# Preventing Theft of Quality of Service on Open Platforms

Kwang-Hyun Baek and Sean W. Smith
Department of Computer Science
Dartmouth College
Hanover, NH: 03755
{jbaek,sws}@cs.dartmouth.edu

## Abstract

*As multiple types of traffic converge onto one network, frequently wireless, enterprises face a tradeoff between effectiveness and security. Some types of traffic, such as* voice-over-IP (VoIP), *require certain* quality of service (QoS) *guarantees to be effective. The end client platform is in the best position to know which packets deserve this special handling. In many environments (such as universities), end users relish having control over their own machines. However, if end users administer their own machines, nothing stops dishonest ones from marking undeserving traffic for high QoS. How can an enterprise ensure that only appropriate traffic receives high QoS, while also allowing end users to retain control over their own machines?*

*In this paper, we present the design and prototype of a solution, using SELinux, TCPA/TCG hardware, Diffserv, 802.1x, and EAP-TLS.*

## 1 Introduction

Many enterprise IT infrastructures are experiencing "convergence." Increasingly many types of information services—such as telephony and multimedia—are moving onto users' general-purpose computers, and onto the network. Indeed, our university has already migrated to VoIP, and plans to run only one network in new dormitories, instead of three (intranet, telephone, and cable TV).

However, in order to provide acceptable performance, some of these applications require the network to ensure higher minimal QoS for their traffic. In a large campus environment, with users, occasionally uncooperative, who value individual and departmental autonomy over their machines, this situation creates a set of challenges:

1. The network and the user machines need to conspire together to ensure that traffic from appropriate applications receives the appropriate quality of service.

2. Rogue users, even with root access to their own machines, should not be able to steal high QoS for traffic from applications that do not merit it.

3. The users need to otherwise retain the control over their own machines to which they are accustomed.

### 1.1 Quality of Service

Quality of Service (QoS) is a concept of how "good" offered networking services are. QoS can be characterized by a number of specific parameters, ranging from low-level parameters, such as packet loss and guaranteed bandwidth, to end-to-end notions, such as *delay* (how long it takes to send information from one end node to another) and *jitter* (the variance of delay when sending multiple packets). When collision occurs or a a router's routing queue is exhausted, packets may be lost—resulting increased delay or jitter.

Specific applications may have specific QoS needs. For example, a multimedia network application might need to reconstruct audio and video signals from a stream of packets it receives. If the traffic experiences delay, the application gives an unpleasant half-duplex feel to the users. If it experiences jitter, the audio and video signals get interrupted causing distortion and unintelligible audio.

QoS architectures can provide QoS in terms of *guaranteed* services or *differentiated* services. Guaranteed services can guarantee certain QoS for a network application. Thus, a network application can specify the minimal delay, jitter, packet loss for its traffic. In contrast, differentiated services attempt to provide better QoS to special classes of traffic. In this case, a network application may ask the network routers to treat its traffic with special priority. The routers may expedite the forwarding of the application's packets or assure

that the application's packets do not get dropped. Differentiated services, however, do not make any guarantees on QoS.

Two networking architectures dominate current practice for QoS. *Diffserv* [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 7] is an example of differentiated services. In contrast, *Integrated Services (Intserv)* [11, 12, 13, 14, 15, 16, 17, 18] offers guaranteed services. While Intserv gives the network more control over which application can be given specific QoS, Diffserv's approach is more simple and scalable.

### 1.2 Theft

However, when Bob authorizes Alice to use his QoS-enabled network, he wants to make sure that Alice does not abuse the QoS architecture. For example, Bob may want to enforce that Alice's peer-to-peer file-sharing traffic, which Alice modified to resemble VoIP traffic, does not hoard the network resources that the VoIP traffic from other users needs. To guard against this kind of *theft of QoS*, Bob might set up a *QoS policy* that dictates what level of QoS that Alice's applications should receive.

Current QoS networking architectures enforce their QoS policies at the routers or gateways. These devices inspect the packets themselves, to determine which deserve higher QoS. Routers and gateways, however, cannot gather enough information from packet inspection to identify which application at the end nodes generated the packets—especially when the end nodes can shape their low priority traffic to appear as high priority traffic. Routers and gateways may attempt to use information in the packets—hardware address, IP address, port number, application protocol number, length of the data, identifiable patterns in the data. However, many existing techniques, such as MAC address spoofing and IP address spoofing, can easily change these values and bypass QoS policy enforcement at the router or gateway.

### 1.3 The Problem

Standard QoS networking architectures thus require that the network believes the application-labeling information that end nodes put on packets—except the root users on some end nodes may wish to cheat. We need a way to ensure that the network can believe this labeling information, despite such adversaries, while still providing users with open computing environments.

### 1.4 Our Solution

In this project, we provide a solution to this problem using trusted computing hardware now becoming commercially ubiquitous. To gain access to the network, an end node must prove knowledge of a private key. A kernel-level module at the end node can use this private key—and also is responsible for marking the QoS level on packets. When operating correctly, this module will follow the enterprise's QoS policy. Integrating measurements of the operating environment for this module into an on-board *trusted platform module (TPM)* ensures (for some level of adversary) that the module can use the private key only when it is operating correctly. Building this in Linux permits users to otherwise have freedom in configuring their machines; using the NSA's SELinux variant protects against malicious users with *root* access, who might otherwise subvert the packet-marking or steal use of the network-access private key.

This project builds on our previous Enforcer Linux and SELinux projects [19, 20, 21], and uses the 1.1b TPM from the *Trusted Computing Group (TCG)* (formerly the *Trusted Computing Platform Alliance, TCPA*) [22, 23].

#### This Paper

Section 2 reviews the building blocks of our design. Section 3 describes how we put these pieces together and presents the big picture of the proposed system. Section 4 discusses our prototype and presents some performance measurements. Section 5 provides a security analysis. Section 6 discusses related work. Section 7 concludes with some directions for future work.

## 2 Building Blocks

### 2.1 The Diffserv Architecture

We need a network architecture to provide quality of service.

As discussed earlier, Diffserv [1] is a QoS architecture that categorizes network traffic into QoS classes so that traffic of a higher priority class receives better QoS than the traffic that belongs to a lower priority class. Diffserv appears to be the QoS architecture most widely used in practice.

Diffserv consists of components for packet classification, packet marking and *per hop behavior (PHB)* enforcement. In Diffserv, network QoS policies divide network traffic into different QoS classes. The *Differentiated Services Code Point (DSCP)* value in the *Differentiated Services (DS)* field[1] in the IP header is used to classify a packet. The network specifies what type of packets belong to which class and maps the classes to different DSCPs (packet classification). Ingress network nodes, such as border routers and gateways, inspect packets and mark the DSCP of each packet according to the QoS policy of the network (packet

[1] The DS field supercedes the IPv4 Type of Service octet and the IPv6 Traffic Classifier octet.

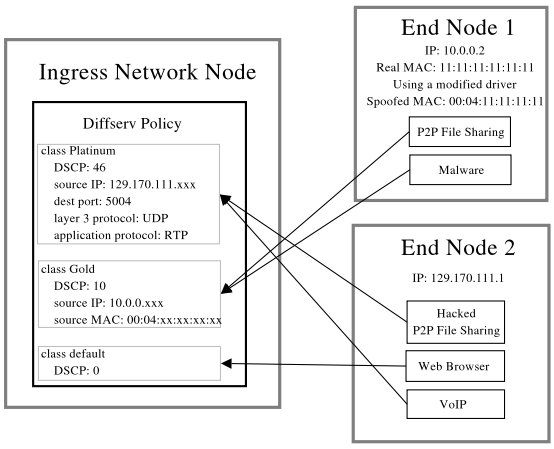



**Figure 1. The limitations of centralized Diffserv.**

marking). Other routers of the network then handle the packet according to the QoS level associated with the DSCP value of the packet (PHB enforcement). PHB mechanisms, such as *assured forwarding(AF)* [3] and *expedited forwarding (EF)* [4], are used to provide better QoS to the traffic with higher priority.

As we discussed in Section 1, because the ingress network nodes do not know which application is issuing the packets, they must rely on the information within the header of each packet when they classify and mark the packets. This limitation hinders the network administrator from making an application-level QoS policy. Once the attacker figures out the packet-level QoS policy, the attacker can form low-priority packets to resemble high priority packets to bypass the QoS policy enforcement. As shown in Figure 1, End Node 1 can get all its packets—including the ones issued by malware in its machine—classified illegitimately as "Gold" by using a modified network hardware driver that can spoof the source MAC address. Furthermore, End Node 2 can get all of its peer-to-peer file sharing traffic marked "Premium" by hacking the program to modify the packets to resemble VoIP traffic, which often uses RTP protocol and destination port 5004.

Thus, it is desirable to move classification and marking to the end nodes that produce the packets, where we know which applications are issuing the packets. Then we can have more fine-grained, application-based QoS policy. However, this approach works only if we can guarantee that all the end nodes will obey the network QoS policy when marking the DSCP field of the packets. We can provide such guarantees using trusted computing hardware and higher-assurance operating systems.

## 2.2 TCG Hardware

We need a way for the network to be able to trust the QoS labeling carried out on end nodes.

Trusted computing tries to answer the the following question: how can Alice trust computation that occurs in Bob's computer? The TCG has come up with an answer for general-purpose computing platforms that targets a tradeoff between affordability and security. In the TCG design, a Trusted Platform Module [22, 23] measures and attests the configuration of Bob's computer to Alice. The TPM can provide further assistance in security protocols by providing *sealed storage* that binds data (such as private or secret keys) to a specific configuration of Bob's computer.

In the PC implementation, the TPM is mounted on the motherboard and (via careful interleaving with bootstrap) measures the host machine's hardware and software configuration to sixteen *platform configuration registers (PCRs)* in the form of SHA-1 digests, loaded in a one-way format. The first eight PCRs are filled with hashes of BIOS, hardware configurations, ROM, the bootloader and its configuration. The usage of the other eight PCRs is up to OS and application designers, and may be used to record the configuration of the applications and drivers. As noted earlier, the TPM can also seal data items to the system configuration, as represented by a suite of specific PCR values. If a data item is an RSA private key, the TPM can also be configured to never unseal it—but rather only perform the RSA private key operation when the system configuration is appropriate.

The 1.1b TPM has been shipping on many IBM platforms for several years. A 1.2 version has been announced, but (at the time of our experiments) was not yet available.

Whether or not one agrees with the TCPA/TCG architecture or some of its potential commercial applications, two facts remain. This hardware is becoming commercially ubiquitous, so a feasible deployment base exists if one wants to build a trusted computing application. Also, the specifications for the TPMs are public, so one *can* build to it.

## 2.3 The Enforcer LSM

We need a way for the end node OS to actually work with the TPM.

The TCG has specified a *TCG Software Stack (TSS)*, and an open-source implementation of this recently became available [24]. However, we hadn't wanted to wait for that, so our lab had earlier built the *Enforcer* open-source integration of the TPM with Linux, independent of the TSS [19]. Sailer et al. presented a design extending the TSS approach to the application layer in Linux [25].

For our experiments, we chose the Enforcer platform [21].

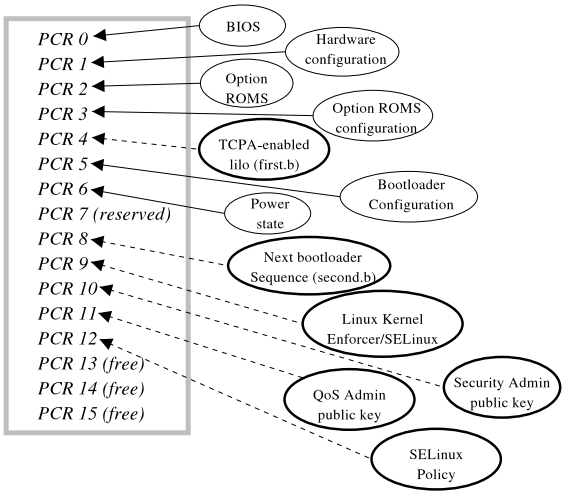



**Figure 2. Enforcer's long-lived configuration. The TPM reports the configuration of hardwares, BIOS, ROMS, and the bootloader in the first 8 PCRs. The Enforcer uses PCRs 8–12 to report the configuration of the rest of the long-lived components.**

The TPM architecture binds data to host configuration. Standard PKI-based authentication and authorization systems use a keypair, whose private key belongs to that entity and whose public key has been certified by some authority. Using such a scheme in a TPM-equipped host naturally suggests using the TPM to bind this private key to the host configuration. The configuration of the host machine, however, may change frequently, for example, with each change to a trivial file. Does such a change preserve the original entity, or not? If so, do rebind the secret somehow, or must the host make another trip to the CA?

The Enforcer architecture solves the problem by separating the configuration of a host machine into three levels: long-lived kernel and hardware configuration, medium-lived software, and short-lived operational data.

The Enforcer software includes modified boot code and a *Linux Security Module (LSM)*. For PKI-based host authentication, the Enforcer uses the TPM to tie the use of the RSA private key to a known, trusted long-lived configuration of the host machine through the use of PCRs. Figure 2 illustrates the long-lived configuration of the host machine reported by the PCRs. During the boot-up of the host, the PCRs are populated with with measurements reflecting the hardware configuration, and key software such as BIOS, the kernel, and the Enforcer code. If the PCRs matches the known configuration, then the TPM makes secrets available to the Enforcer for that boot cycle. One of these secrets is a private key matching a public key that has been certified in an X.509 identity certificate isssued for this machine.

For medium-lived software and files, a remote *Security Admin* issues a signed policy which describes a file structure that it believes to be trustworthy. The Enforcer, at the beginning of each boot cycle, checks the signature of the Security Admin's policy and loads the policy into the memory. To prevent from using a forged policy, the Security Admin's public key is hashed in a PCR and is part of the long-lived configuration, so if an attacker tries to modify the public key file, the secret bound the long-lived configuration becomes not accessible. At run time, the Enforcer intercepts relevant syscalls touching files and checks them against this policy.

Finally, the short-lived operational data is protected through the use of an encrypted loopback filesystem. If tamper is detected in the long-lived or medium-lived configuration, the Enforcer can unmount the filesystem and deny the access to the secret necessary to decrypt the data.

Marchesini et al. [19] showed that the Enforcer platform can be used protect the integrity of SSL server to give higher level of assurance to its customer that the private key is well-protected. In their implementation, an Apache Web server's private key is bound to the long-lived configuration of the Enforcer platform and is certified by the CA that vouches for the Enforcer kernel. The Enforcer enforces the medium-lived configuration of Apache, necessary scripts, and libraries through the use of the Security Admin's policy. Finally, operational data is kept in the encrypted loopback filesystem.

### 2.4 Adding SELinux

We need a way to protect against malicious root users.

Even with TCG hardware, the superuser in the traditional Unix/Linux access control model can nullify the binding between the configuration and the secret. For example, the superuser can use a debugger to read the memory location where the secret is loaded after it is released from the TPM. To solve this problem, Marchesini et al., in their later work [20] added *Security Enhanced Linux (SELinux)* to the Enforcer to provide *software compartments* to limit the superuser. SELinux [26, 27] is a Flask-based operating system that offers *mandatory role-based access control*. In the mandatory role-based access control model, a policy assigns roles to subjects—processes and users—and types to objects—memory locations, devices, files and sockets—and then describes how the subjects in each role can access the objects in each type. Using the mandatory role-based access control policy, SELinux can confine each application to its own compartment of objects called *domain*. Separating objects in the system into separate domains can, thus, prevent the superuser from spying on arbitrary memory location of the domain where it does not have permission to read memory.

## 3 Putting the Building Blocks Together

TCG hardware with the Enforcer and SELinux LSM provides a practical way to bind a secret to trustworthy kernel-level code, which in turn can use a host-specific private key only when the rest of the machine abides by some policy schema. In the SSL example, the Enforcer will not export the SSL private key outside the use of Apache web server, and ensures that the server can use it only when it is configured according to the latest safety guidelines and the Web content is suitably guarded by the OS.

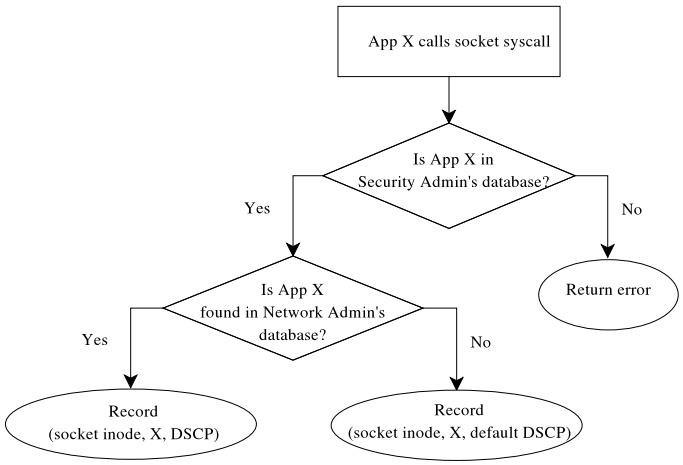
We can use a similar approach to solve the QoS theft problem. We choose a standard PKI scheme to require authorization for network access. We set up the trusted module at the end node to know that host's secret—and also to ensure that packets exiting the machine into the secure tunnel are marked correctly according to the network's QoS policy.

### 3.1 Distributed Packet Marking

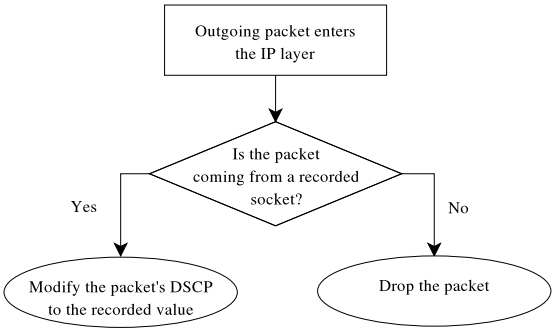
We introduce an additional remote party called the *QoS Admin*, who, similar to the Security Admin, issues a signed policy. This policy maps each network application to a DSCP. During the machine's bootup process, the QoS Admin's public key is also loaded into the PCR along with the Security Admin's public key, and the policy's signature is verified when the Enforcer loads the policy. The QoS Admin may need to consult the Security Admin policy when creating the QoS policy, since what an application does when executed can depend on other aspects of the system configuration besides the binary. Additionally, by granting an application the right to use a high QoS, the QoS Admin may also be granting that right to `fork`ed children.

Figure 3(a) describes how our kernel-level code can correctly identify the mapping between the packets and network applications. We added socket hooks to the Enforcer/SELinux LSM so that when INET or INET6 type sockets are created, the LSM records the socket's inode number, the program that created it, and the DSCP for the program in the QoS Admin's policy. The hook verifies that the program is listed in the Security Admin's policy and looks up the QoS Admin's policy for the hash of the program and the corresponding DSCP. If the program is not in the Security Admin's policy, the socket is not recorded and all the packets issued from the socket are dropped. Moreover, if the DSCP for the program is not specified in the policy, the packets from the socket will be assigned the default DSCP and will only receive best-effort service from the routers.

Figure 3(b) shows how we use the recorded socket inode and DSCP value to mark the outgoing packets. Packet marking happens at a POSTROUTING Netfilter hook [28]




(a) Socket hooks.




(b) Marking DSCP.

**Figure 3. Socket and Netfilter Hooks. (a) We added hooks to socket syscalls to map socket inode numbers to programs. We then use this mapping to identify which programs are issuing which packets. If the socket is created by a program that is not listed in the Security Admin's policy, the hook does not record the socket inode; all the packets issued from such a socket to get dropped at the kernel IP stack. The hook also finds the DSCP for the program in the QoS Admin's policy and records this value along with the socket's inode number. If the QoS Admin's policy does not include the program, then the hook records the default DSCP. (b) The Netfilter POSTROUTING hook at the kernel IP stack uses the recorded socket inode and DSCP values to mark the DS field of the IP packet before the packet is sent to the device driver. If the hook does not recognize the socket inode, then the packet is dropped.**