that are inherited from another process and to determine which request got latched on a stream socket.

Note that, for datagram and raw sockets, a net_security_setnum(num=-1) followed by a successful input operation will still cause net_security_getnum to return -1; datagram and raw-socket applications that want to know which request was matched MUST use control messages to receive that information. For stream sockets, this function will return the number of the matched request because the original setting of negative one will have been replaced by the matched request number as part of the latching process.

Both of these calls SHOULD be implemented as much as possible in kernel space and SHOULD be implemented as a setsockopt() and getsockopt() call, respectively.

3.7.4. Request Selection Control Message Functions

```
int net_security_receivenum(int s, int onoff);
int net_security_cmsgtonum(void *cmsg, int cmsglen, int *num);
int net_security_numtocmsg(int num, void *cmsg, int *cmsglen,
                           int maxlen);
```

If the socket is a datagram or raw socket, the application can, on a per-packet basis, set the recipe to use for a particular packet or get the recipe matched for a particular packet. This is done using the sendmsg() and recvmsg() functions' control message facility and these functions. This can be used, for example, to "reflect" security

properties. Note that all of these operations MUST fail with errno=EPERM for stream sockets.

Whether or not the request number is returned for input operations must be selected in advance by the application. The net_security_receivenum() function is used to turn receipt on and off. A value of one for onoff turns receipt on; a value of zero for onoff turns receipt off. Implementations MUST return EINVAL for any other value of onoff.

The net_security_cmsgtonum() function is used to parse a control message pointed to by cmsg of length cmsglen and return the message recipe number in num. If no received message number information is available, this function MUST return an error.

The net_security_numtocmsg() function is used to add a request to use the recipe number num to the control message cmsg with a current length of cmsglen and a total allowable length of maxlen. If successful, the current length is updated.

The net_security_receivenum() function SHOULD be implemented as much as possible in kernel space and SHOULD be implemented as a setsockopt() call. The net_security_cmsgtonum() and net_security_numtocmsg() calls SHOULD be implemented entirely in user space.

Note that calling sendmsg() and recvmsg() with an appropriately constructed message header is the responsibility of the application.

3.7.5. Identity Selection Functions

```
int net_security_setlocalident(int s, int identtype, char *identstr);
int net_security_getlocalident(int s, int *identtype, char *identstr,
                               int maxlen);
int net_security_setremoteident(int s, int identtype, char *identstr);
int net_security_getremoteident(int s, int *identtype, char *identstr,
                                int maxlen);
```

The net_security_setlocalident() and net_security_setremoteident calls set the local and remote identities for I/O operations. If the socket is a stream socket that has been latched, these calls MUST fail with errno=EPERM.

The net_security_setlocalident() function is used to select the local identity used for security associations allocated to the socket s. If identstr is NULL, the actual identity string used will be generated by the system based on the application's process state and the value of type. If identstr is not NULL, the identity string in identstr SHOULD be passed to key management for negotiation of security associations. This function SHOULD check to make sure that identstr has a content that is valid for the given identtype and that its content is appropriate for the application's process state. For example, when using a USER_FQDN identity certificate, it is usually an error for an application to specify a certificate where the local username resolves to a user ID that is not one of those available to the process. Systems MUST take care NOT to trust the application or any part of this function that might reside in user space or identity spoofing attacks may result.

The net_security_setremoteident() function is used to select the remote identity used for security associations allocated to the socket s. If identstr is NULL, any identity of the specified type will be accepted. If identstr is not NULL, the identity string in identstr SHOULD be passed to key management for negotiation of security associations. Security associations with an identity string other than the one specified MUST NOT be allocated to the socket s. Note carefully that use of this function without a priori knowledge of exactly what identity information a remote system will send will result in an application being unable to communicate. Most applications SHOULD NOT use this function at all and SHOULD instead leave this decision making completely to the policy engine.

The net_security_getlocalident() function is used to retrieve the local identity used for security associations allocated to the socket s. It is only valid for stream sockets; if used on a datagram or raw socket, this function MUST fail with errno=EPERM. The innermost local identity latched into the socket state is returned. If no identity information is present (for example, because the socket is not yet connected or because that operation didn't result in the allocation of security associations), this function MUST fail with errno=ESRCH. If the local identity string used is longer than maxlen, this function must fail with errno=ENOSPC.

The net_security_getremoteident() function is used to retrieve the remote identity used for security associations allocated to the socket s. It is only valid for stream sockets; if used on a datagram or raw socket, this function MUST fail with errno=EPERM. The innermost remote identity latched into the socket state is returned. If no identity information is present (for example, because the socket is not yet connected or because that operation didn't result in the allocation of security associations), this function MUST fail with errno=ESRCH. If the local identity string used is longer than maxlen, this function must fail with errno=ENOSPC.

All four of these calls SHOULD be implemented as much as possible in kernel space. The first two SHOULD be implemented as setsockopt()

calls; the second two SHOULD be implemented as getsockopt() calls.

3.7.6. Identity Selection Control Message Functions

```
int net_security_receiveident(int s, int which);
int net_security_cmsgtolocalident(void *cmsg, int cmsglen,
                          int *identtype, char *identstr, int maxlen);
int net_security_localidenttocmsg(int identtype, char *identstr,
                                void *cmsg, int *cmsglen, int maxlen);
int net_security_cmsgtoremoteident(void *cmsg, int cmsglen,
                          int *identtype, char *identstr, int maxlen);
int net_security_remoteidenttocmsg(int identtype, char *identstr,
                                void *cmsg, int *cmsglen, int maxlen);
```

If the socket is a datagram or raw socket, the application can, on a per-packet basis, set the identities to use for a particular packet or get the identities used for a particular packet. This is done using the sendmsg and recvmsg functions' control message facility and these functions. This can be used, for example, to "reflect" security properties. Note that all of these operations MUST fail with errno=EPERM for stream sockets.

Whether or not identity information is returned for input operations must be selected in advance by the application. The net_security_receiveident function is used to select which, if either, identities are returned for received packets. The argument which is a bitmap field; if the bit with value NET_SECURITY_RECEIVEIDENT_LOCAL is set, local identities are to be received, and, if the bit with value NET_SECURITY_RECEIVEIDENT_REMOTE is set, remote identities are to be received. Implementations MUST return EINVAL if any other bits are set. Applications SHOULD not have receipt of identities enabled unless they are really going to use that information because the processing involved in making that information available to the application might be expensive.

The net_security_cmsgtolocalident() and net_security_cmsgtoremoteident() functions are used to parse a control message pointed to by cmsg of length cmsglen and return the identity type in identtype and the identity string in identstr. If the identity string is longer than maxlen or no received message number is available, this function MUST return an error.

The net_security_localidenttocmsg() and net_security_remoteidenttocmsg() functions are used to add a request to use the identity type identtype and, if not NULL, the identity string identstr to the control message cmsg with a current length of cmsglen and a total allowable length of maxlen. If successful, the current length is updated.

The net_security_receiveident() function SHOULD be implemented as much as possible in kernel space and SHOULD be implemented as a setsockopt() call. The other calls SHOULD be implemented entirely in user space.

Note that calling sendmsg() and recvmsg() with an appropriately constructed message header is the responsibility of the application.

3.7.7. String Conversion Functions

```
int net_security_strtorequest(char *str,
                      struct net_security_operation *ops, int *numops);
int net_security_requesttostr(struct net_security_operation *ops,
                                   int numops, char *str, int maxlen);
```

The most common use of this API is intended to be where a user specifies a string parameter to the application that represents the user's requested security properties. These functions convert such a string to a request that can be passed to a kernel and vice versa.

The net_security_strtorequest() function converts the string str to a request suitable for passing to net_security_setrequest(). The request is returned in the buffer pointed to by ops. Note that numops MUST be initialized to the maximum number of operations that may be stored in the buffer ops before this call and the actual number of operations returned is stored in numops upon success.

The net_security_requesttostr() function converts the request ops of length numops to a string suitable for printing. The string is returned in the buffer pointed to by str. If the string would be longer than maxlen, this function MUST return an error.

These functions SHOULD be implemented in user space.

3.7.8. EPOLICY

This API defines a new errno value, EPOLICY. This value shall have the string definition (for functions like strerror()) of "Operation failed by policy". Output operations that fail as a result of a request and/or the system policy (for example, if require or never preferences are involved) MUST return this value. Input operations that result in a condition where the system would never receive packets for a socket (due to the interaction between the system's policy and the application's request) MUST fail and return this value.

[cmetz: I know that I should avoid defining a new errno value because it's painful for people, but I think that this is an error condition

that needs to be handled separately.]

3.7.9. Blocking Behavior

System implementations MUST make security operations either blocking or non-blocking depending on the application's I/O style request. That is, if the application has requested non-blocking I/O (e.g., via fcntl(F_SETFL, ...)), then security operations MUST also be non- blocking, and vice versa. For blocking I/O, the application MUST block while key management operations are taking place and while security processing (including policy processing and cryptographic operations) is performed. For non-blocking I/O, the application MUST be allowed to be execute while key management operations are taking place and SHOULD be allowed to execute while security processing (including policy processing and cryptographic operations) is performed (within the constraints of a time-slicing system, if appropriate). In the case of use and default preference levels, if SA negotiation is in progress and the system's policy configuration would allow the packet to be transmitted without security processing (e.g., as cleartext), the system MAY do so until a SA has been successfully negotiated.

4. Discussion

Applications that use the loopback interface SHOULD request that security services NOT be provided for that communication (i.e., preference=never). An application has no choice but to trust that its OS is doing the right thing. Therefore, encryption and authentication of data over the loopback interface is usually nothing more than a waste of system resources. There are some cases where it's useful to have the system provide security services on loopback traffic, but implementations typically blindly request encryption and authentication without checking for loopback and end up just wasting cycles. Systems SHOULD have policy rules that aggressively prevent applications from actually doing encryption and authentication over loopback.

[cmetz: more stuff will go here]

Future Work

This document needs lots of examples and discussion of border cases. This will be coming in a future revision.

A companion document describing one possible design and implementation of a policy engine is in progress.

Acknowledgments

Randall Atkinson, Ron Lee, Daniel L. McDonald, and Chris Winters provided useful feedback on earlier versions of this document.

Parts of this work was done at the Center for High Assurance Computer Systems at the U.S. Naval Research Laboratory. This work was sponsored by the Information Security Program Office (PMW-161), U.S. Space and Naval Warfare Systems Command (SPAWAR) and the Computing Systems Technology Office, Defense Advanced Research Projects Agency (DARPA/CSTO). The author and his co-workers really appreciate their sponsorship of NRL's network security efforts and their continued support of IPsec development. Without that support, this document, among many others, would not exist.

The "CONFORMANCE and COMPLIANCE" wording was taken from [MSST97].

The author has created a mailing list for discussion of this specification and implementations of it. If you would like to be added to this list, send a note to <net-security-api-request@inner.net>.

References

[Atk95a] Randall J. Atkinson, "IP Security Architecture", RFC 1825, August 1995.

[Atk95b] Randall J. Atkinson, "IP Authentication Header", RFC 1826, August 1995.

[Atk95c] Randall J. Atkinson, "IP Encapsulating Security Payload", RFC 1827, August 1995.

[MMP97] D. L. McDonald, C. W. Metz, B. G. Phan, "PF_KEY Key Management API, Version 2", Internet Draft, July 1997.

[MSST97] Douglas Maughan, Mark Schertler, Mark Schneider, Jeff Turner, "Internet Security Association and Key Management Protocol (ISAKMP)", Internet Draft, February 1997.

[Piper97] Derrel Piper, "The Internet IP Security Domain of Interpretation for ISAKMP", Internet Draft, February 1997.

Disclaimer

The views and specification here are those of the author and are not necessarily those of his employer. His employer has not passed judgment on the merits, if any, of this work. The author and his

employer specifically disclaim responsibility for any problems arising from correct or incorrect implementation or use of this specification.

Author's Address

Craig Metz
The Inner Net
Box 10314-1933
Blacksburg, VA 24062-0314
(DSN) 354-8590
cmetz@inner.net

Revision History

01 Replaced all setsockopt()/getsockopt() calls with function calls expected to be front-ends; this follows POSIX p1003.1g's lead and might also be easier to implement on some systems. Replaced identity specification on each E/A request with a selection global to all requests. Split function calls up into separate sections. Removed magic cookie. Removed the "request" as a block of "recipes" and renamed "recipe" to "request"; made other changes that follow from this. Updated system organization graph.

00 Initial draft.