IPSec/PHIL (Packet Header Information List): Design, Implementation, and Evaluation

Chien-Long Wu, NC State University, Raleigh, NC
S. Felix Wu, University of California, Davis, CA
Ravindar Narayan, Cosine Communication, Red Wood City, CA

Abstract

For most TCP/UDP/IP applications, when a packet or a message arrives, usually only the payload portion of the original packet can be obtained by the application. For instance, if a packet has been delivered through some IPSec tunnels along the route path, then the application, in general, will not know exactly which tunnels have been used to deliver this particular packet. The IPSec/PHIL (Packet Header Information List) interface has been designed and implemented such that an "authorized" application is able ***to know*** *which set of IPSec tunnels has been used to deliver a particular incoming packet. Furthermore, IPSec/PHIL enables the* ***controlability*** *over which set of IPSec tunnels will be used to send a particular outgoing packet. IPSec/PHIL is a key component in the DECIDUOUS decentralized source tracing system to correlate the IPSec information with intrusion detection results. Other IPSec/PHIL applications we have built include a SNMPv3 security module using IPSec as well as a IPSec tunnel switching router.*

1 Introduction

IP security (IPSec) protocol suite [1, 2, 3, 4] is a series of guidelines for the protection of Internet Protocol (IP) communications. It provides ways for securing private information transmitted over public networks. The currently available IPSec-based applications in the market are predominantly Virtual Private Networks (VPNs). VPNs provide Network-to-Network security by setting up SAs (Security Associations) in the tunnel mode between Gateways of the networks, and these tunnels secure the aggregated data flowing from one policy domain to another through IPSec gateways.

For most TCP/UDP/IP applications, when a packet or a message arrives, usually only the payload portion of the original packet can be obtained by the application. If a packet has been delivered through some IPSec tunnels, then the application, in general, will not know exactly which tunnels have been used to deliver this particular packet. For instance, an intrusion source tracing system (such as DECIDUOUS [??]) might be very interested in analyzing not only the payload but also which particular IPSec tunnels have been used to deliver these attack packets with obviously spoofed source IP addresses.

For application-layer protocols such as SNMP and LDAP, it is usually not natural and feasible to use IPSec to secure the application-layer traffic, and thus a separate security mechanism is needed. In this paper, we will show that, with a simple extension of the socket API, the security mechanisms and capabilities of IPSec can support the security requirements of some applications. We called this new interface: IPSec/PHIL (Packet Header Information List [5]) API.

A third issue is regarding the support of end-to-end security using IPSec, while it is impossible to directly build a IPSec security association from the source to the destination. For instance, in an inter-domain environment, it might not be always possible to negotiate directly between two IP nodes belonging to two different domains. Things get more tricky if an intermediate gateway will perform network address translation (NAT). We will show later that how to utilize the PHIL API to support "packet switching" among a set of IPSec tunnels such that it is possible to use a set of tunnels collaboratively (an IPSec tunnel path, more specifically) to secure the information end-to-end.

2 Motivation

In the FreeSWAN implementation [7] of IPSec:

- At the time of processing the **<u>in-bound IPSec traffic</u>**, all the IPSec headers are discarded at the IP layer; and when the applications do receive data, the data is devoid of all IPSec headers, thus there is no way of knowing whether the incoming packets were secured. If the packets were secured, then it is also hard to figure out what level of security was afforded and which end host or Security Gateways provided the security.
- Also, there is no way for user applications to control the **<u>out-bound IPSec traffic</u>** through a specific SA (security association). More specifically, since we are unable to bind an SA to a particular socket port in the application layer, we can not support end-to-end application-layer security using IPSec.

Based on our observation and experience with IPSec, we believe that IPSec's capabilities can be greatly extended if we have a good interface to access the security services provided in the IP layer. Naturally, we would like to have the following two capabilities:

- For incoming traffic it provides an API such that the application developers are able to extract security information such as the security afforded to a particular segment of data received at the application layer from the kernel.
- For the outgoing traffic, PHIL-API provides the functionality to interact with the kernel's Security Association database (SAdB) and Security Policy database (SPdB) to query information about the existing SAs and the security level afforded to their outgoing data. It should also provide a way for the outgoing process to be able to override the default security policy.

3 Specification of Packet Header Information List (PHIL) API

The key feature in PHIL-API is the "PHIL" information, which is a "list" data structure containing the IPSec related information. In regular protocol stack processing, all header information about IPSec has been stripped out before the payload being passed to the transport and application layers. However, in PHIL, extra IPSec information will be attached to the payload all the way up or down. The architecture of PHIL and its relationship with the OS kernel is depicted in the following figure. In this section, we first introduce the set of PHIL-API calls for accessing the IPSec services provided by the kernel. In the subsequent sections, we will then show how they are implemented and applied.

3.1 PHIL-API for Incoming Traffic

- A TCP or UDP socket opened with a socket system call should first be enabled (or maybe later disabled) to receive the PHIL information along with the application data. The following two functions are designed to control the PHIL functions:

  int phil_enable (int sockfd, int mode)
  int phil disable (int sockfd)

This function call prepares a socket to receive PHIL information and the mode parameter can either be EXHAUSTIVE or NON_EXHAUSTIVE. If the receive mode is set to NON_EXHAUSTIVE mode then the PHIL information consists only of the SPI value associated with each data segment that was received. Otherwise, in the EXHAUSTIVE mode, the PHIL information consists of the entire set of security parameters associated with the packet (explained in the ***phil_recvfrom()*** call in detail later). This function must be called before any other PHIL-API calls are used in the input mode. This call can be used at any time on a socket that is open and performing normal send/receive operations. **After PHIL enabling a socket an application**

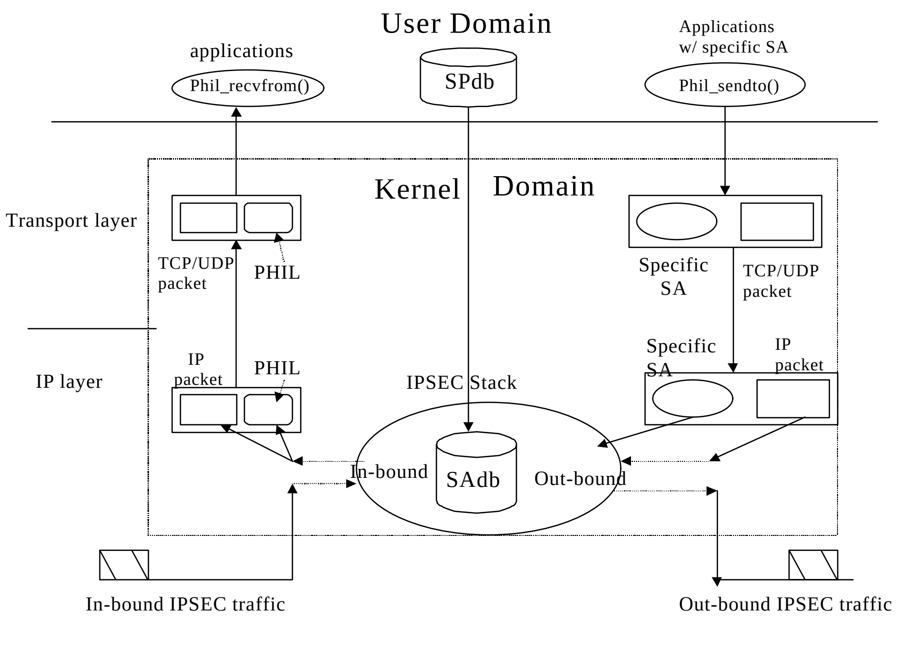
can continue to use normal socket API calls on the socket. In case of UDP server applications, a single enabled call on the socket will facilitate the PHIL information regarding packets from various sources. In case of TCP concurrent server applications, each child socket that is spawned to handle a new connection should be PHIL enabled so that PHIL information for each connection is properly received.

- For TCP applications:
  int phil_accept (parameters to accept( ), char *phil_buf, int phil_len)

In addition to the return values of the corresponding normal API call ***accept( ),*** this call returns ***phil_buf***, which is a character buffer containing the PHIL information. The ***phil_accept( )*** call is for TCP concurrent mode. The parent socket should provide policy information about which peers can be trusted based on the PHIL information. The server can then decide whether or not it should accept the new request.

- For receiving data:
  int phil_recvfrom (parameters to recvfrom( ), char *phil_buf, int phil_len, int *dsegs)

In addition to the return values of the corresponding normal call ***recvfrom( )***, this call returns ***phil_buf***, which is a character buffer that contains the PHIL information, and ***dsegs***, which is the number of TCP data segments constituting the total data bytes being read from this call. A ***phil_recvfrom( )*** call is intended to be used to retrieve the data plus the PHIL information for both UDP and TCP applications. The ***phil_buf*** points to one of the two PHIL information structures depending on the receive mode that was set.



3.2 PHIL-API for Outgoing Traffic

- The outgoing data can be directly linked to one or more SA's by binding the ***spi*** values of SA's (Security Associations) to the socket.
  int phil bind (int sockfd, unsigned long *spi array, int size)
  int phil unbind (int sockfd)

The bind call merely records a possible set of SPI's that could be used by this socket in its sessions, the actual and specific value for each send operation can be different and should be specified at the time of a send call, also specifying a set of SPI's through the ***phil_bind( )*** call does not mandate a socket to send its data securely all the time.

- The following call is a modification to the data output call.

  int phil_sendto (parameters to sendto( ), long *spi_arr , int size)

The ***spi_arr*** array describes the SPI value(s) of our preference in sending the data (in the case where we have a choice of sending the data over several possible SAs). If the application does not know the set of the possible SA's, it can query the SAdB (Security Association database) for the SPI values through the query ***spi( )*** call described later. The value of the SPI's should be a subset or equal to the SPI values that were bound to the socket using the ***phil_bind( )*** call earlier. Through the ***phil_sendto*** call it is possible to send a stream of data bytes from a single application process over different SAs, which provides different levels and features of security for different data types. It should be emphasized, though, that the application's preference of sending its data over an SA will be honored only if it conforms to the SPdB.

3.3 Miscellaneous Calls

- An application sometimes might want to know the availability of an SA and their SPI values before they send the data:

  int phil_qspi (struct sockaddr* src, struct sockaddr * dst, char * buf, int size, int *num)

When no SA's exist to protect the data between the given source and destination address, an error is returned. In case of VPN, the source IP address is most likely the IP address of the security gateway on which the application is running. If there are one or more SAs that match the source and destination IP addresses then they are returned through the SPI structure as a linked list.

- Based on the SPI values, the applications can further query the SAdB for the source of authentication.

  int phil qsadb (unsigned long spi, struct sockaddr *dst, char *buf, int size)

This function is used to query the SAdB for detailed information about the SA.

4 Implementation

4.1 Linux FreeS/wan IPSec implementation

Linux FreeS/wan is an IPSec implementation for Linux; and it is freely available from **http://www.freeswan.org/**. Our PHIL/PHIL-API implementation is on Linux kernel version 2.0.36 with FreeS/wan version 1.0.

4.2 PHIL-API

For the incoming traffic, the FreeS/wan implementation is BITS (Bump In The Stack) implementation, and therefore the security headers are stripped before the packet is handed over to the IP layer. Therefore, we create the PHIL information at the time of verifying and decrypting a secured packet in the BITS implementation. Since the memory management in Linux is handled within the "***sk_buff***" structure, we modified the structure to include the PHIL related information. As the same ***sk_buff*** traverses the entire network stack up to the socket layer, we copy back the PHIL information from the ***sk_buff*** to the user buffer at the same time when the user socket data is being copied from the ***sk_buff*** into the user buffer. With this motive, a pointer to the PHIL information structure called the ***spi_info*** is appended to the ***sk_buff*** structure. As the number of

SA's between a pair of nodes is not constant, each SA memory element is dynamically allocated at the time of IPSec header processing. The "***spi_info***" pointer in the ***sk_buff*** structure is the starting point for the singly linked list of these memory elements called "***spi_structs***", and the order of these ***spi_structs*** determines the order of applying SA's to the packet. This approach optimizes the memory allocation and does not burden the ***sk_buff*** structure unnecessarily when no SA's exists. The following is the definition of ***spi_structure***:

```
typedef struct spi_struct {
      unsigned long  spi;
      unsigned long  dst_addr;
      unsigned long  src_addr;
      unsigned short dst_port;
      unsigned short src_port;
      unsigned short xprot;      /* TCP or UDP */
      unsigned int   data_len;
      unsigned int   sec_proto;
      spi_struct    *next_ssp;
};
```

4.3 In-bound processing in Linux FreeS/wan IPSec

At the data-link layer, after receiving an Ethernet packet, the network card triggers ***ei_interrupt( )*** which then transfers control to ***ei_receive( ),***a function allocating the ***sk_buff*** structure and initializing it with the received data (and here is where the journey begins for the ***sk_buff*** structure). At this time, the function ***netif_rx( )*** is called with the packet as the argument and it adds the packet to the input queue. All these activities are done within the interrupt call ***ei_interrupt( ).*** Before returning from this interrupt call, a bit in the network implementation's bottom half mask, called "***bh_mask***", is set. The kernel checks the ***bh_mask*** regularly when there are no system calls and interrupts to handle. If any of the ***bh_mask*** bits is set then the service function corresponding to the set bit is called through the function ***do_bottom_half( )***, and the service function under this case is the ***net_bh( )*** function. The ***net_bh( )*** function de_queues the packets from the input queue and passes it to the ***ip_rcv( )*** function which de-multiplexes the packet to one of the transport layer's receiving functions. In the case of IPSec AH and ESP processing, the packet is *queued again* into the system input queue (and to be picked up by the ***net_bh( )*** function again). The transport layer receiving function will enqueue the packet into the appropriate sockets. For an UDP server socket, we have only one receiving queue, whereas, in case of TCP server, every client will have a dedicated receiving queue at the server socket.

When the user issues a ***recvfrom( )*** system call to the socket, the handling function in the kernel locates the appropriate socket and dequeues the packet from the queue to copy the data to the user buffer and return. If the ***phil_recvfrom( )*** is used instead, then the PHIL associated with the data is also passed up to the application in addition to the user data. The PHIL information associated with a ***sk_buff*** in the kernel will be destroyed at the time of releasing that particular ***sk_buff***.

4.4 Out-bound processing in Linux FreeS/wan IPSec

The outgoing packets in the Linux FreeS/wan IPSec are secured only after they cross the IP layer because of the BITS implementation. Thus, each IP packet is secured by the <u>device transmit function</u> before giving the packet to the actual device itself, Figure 3 illustrates this functional flow. Every ***sk_buff*** used in the outgoing packet path contains a back pointer to the socket structure that owns the ***sk_buff*** (and hence the data contained in the ***sk_buff***). Therefore, even at the device transmit level, the pointer to the socket is available to cross check with the SPI values bounded through the ***phil_bind*** call. And, the phil_arr parameter in phil_sendto will choose the SA's for the current packet. When this packet arrives at the receiver side, the SPI values in the

IPSec packet headers are matched first with the existing SA's. If each SPI (and hence each SA) is associated with a data sensitivity level, an application can received different levels of security to cater to its data's multi-sensitivities. In other words, this application can secure every single byte of its data using a different SA.

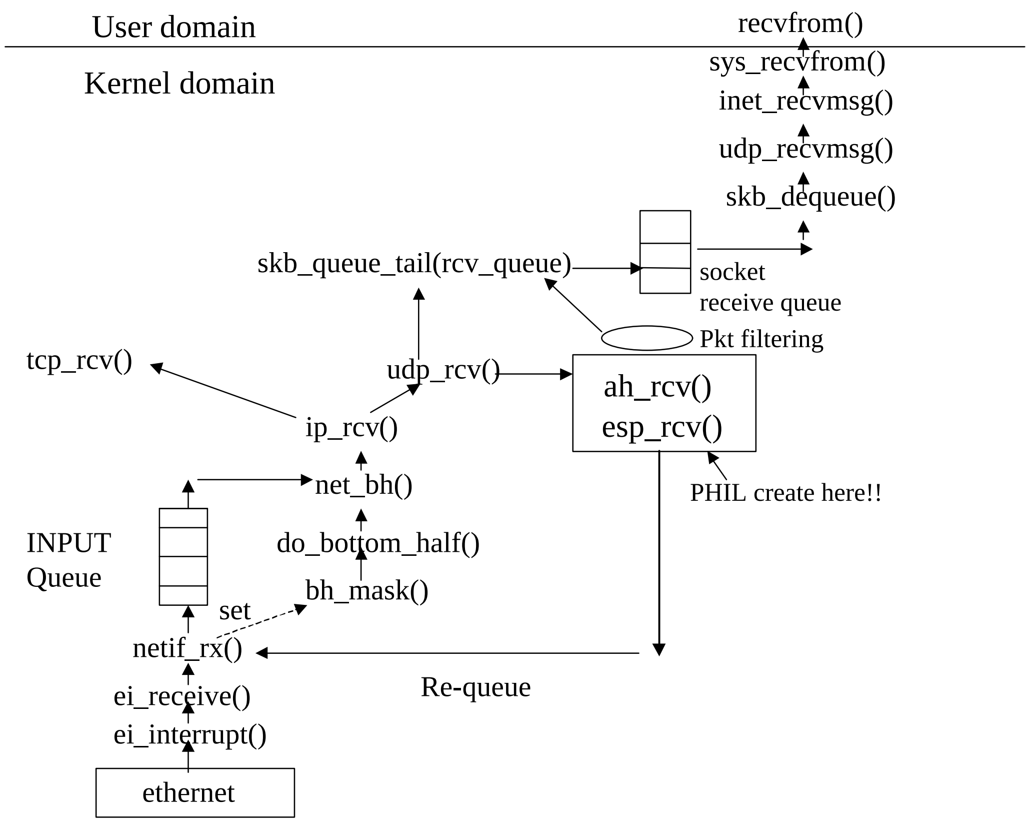


Figure 2: Incoming packet process in Linux kernel (in UDP case)

5 Application

5.1 *PHIL Switching*

Traditionally, an IP router will forward packets solely based on their destination addresses. In the DECIDUOUS project [5,6], in order to support “inter-domain collaboration,” a router needs to switch the packets based on the “incoming” IPSec tunnels. Due to the space limitation, we will discuss very briefly how this new feature is used in DECIDUOUS. The functions of PHIL-switching router can be summarized as:

- Any incoming IPSec traffic, if matched any entry of the PHIL-switching table, will be forwarded to a specific security path.
- If the incoming traffic is non-IPSec, it will be processed as normal IP traffic.
- A selected set of inbound SA’s can be aggregated into one outbound SA or dropped.

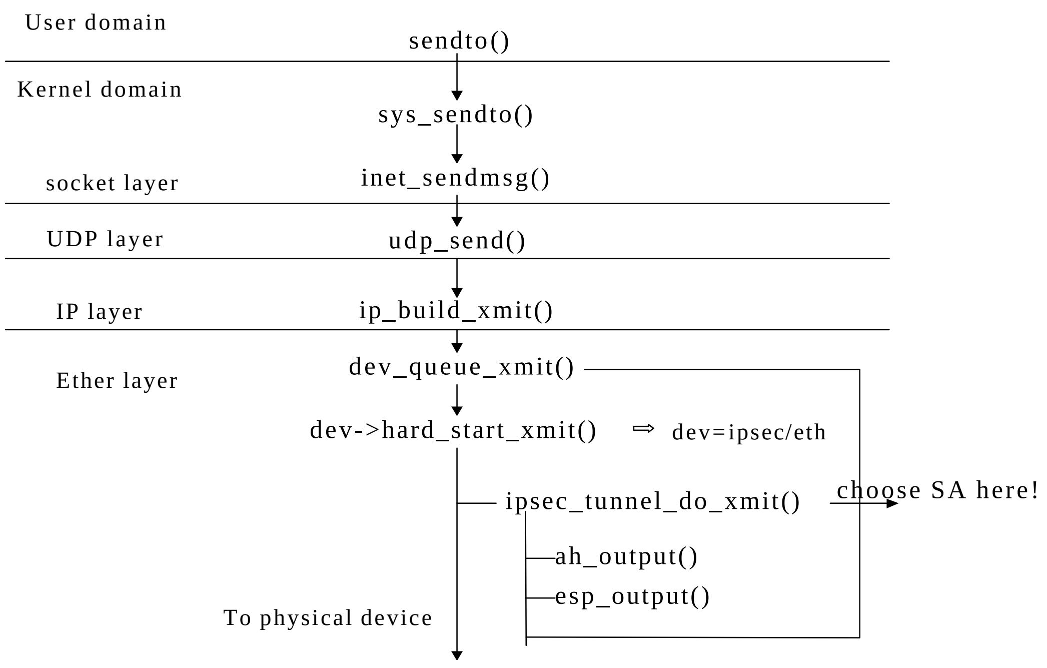


Figure 3: Outgoing packet process in Linux FreeS/wan IPSec.

5.1.1 PHIL-Switching controller

The PHIL Switching controller provides an interface for users to add, delete or flush PHIL switching table. In our implementation, the controller has two different interfaces: one is a client-server model using UDP, the other is the SNMP MIB model. In the latter case, PHIL-switching is under the control of SNMP agent through the PHIL-switching MIB. In either case, the user can "read, add, delete" and "flush" the PHIL-switching table.

5.1.2 User-Level Switching Entity

For any incoming packets with header fields such as [SPI, security protocol, source and destination addresses, protocol, source and destination ports], the switching entity looks up the PHIL switching table. Then, it will "switch" the incoming packets into different tunnels using specified SPI according to the switching table entry. To realize the concept of PHIL-Switching in the user level, we use "*divert sockets*" to intercept IP packets from kernel to the user-level switching process. After the switching table look up, the intercepted packet will be forwarded using the ***phil_sendto( )*** with a specified SPI/SA.

5.1.3 Deciduous Collaboration

In an intra-domain environment, it is easy to establish SA among routers. It is in general not possible to build up SA directly among any pair of routers in different domains. With the realization of PHIL-switching, DECIDUOUS can collaborate with a security gateway in another domain to establish a IPSEC SA tunnel path, which emulates a direct SA across multiple domains.

We assume that SA can only be established between the border security gateway of collaborating domains. In Figure 6, six SA's (A1, A2, B1, B2, C1, C2) have been established between victim and local border router, between remote and local border routers, and within the remote domain.

Now if IDS (Intrusion Detection System) detects attacks, it will report the detected attacks to local DECIDUOUS process. In the report, it will show that the attacks have been launched through SA C2. With the local PHIL-Switching table, we can tellthat the attacks are indeed from SA B2. And, finally, the remote domain will be notified and it can further track down the source by correlating SA B2 with SA A2.

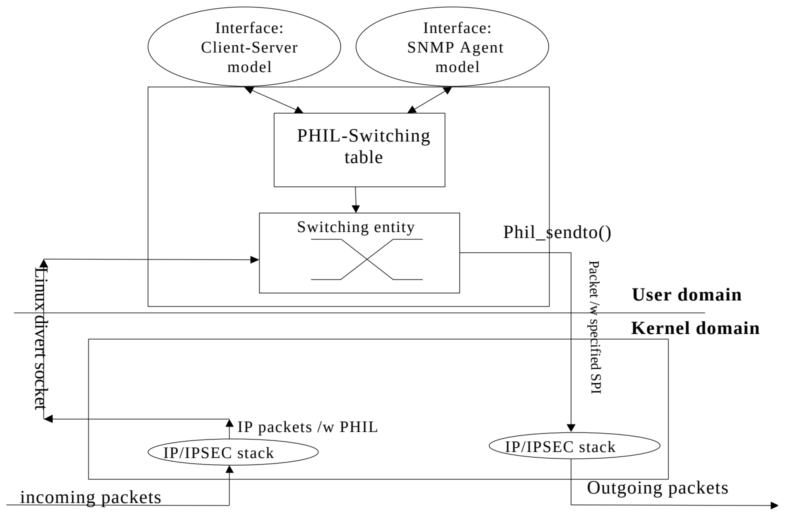


Figure 5: PHIL-switching implementation.

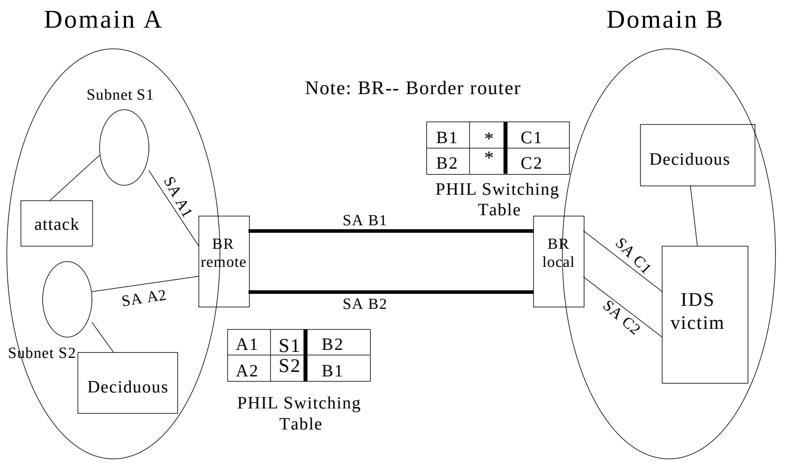


Figure 6: DECIDUOUS Collaboration with PHIL-Switching

5.2 SNMP over IPSEC

The earlier versions of SNMP (SNMPv1 and SNMPv2) [11,12] use the **community** feature for a simple and unsecured password-based authentication. To improve the security concern, therefore, SNMPv3 introduces the concepts of ***snmpEngineId*** and ***securityName***. ***snmpEngineId*** uniquely identifies an SNMP engine that provides services for sending and receiving messages, authenticating and encrypting messages, and controlling access to managed objects. ***securityName*** is a human readable string representing an individual on whose behalf the services are provided or processed. Each ***securityName*** is associated (or configured) with a ***securityLevel*** parameter which is stored in the ***Local Configuration Database (LCD)***. When a user issues a command or requests information, LCD is queried to determine the security requirements for the given ***securityName*** and ***snmpEngineID***. If the securityLevel specifies that the message is to be authenticated, then the message is authenticated according to the user's authentication protocol. Privacy and timeliness modules are called depending on the ***securityLevel***.

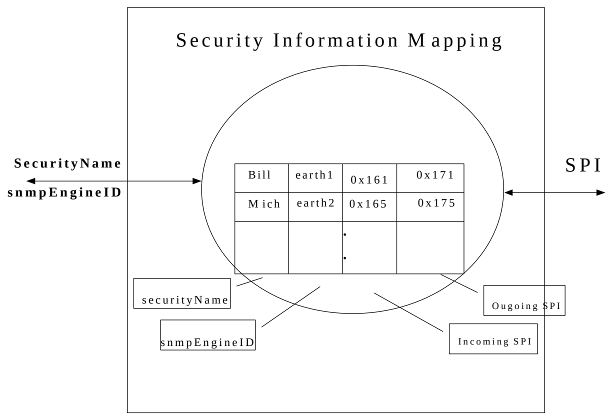


Figure 10: Security Information Mapping for SNMP over IPSec

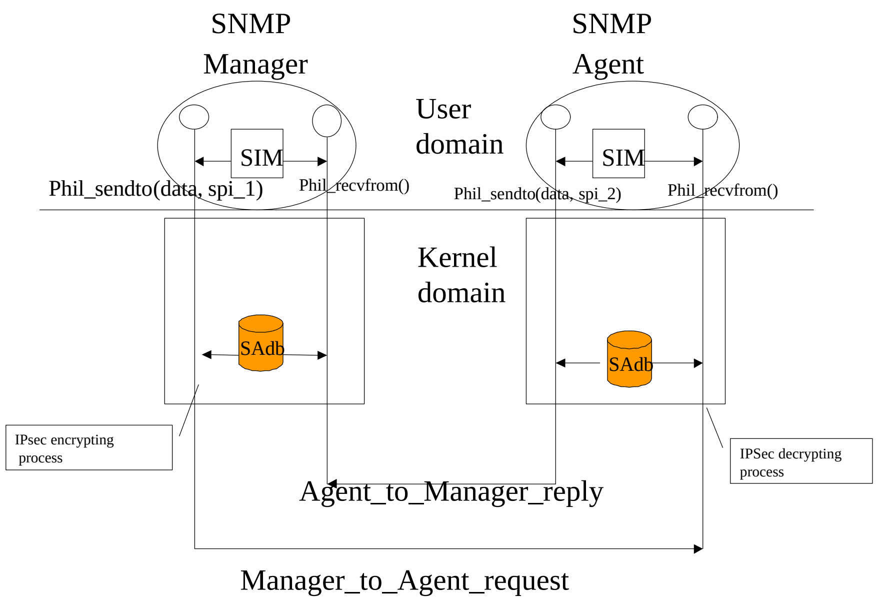


Figure 11: SNMP security architecture using IPSec.

For supporting SNMPv3 security on IPSec, first, we modified the SNMPv3 to support security information mapping (SIM) as shown on Figure 10. For example, The ***securityName***(Bill) and ***snmpEngineID***(Earth1) will generate two SPIs: SPI 0x161 for incoming, and SPI 0x171 for outgoing. Hence the SNMP packet data unit from SNMP manager to SNMP agent will be encrypted/decrypted using SPI 0x171; and when SNMP agent receives the request from SNMP manager, SNMP agent will use ***securityName***(Bill) to associate with the replying message and the replying security information (SPI 0x 161). With the PHIL-API (***phil_sendto( )***, and ***phil_recvfrom( )***), we can apply the specified SPI (generated by SIM) in the IPSec process.

6 Performance Evaluation

In this section, we evaluate the latency introduced by PHIL and PHIL-switching. The major delay is the PHIL extracting process (incoming side) and PHIL matching-and-replacing process (outgoing side). In PHIL-switching, we are interesting in combining with divert socket delay and switching entity delay. The measurement configuration consists three 450MHz Pentium II PC equipped with 10Mbit/sec ethernet cards. All of them are running Linux 2.0.36 platform with FreeS/wan 1.0.

6.1 FreeS/wan IPSec with PHIL implementation

The configuration (Figure 12) of measurement includes two machines: both are 450MHz Pentium II equipped with 10Mbit/sec ethernet cards. We use echo client/server (TCP) to launch 100000 packets each time and then record the average time. We also test the different data sizes in 16, 32, 64, 128, 256, 512, 1024 bytes. For any packet great than MTU, the packet will be fragmented and the PHIL process could be executed more than two times (depends upon the data size) in each packet. Hence we limited the packet size to be less than MTU such that we can simply calculate the average time delay for each packet.

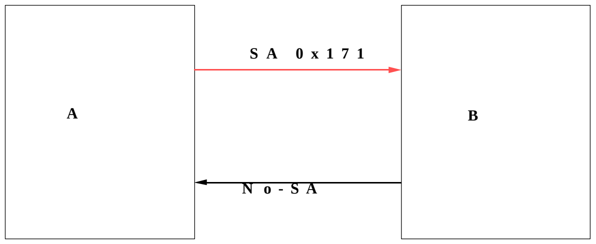


Figure 12: The configuration for PHIL evaluation

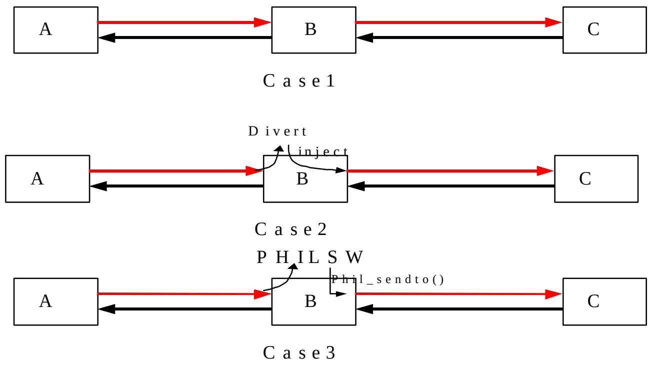


Figure 13: The configuration for PHIL Switching Evaluation

Three test cases are:

Case 1: original Linux 2.0.36 kernel, no IPSec, and no PHIL.
Case 2: Linux 2.0.36 kernel with FreeS/wan IPSec; no PHIL.
Case 3: Linux 2.0.36 kernel with FreeS/wan IPSec and PHIL.

The result is shown on table 6-1. From case1 and case 2, we show the IPSec overhead; and by comparing case2 and case3, we show the PHIL implementation performance.

Table 6.1 The test result of PHIL

size(bytes)	Case 1	Case 2	Case 3	Case3-Case2
16	363 us	ESP: 505 us	ESP: 506 us	ESP: 1 us
		AH: 502 us	AH: 502 us	AH: 0 us
32	398 us	ESP: 538 us	ESP: 539 us	ESP: 1 us
		AH: 535 us	AH: 536 us	AH: 1 us
64	465 us	ESP: 616 us	ESP: 617 us	ESP: 1 us
		AH: 613 us	AH: 613 us	AH: 0 us
128	601 us	ESP: 761 us	ESP: 762 us	ESP: 1 us
		AH: 756 us	AH: 757 us	AH: 1 us
256	870 us	ESP: 1050 us	ESP: 1051 us	ESP: 1 us
		AH: 1045 us	AH: 1046 us	AH: 1 us
512	1410 us	ESP: 1628 us	ESP: 1630 us	ESP: 2 us
		AH: 1625 us	AH: 1625 us	AH: 0 us
1024	1410 us	ESP: 1628 us	ESP: 1630 us	ESP: 2 us
		AH: 1625 us	AH: 1625 us	AH: 0 us

Since 3DES causes heavy CPU loading in ESP protocol, we modify kernel such that it is able to support light-ESP. Light-ESP just creates ESP header and trailer but it does not encrypt data payload. From Table 6.1, we observed that the delay introduced by PHIL (compare with Case2 and Case 3) is 1-2 micro-seconds.

6.2 PHIL-switching

To evaluate PHIL-switching, we set up the test configuration as shown in Figure 13. In each of the three cases, we use a program to send out packets in 10000 times and record the average time (total time/10000). In case 2, we use divert socket to obtain the incoming IP packet and then re-inject the diverted IP packet into kernel. In case 3, we also use divert socket to get the incoming IP packet and then use ***phil_sendto( )*** to push the data back to the kernel with specified SPI/SA.: Our test results are summarized in Table 6.2.

Figure 13: The configuration of evaluating PHIL-switching.

Table 6-2 The test result of evaluating PHIL Switching.

Data size	Case1	Case2	Case3	Divert socket overhead	PHILSW overhead
16	777 us	1845 us	1860 us	1068 us	1083 us
64	979 us	2458 us	2361 us	1479 us	1382 us
256	1879 us	19407 us	13698 us	17528 us	11819 us
512	2860 us	39012 us	27679 us	36152 us	24819 us
1024	5020 us	41439 us	29536 us	36419 us	24516 us

The result shows that the divert socket and PHIL-switching overhead is data size dependent. For example, the PHILSW overhead in 64 bytes is 1382 us. As data size increases to 1k bytes, the overhead is 24516 us. The result also shows us that the major overhead comes from the divert socket, since transferring data from kernel to user is quite expensive.

7 Remarks

IPSec has been standardized and widely deployed for securing private information over the Internet. Currently, VPN is the major application for IPSec since higher layers can not easily access and control IPSec-layer security services. Our PHIL-API design and implementation provides a possible bridge between IPSec and other security applications. We have demonstrated the usefulness of this new API for applications such as DECIDUOUS and SNMPv3. We believe that many other secure Internet applications can be built directly on top of IPSec, without having to re-develop yet another security module in the application layer. Furthermore, in the near future, we expect to see more and more hardware acceleration for IPSec (e.g., 3Com's IPSec NIC, and CellTech's Gigabit IPSec chip). Therefore, for high performance applications, it will be much more attractive to use IPSec/PHIL than the software-based lower-throughput transport/application layer protocols such as TLS or SSL. Practically, the PHIL-API is useful when the applications need to run on platforms not supporting other security protocols (e.g., TLS). Finally, through our implementation and evaluation, we have shown that the overhead (memory space and cpu time) in providing PHIL is quite reasonable – 1 to 2 microseconds per packet in software. The PHIL service has been integrated into the DECIDUOUS system, which can trace the true attack sources in a few seconds on top of a 10+ node testbed.

8 References

[1] S. Kent, and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

[2] S. Kent, and R. Atkinson, "IP Encapsulating Security Payload (ESP)," RFC 2406, November 1998.

[3] S. Kent, and R. Atkinson, "IP Authentication Header," RFC 2402, November 1998.

[4] Angelos D. Keromytis, John Ioannidis, and Jonathan M. Smith, "Implementing IPSec," IEEE, August 1997

[5] H.Y. Chang, S.F. Wu, et al., "Deciduous: Decentralized Source Identification for Network-based Intrusions", appeared in 6th IFIP/IEEE International Symposium on Integrated Network Management, IEEE Communications Society Press, May 1999

[6] H.Y. Chang, S.F. Wu, et al., "Design and Implementation of a Real-Time Decentralized Source Identification System for Untrusted IP Packets", appeared in DARPA Information Survivability Conference and Exposition(DISCEX 2000), IEEE Computer Society Press, January, 2000.

[7] Linux FreeSwan Web site: http://www.xs4all.nl/~freeswan/

[8] Ravindra Narayan, "Socket API Extensions to Extract Packet Header Information List (PHIL)" Master thesis, May 1999. http://www.lib.ncsu.edu/etd/public/etd-3721141949931381/etd-title.html

[9] The Linux Kernel Archives: http://www.kernel.org/

[10] M. Beck, H. Bohme, M. Dziadzka, and U. Kunitz, "Linux Kernel Internals" Addison-Wesley Second version, 1998.

[11] Harrington, D., Presuhn R., Wijnen B., "An Architecture for describing SNMP management frameworks", RFC 2271, January 1998.

[12] D. Levi, P. Meyer, B. Stewart, "SNMPv3 Applications," RFC2273, January 1998

[13] Blumenthal, U., Wijnen B., "User-based Security Model (USM) for version 3 of the Simple Network Management Protocol", RFC 2274, January 1998.