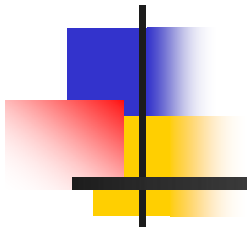
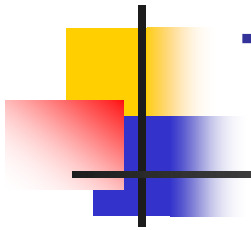


TCG 软件栈



TSS

- TCG软件栈(TSS)是可信平台模块和使用TPM功能的应用程序之间的支撑软件
  - 为操作系统和应用软件提供使用TPM的入口
  - 提供对TPM的访问、安全认证、密码服务和管理TPM资源等重要功能。

TSS

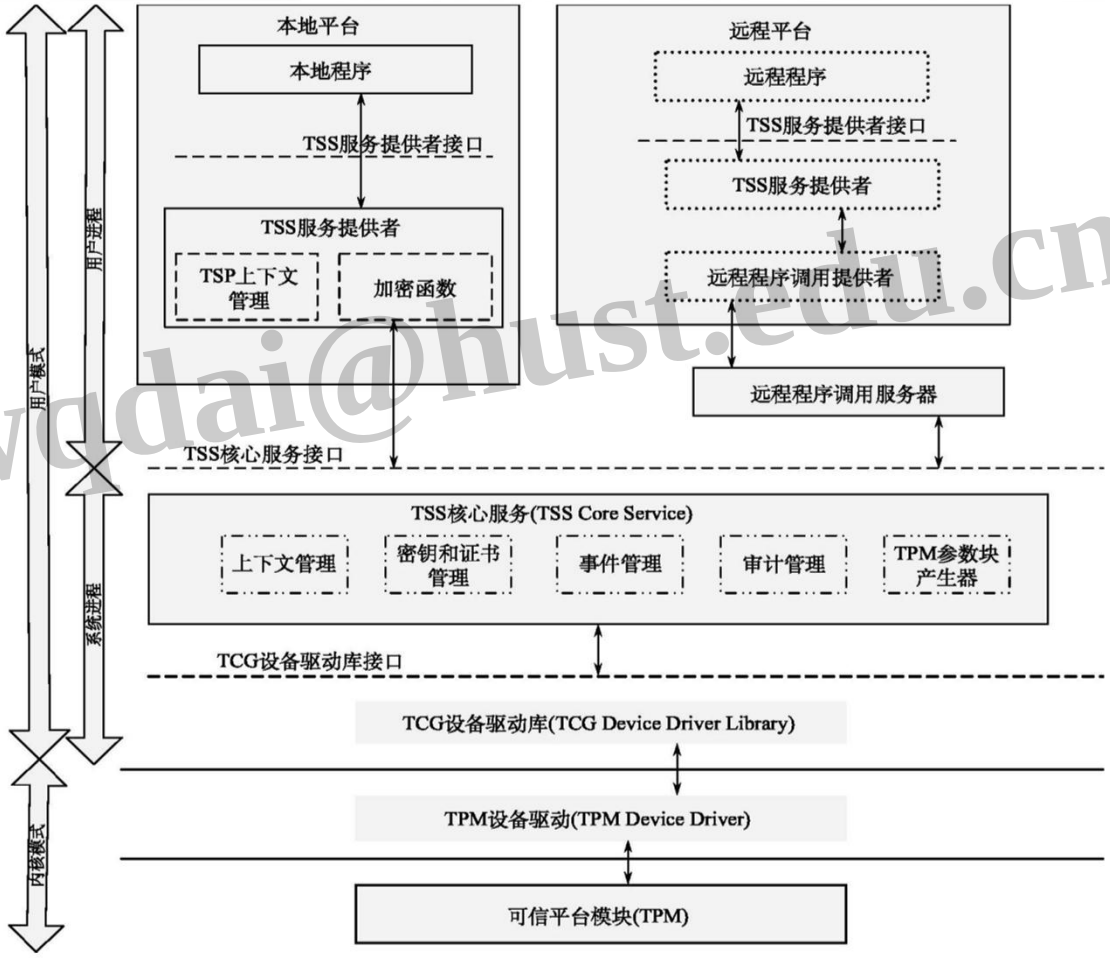
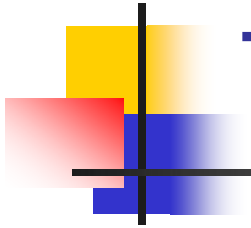


图 7.1　TSS 结构图



TSS

- TSS 的主要设计目标:
  - (1) 提供使用 TPM 功能的入口点；
  - (2) 提供对 TPM 的同步访问；
  - (3) 采用适当的字节排序和对齐为应用程序提供对底层指令流的屏蔽；
  - (4) 管理 TPM 资源(包括创建和释放)。

TSS总体结构及功能

- TSS 结构不依赖任何平台与操作系统。虽然各个模块和部分的状态在不同的平台和操作系统上会改变，但这些模块之间的通信和相互之间的关系都是一样的。

TSS总体结构及功能

- 平台模式：
  - (1) 内核模式:这是操作系统的设备驱动和核心组件常驻的地方。该区域的代码是用来维 持和保护用户模式中正在运行的应用程序 ,这些代码需要有管理员的某些权限才能修改。
  - (2) 用户模式:这是用户应用程序和服务执行的地方。这些服务和应用通常在用户的请求下才执行,但也可能在系统启动时被载入。操作系统提供一个或者多个受保护的区域,这些区域内的应用和服务在运行时相互提供保护。用户模式主要有两种应用：

TSS总体结构及功能

- (1) 用户应用。这些代码只有在用户请求或允许时才执行。由于这种代码通常比较新,并由用户提供，所以它不像平台上执行的其他代码那样具有可信性。
- (2) 系统服务。这些应用通常在操作系统的初始化过程中作为启动脚本的一部分执行或者由服务器请求执行。这些应用为用户提供普通的可信服务 。因为代码在各自的进程中执行,与用户应用程序分开 ,并且由操作系统执行,所以它比用户应用程序可靠,甚至可以与内核模式的安全性相比. 在Windows操 作系统中,这称为系统服务;在UNIX中,这称daemon (后台程序) 。

TSS总体结构及功能

- 因为TPM协议的安全性设计不依赖于数据传输的安全性
  - 任何一个TSS模块组件或者RPC（Remote Procedure Call）调用事件都不会影响到TPM的可信性
  - 所有TPM以外的模块、组件和接口与TPM相比都被认为是不可信的

TCG服务提供者

- 应用程序能依靠TSP 执行TPM 提供的可信功能
- 这一模块还提供 TPM 不提供的少量的辅助功能,如签名验证算法等
  - 任何的应用程序想要使用 TPM功能,都必须使用TSP并通过其接口 TSPI
  - 每个程序有属于它自身的TSP ,这个TSP 在任何进程使用它时作为库被载入
  - 每一个应用程序都可能有一个TSP ,在多任务操作系统中可以有多个TSP正在运行。

TCG服务提供者

- TSP接口(TSPI): 一个面向对象的接口，它与应用程序在同一个进程中
  - 当应用程序从用户那里收集到身份鉴定数据时，需要TSP 接口把数据传送到TSP进行处理

TCG服务提供者

- TSP上下文管理 (TSP Context Manager)
  - 提供动态处理以方便应用程序高效使用 TSP资源
  - 每一个处理都提供与TSP 关联的上下文
  - 在应用程序中，不同的线程可共享同一个上下文，也可以各自拥有一个上下文
- TSP密码函数(TSP Cryptographic Functions)
  - 为了充分使用TPM的保护功能,必须提供密码函数
  - 不需要对这些支持函数进行保护,典型的例子包括散列算法(Hash)和字节流产生器。

TCG 核心服务(TCG Core Service)

- TSS 核心服务(TCS)
  - 对所有在同一个平台上的TSP 提供统一的操作
  - 一个 TPM 只有一个TCS 实例
  - TCS 提供存储、管理和保护密钥的操作，也提供与平台相关的隐私保护
  - TCS 的执行就像是用户空间的守护进程

TCG 核心服务(TCG Core Service)

- TCS 接口(TCS Interface ,TCSI)
  - 允许多线程访问TCS，但它的每个操作都是原子操作
  - 在大多数情况下，它作为一个系统进程存在，与应用进程和服务提供者进程区分开
  - TCS和TSP的通信必须经过TCSI ，否则TSP和TCS传送的数据不能保证安全性，两者的通信会阻断

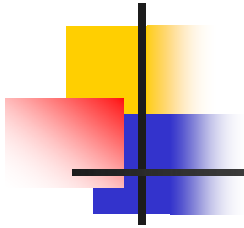
TCG 核心服务(TCG Core Service)

- TCS 上下文管理(TCS Context Manager)
  - 动态管理TSP和TCS 的资源，每次处理都为一组TCS的相关操作集提供上下文
  - TSP 中不同的线程可共享同一个上下文，也可以各自拥有一个上下文。

TCG 核心服务(TCG Core Service)

- TCS 密钥和证书管理(TCS Key&Credential Manager)
  - 负责管理、存储与平台相关的密钥和证书,并能为被授权的应用程序访问。

TCG 核心服务(TCG Core Service)



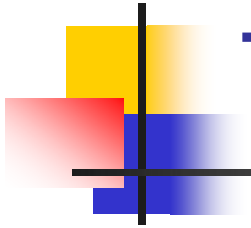
- TCS 事件管理(TCS Event Manager)
  - 管理与PCR相关的事件
- TCS的TPM参数块产生器(TCS TPM Parameter Block Generator)
  - 调用 TCS 的都是C语言风格的函数，通过字节流管理块与TPM 通信，此组件把传入TCS 的参数转换成TPM 所需的字节流。

TCG设备驱动库(TDDL)

- TCG 设备驱动库是TCS 和内核模式下的 TPM 设备驱动之间的中间件模块
  - TDDL 提供 了用户模式的接口，也提供内核态和用户态之间的转换。
  - TPM 不允许多线程执行，所以TDDL 是一个单实例的单线程模块。
  - TDDL通过TDDLI这个唯一的接口访问不同的TPM 硬件

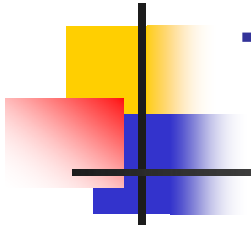
TCG设备驱动库(TDDL)

- TDDL 接口包括三种类型的功能。
  - 维护功能(打开、关闭和获得身份) :维护与TDD的交互
    - 提供函数(如Open, Close, GetStatus)来控制与TDD的通信
  - 间接功能(获得功能和设置功能) :获取或者设置TPM /TDD/TDDL的属性
    - 提供函数(如GetCapability和 SetCapability)来得到并设置TPM、TDD和TDDL的属性
  - 直接功能(发送和取消):发送或者取消 TPM的命令传输。
    - 通过函数(Transmit 和Cancel)发送和取消 TPM命令



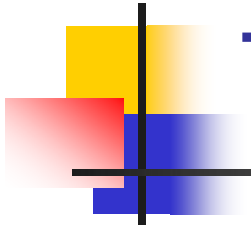
TCG设备驱动

- TPM是一种硬件，要想使用硬件必须要有设备驱动，即： TDD (TPM Device Driver)
  - 在系统启动时，TDD会被装载入系统，然后才能使用TPM提供的功能
  - TDD是使用TPM的一个核心部件,它从TDDL上读取字节流信息，发送给TPM，并接收反馈信息给TDDL



TCG设备驱动

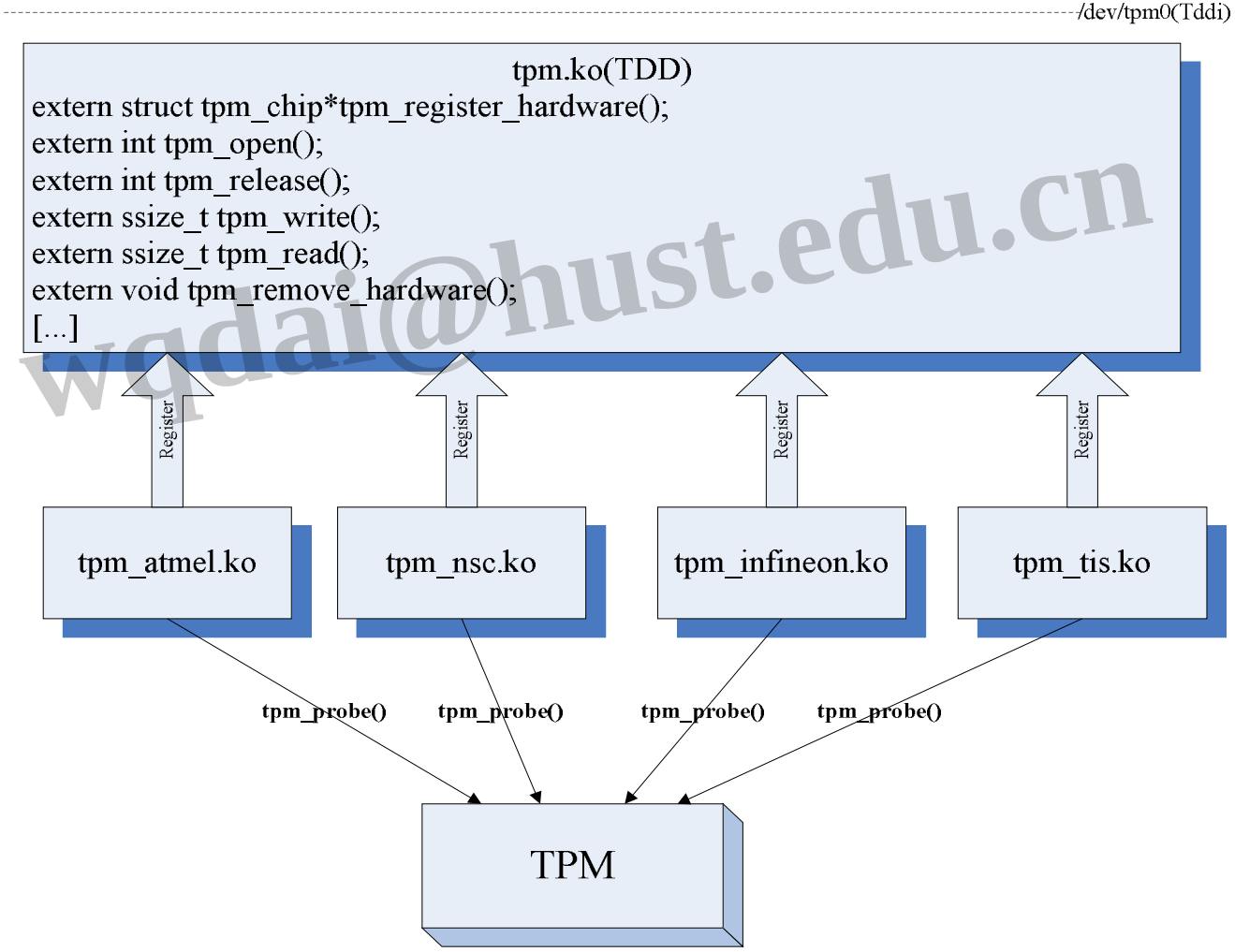
- IBM开发了一种适用Linux 的标准接口Linux TDD,可以使通信独立于TPM 芯片
  - 有利于应用程序的开发，只需要支持一种TPM通信方式即可。



TCG设备驱动

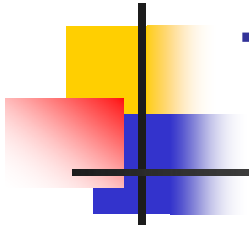
- Linux TDD 与Linux 融为一体，甚至允许在同一时间内使用多个TPM 芯片
  - 内核可以通过Linux 设备文件系统双向访问Linux TDD。

Linux TDD



TSS的调用方法

- 本地调用(LPC)
  - 本地调用就是在同一个进程或地址空间中直接调用其他的应用程序、模块或组件
- 远程调用(RPC)，远程调用提供进程间的交互，交互信息包括:
  - 为参数和结果编组和解组的规则集;
  - 进程间传递信息的编码和解码的规则集;
  - 触发单个调用,返回该调用的结果，以及取消该调用;
  - 需要维持的进程结构，以及由参与进程共享的参考状态。



TSS的调用方法

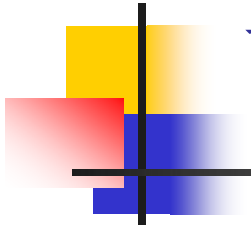
- 对远程连接进行过滤是由拥有者确定哪些命令可以被运行的最好方法
  - 确定一个连接上下文可以允许哪些命令是一种过滤方法
  - 在这种方法中，TCS 可以选择为无状态的或有状态的。

TSS的调用方法

- 无状态。部署一个无状态的TCS 更为容易，但也更容易遭受拒绝服务攻击或者其他的非法使用
  - TCS 无状态的部署中，管理员决定哪些函数能够供访问到本地机器的远程连接

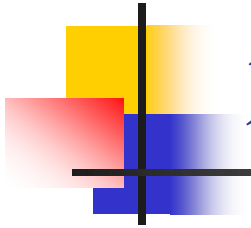
TSS的调用方法

- 有状态。一个监控状态的TCS的部署要复杂得多，但是也能有最好的保护来抵御拒绝服务攻击和减少隐私漏洞
  - 选择某一些类型的“服务”来供远程调用使用，而TCS 就必须实时监控它的状态以保证调用者没有非法使用



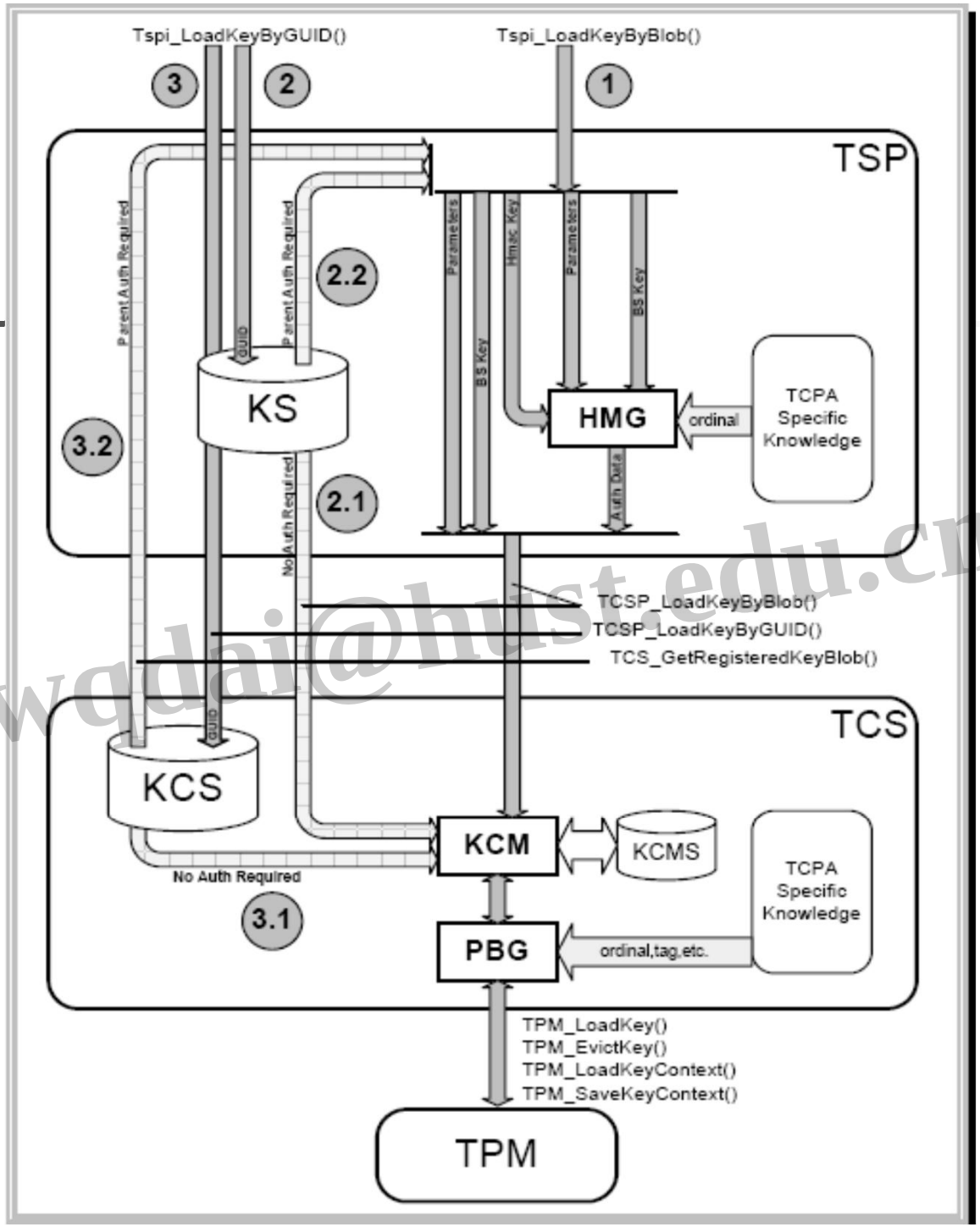
可信划分

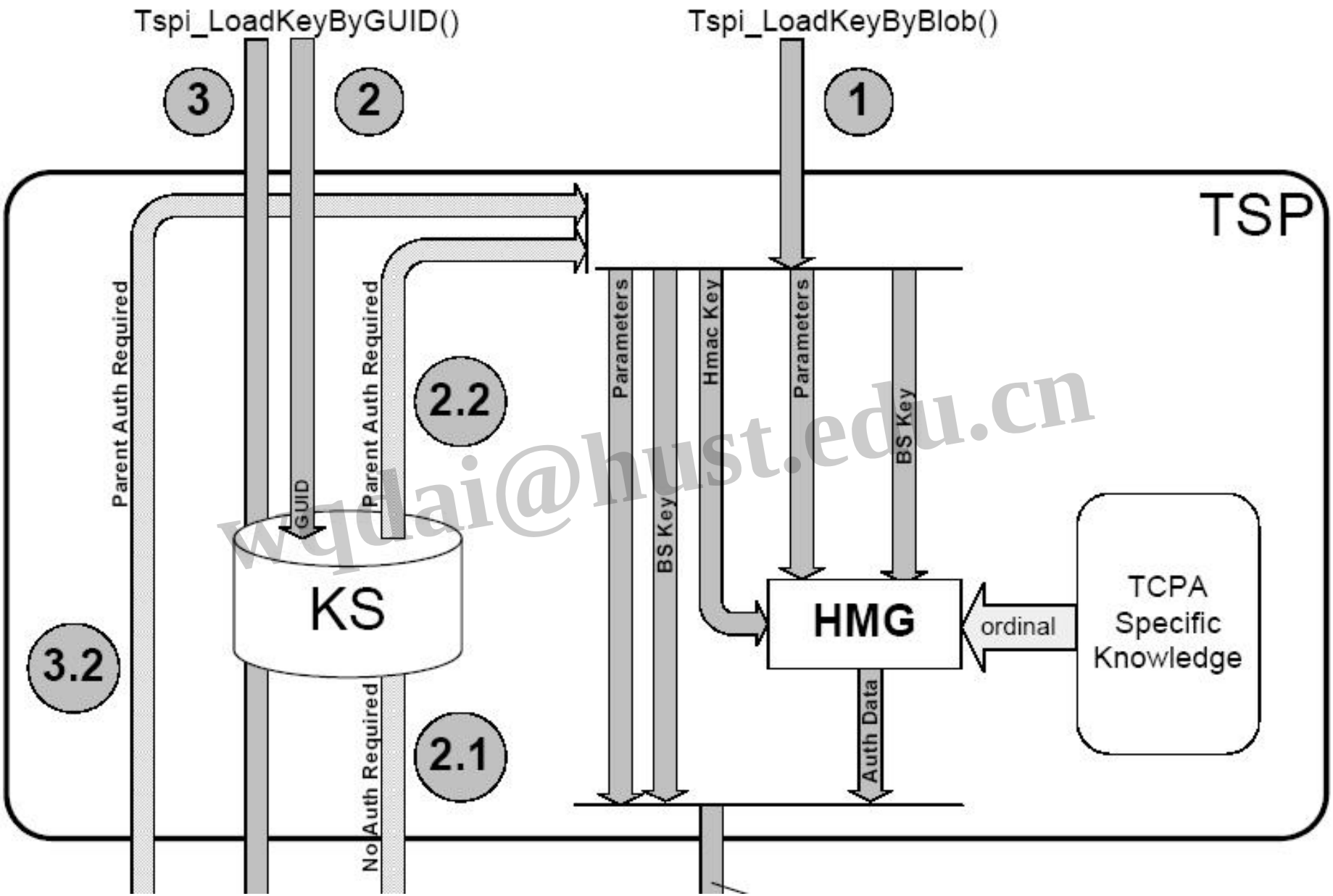
- 用分层的方法把需要确认可信的组件和不需要确认可信的组件分开，这样就能得到一个安全构架
  - 可信计算基(Trusted Computing Base ,TCB)
  - 在TCG的结构中,TPM周围的分界线就是TCB,所有在TPM以外的(包括TSS)组件都是不可信的。

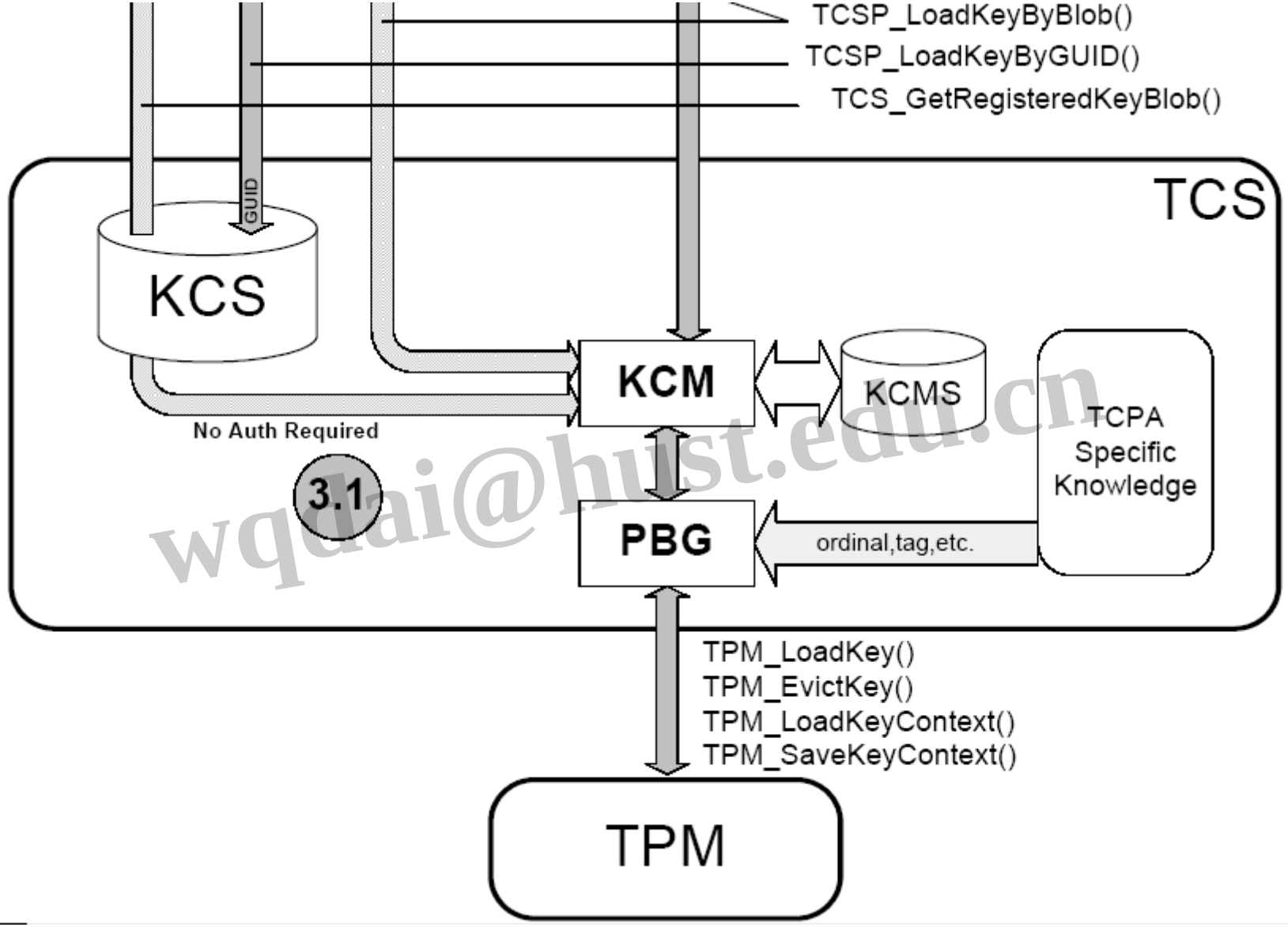


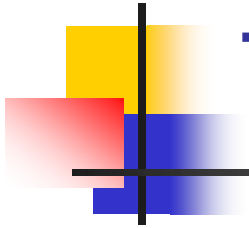
角色

- 实体(人或者机器)通过如下的角色执行 TSS 和 TPM 上的功能
  - TPM拥有者
  - TPM用户
  - 平台管理员
  - 平台使用者
  - 操作员
  - 公共用户



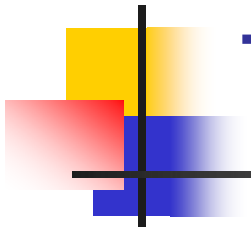






TSP接口

- 提供如下TPM功能的接口：
  - 完整性数据收集和报告服务
  - 安全存储服务
  - 密码学服务
  - 证书服务
- TSP向应用程序隐藏了授权会话相关的管理工作，应用程序不需要开启任何授权会话，而是由TSP开启授权会话并处理会话内部数据



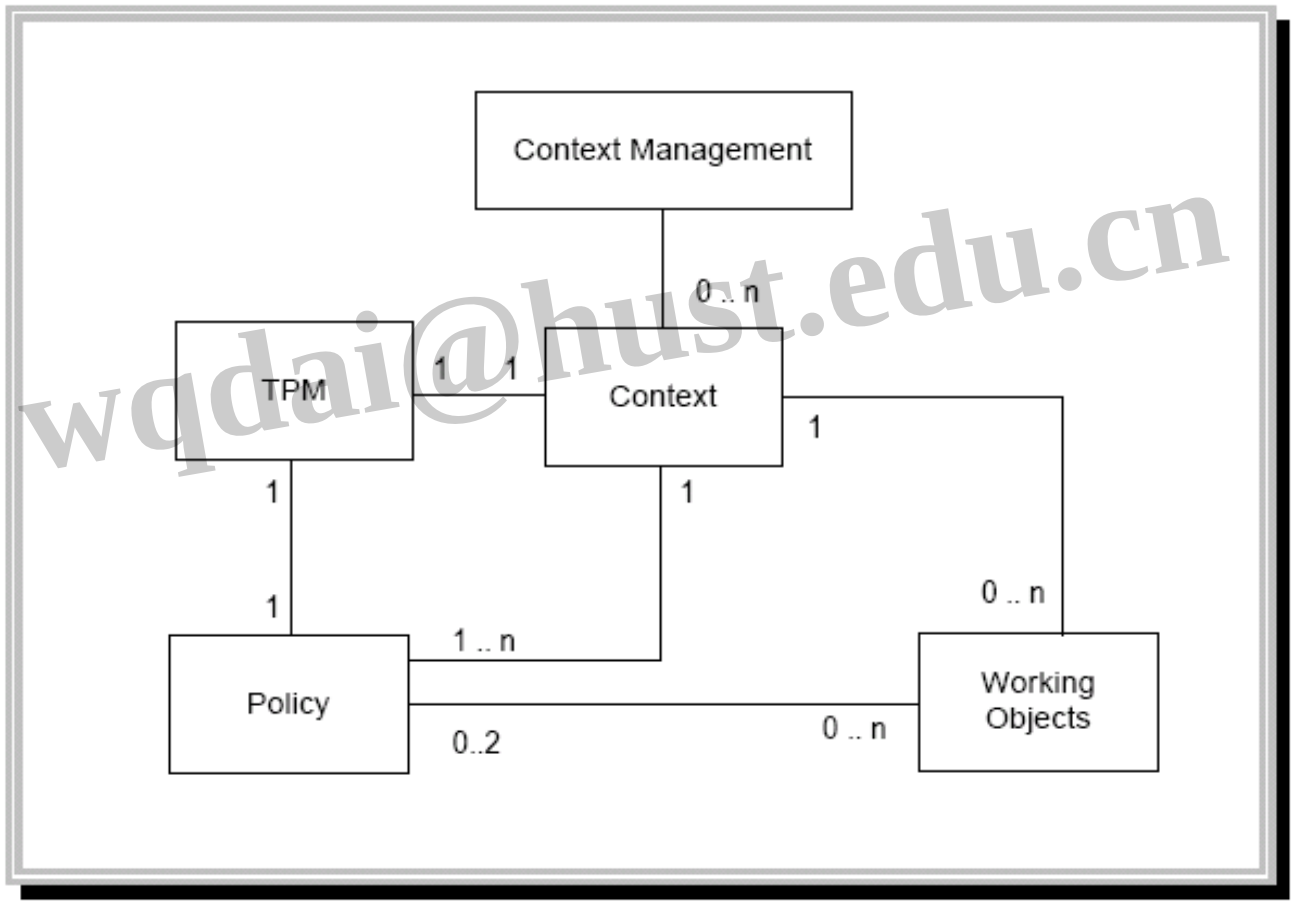
TSP接口

- TSPI定义了如下的类：
  - 上下文类
  - 策略类
  - TPM类
  - 密钥类
  - 加密数据类
  - 哈希类
  - PCR合成类
  - 非易失数据类
  - 可迁移数据类
  - 直接匿名认证类

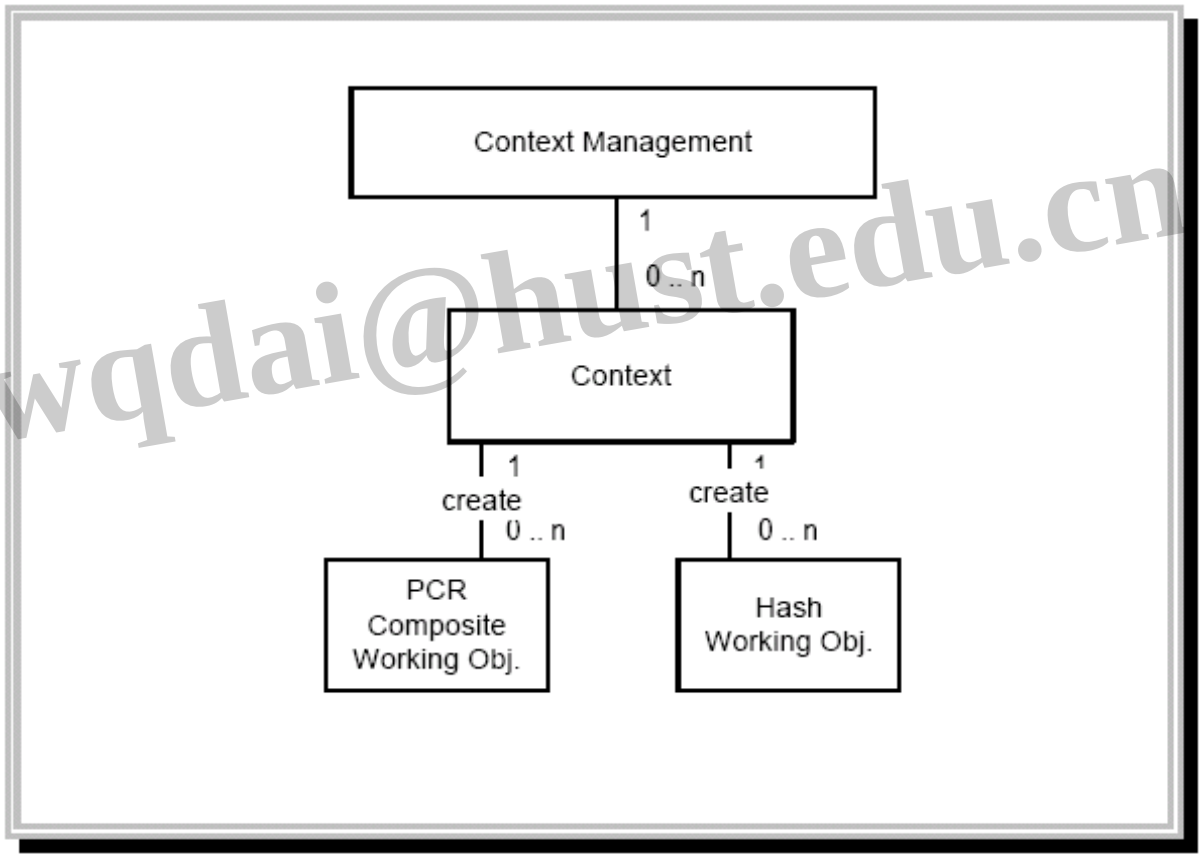
TSP Object Types

Object Type	C Data Type	Available in TSS Version
Context	TSS_HCONTEXT	1.1 and 1.2
Data	TSS_HENCDATA	1.1 and 1.2
Key	TSS_HKEY	1.1 and 1.2
Hash	TSS_HHASH	1.1 and 1.2
PCR Composite	TSS_HPCRS	1.1 and 1.2
Policy	TSS_HPOLICY	1.1 and 1.2
TPM	TSS_HTPM	1.1 and 1.2
Non-Volatile Data	TSS_HNVSTORE	1.2
Migratable Data Object	TSS_HMIGDATA	1.2
Delegation Family	TSS_HDELFAMILY	1.2
DAA Credential	TSS_HDAA_CREDENTIAL	1.2
DAA Issuer Key	TSS_HDAA_ISSUER_KEY	1.2
DAA Anonymity Revocation Authority Key	TSS_HDAA_ARA_KEY	1.2

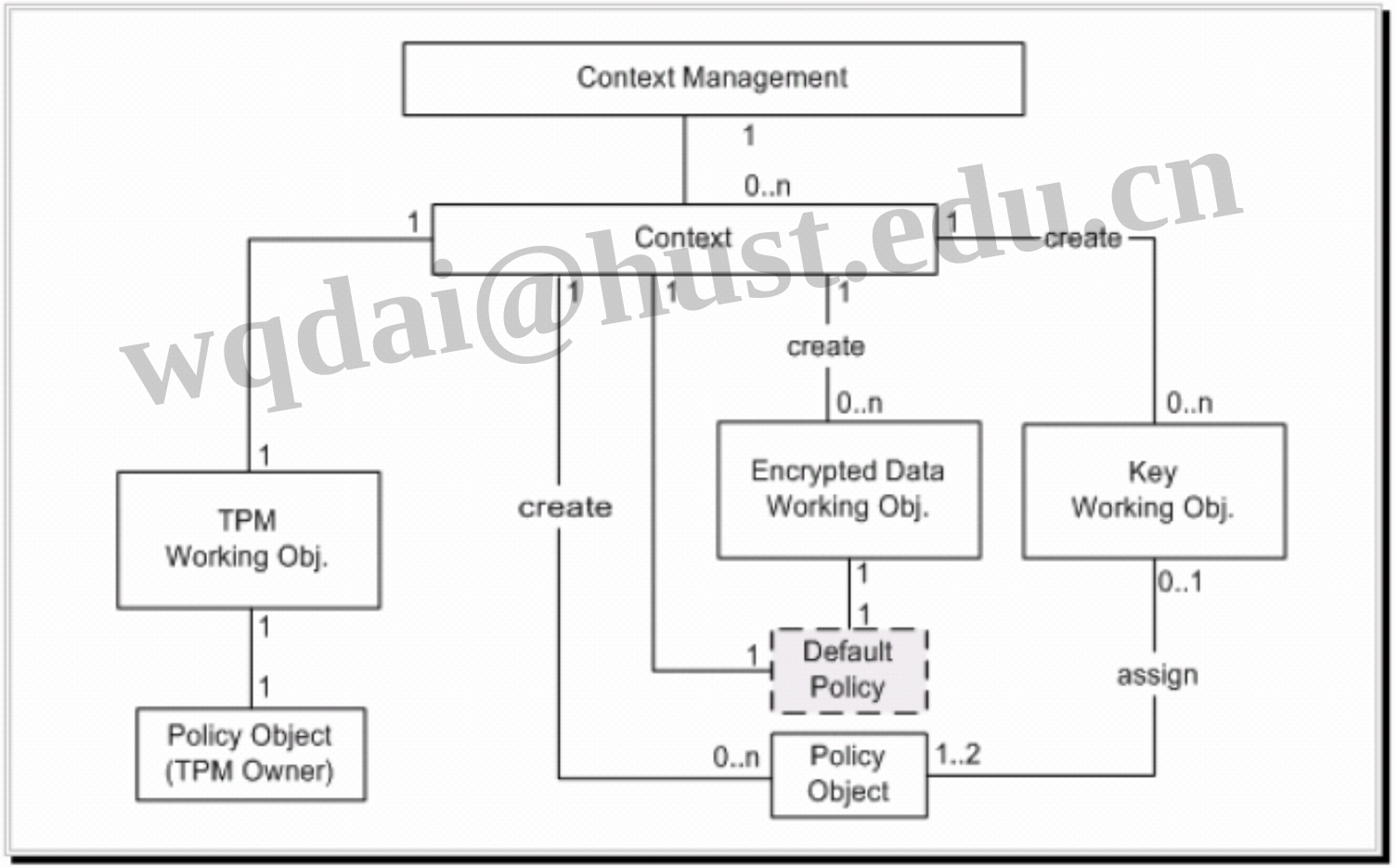
类对象的关系



非授权工作对象关系



授权工作对象关系



上下文类

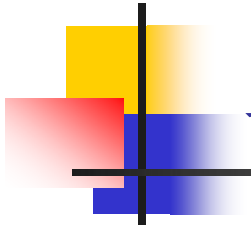
- TSS 应用程序在使用TSS 数据类型前必须初始化上下文对象。
- 上下文对象用来保存当前的TSP 库的句柄,连接本地和远程TCS 提供者,装载密钥,保存从磁盘中恢复的密钥对象,以及创建新的工作对象。
- 每个新创建的对象与创建它的上下文相关联。
- 使用Tspi_Context_Create可以创建新的上下文对象,使用Tspi_Context_Close关闭一个上下文对象。
- 调用 Tspi_Context_Create 之后,会产生两个事件:TSP 生成一个新的上下文对象;一个策略对象被创建并与该上下文相关联。这个策略对象将作为其他授权对象的默认策略。
- 使用 Tspi_Context_CreateObject 创建新的授权对象时,新创建的对象在默认情况下将与上下文对象的默认策略关联。如果使用对象时需要授权,TSP 将查询上下文的默认策略来获得授权数据。

上下文类

- 为了将命令发送给TPM ,TSP 的上下文必须与TCS 提供者相连
  - 使用API 函数Tspi_Context_Connect完成
    - Tspi_Context_Connect的入口参数是上下文的句柄、UTF-16 编码的主机名或目的系统的IP 地址。如要连接本机 ,只要用空指针(NULL pointer)作为目的地址即可。
- 如果成功连接 TCS 提供者,一个TPM 对象将被TSP 隐式地创建,并与上下文关联

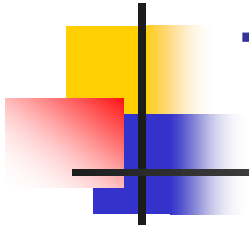
上下文类

```
//创建上下文
result = Tspi_Context_Create(&hContext) ;
if(result ! = TSS_SUCCESS){
  print_error("Tspi_Context_Create" ,result) ;
  exit(result) ;  }
//连接上下文
result = Tspi_Context_Connect(hContext ,get_server(GLOBALSERVER)) ;
if(result ! = TSS_SUCCESS){
  print_error("Tspi_Context_Connect" ,result) ;      exit(result) ; }
//关闭上下文
result = Tspi_Context_Close(hContext) ;
if(result ! = TSS_SUCCESS){
  if(!checkNonAPI(result)){
    print_error(nameOfFunction ,result) ;
    print_end_test(nameOfFunction) ;        exit(result);    }
  else{    print_error_nonapi(nameOfFunction ,result) ;
           print_end_test(nameOfFunction) ;        exit(result);    }  }
else{     print_success(nameOfFunction ,result) ;
print_end_test(nameOfFunction) ;     exit(0);  }
```



上下文类

- 一个应用程序可以创建任意多个上下文对象, 并能与任意多个TCS 提供者相连接
- 在TSS1.2 规范中, 有一个“上下文连接版本”的概念,它是用来控制使用上下文创建的对象的版本 。

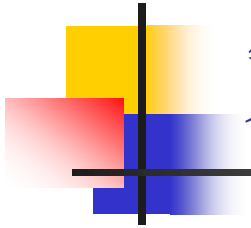


TPM类

- 当TSP上下文连接到TCS 提供者时,一个TPM 对象将被隐式地创建。
- 使用Tspi_Context_GetTPMObject 可以获取该TPM对象的句柄。
- 当TPM 对象创建时,TPM 对象将获得自身的策略对象,该策略对象通常都是用来为TPM 所有者保存授权数据。
- 下表列出了当创建一个所有者授权命令时,所有必要的Tspi 调用集合 。

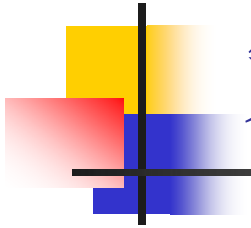
TPM类

```
TSS_HCONTEXT   hContext ;
TSS_HTPM   hTPM ;
TSS_HPOLICY   hOwnerPolicy ;
Tspi_Context_Create(&hContext) ;
//连接到本地 TCS 提供者
Tspi_Context_Connect(hContext ,NULL) ;
// 获得隐式创建的 TPM 对象的句柄
Tspi_Context_GetTPMObject(hContext ,&hTPM ) ;
Tspi_GetPolicyObject(hTPM ,TSS_POLICY_USAGE ,&hOwnerPolicy) ;
//设置秘密
Tspi_Policy_SetSecret(hOwnerPolicy ,... ) ;
// 将所有者授权的命令放到这里
```



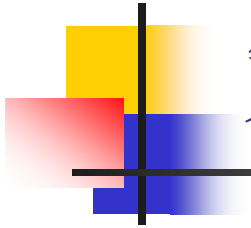
策略类

- 策略对象用来保存命令需要用到的授权数据
  - 在装载密钥、迁移密钥、加解密数据、获得TPM的所有权、获得或设置TPM的敏感属性时,都需要授权数据。
- 策略有三种类型:使用策略、迁移策略和操作策略。
  - 在创建可迁移密钥时才使用迁移策略。
  - 操作策略是为TPM 1.2 的API 函数 Tspi_TPM_SetTempDeactivated 保存授权数据。
  - 其他的策略都被认为是使用策略。



策略类

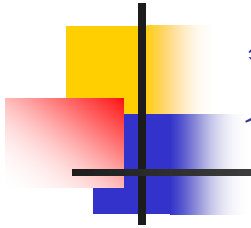
- 策略获得它的授权数据的方法称为秘密模式 (Secret Mode) 。
- 在默认的情况下,所有策略都以秘密模式 TSS_SECRET_MODE_POPUP 创建,也就是说 ,当该策略使用时,TSP 将弹出一个对话框以获取使用者的授权数据。
- 在第一次使用对话框获取授权数据后,授权数据将被保存,任何时候该策略被访问时这些数据可以被使用。
- 在调用Tspi_Policy_FlushSecret 刷新秘密数据后,如果需要使用授权数据,对话框将再次被使用。
- 表7-10 显示了策略对象所有可能的秘密模式。



策略类

表 7.10　策略的秘密模式

秘密模式	描　　述
TSS_SECRET_MODE_NONE	没有秘密和该策略相关联。如果一个授权命令相关的策略使用这种秘密模式，将返回TSS_E_POLICY_NO_SECRET 的错误码。Tspi_Policy_SetSecret 的入参 ulSecretLength 和 rgbSecret parameters 将被忽略
TSS_SECRET_MODE_PLAIN	传递给 Tspi_Policy_SetSecret 的数据将被 SHA-1 算法哈希，结果就是策略的秘密。长度为 0 的数据也允许
TSS_SECRET_MODE_SHA1	传递给 Tspi_Policy_SetSecret 的数据应是 20 字节的 SHA-1 哈希值，并可被修改地设置为策略的秘密
TSS_SECRET_MODE_POPUP	TSS 会弹出对话框从用户中获取秘密数据。Tspi_Policy_SetSecret 的入参 ulSecretLength 和 rgbSecret 将被忽略。这也是任何新策略默认的秘密模式
TSS_SECRET_MODE_CALLBACK	使用回调函数从应用程序中获取秘密数据。使用 Tspi_SetAttribUint32(TSS1.1)或 Tspi_SetAttribData(TSS1.2)在策略对象中设置回调函数的地址。Tspi_Policy_SetSecret 的入参 ulSecretLength和 rgbSecret 将被忽略

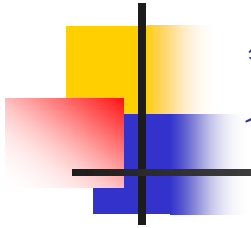


策略类

- 每个上下文对象和TPM 对象都有由TSP隐式创建的策略。
- 其他新创建的TSP 对象(密钥和数据)都可以引用上下文对象的策略
  - 如果一个对象需要唯一的密码才能访问,那么需要创建一个新的策略并与之相关联。
- 以下的程序显示了将策略分配给对象的过程。

策略类

```
TSS_RESULT   result ;
TSS_HPOLICY   hPolicy ;
TSS_HCONTEXT   hContext ;
TSS_HOBJECT   hObject ;
TSS_HENCDATA   hEncData ;
TSS_HTPM   hTPM ;
TSS_FLAG   initFlags = TSS_KEY_TYPE_SIGNING|TSS_KEY_SIZE_2048|
TSS_KEY_VOLATILE|TSS_KEY_NO_AUTHORIZATION|TSS_KEY_NOT_MIGRATABLE ;
//创建上下文
result = Tspi_Context_Create(&hContext) ;
//连接上下文
result = Tspi_Context_Connect(hContext ,get_server(GLOBALSERVER)) ;
//获取策略
result = Tspi_Context_GetDefaultPolicy(hContext ,&hPolicy) ;
result = Tspi _ Context_GetTpmObject(hContext, &hTPM) ;
//创建 hobject 对象
result = Tspi_Context_CreateObject(hContext , TSS_OBJECT_TYPE_RSAKEY , initFlags, &hObject);
result = Tspi_Context_CreateObject(hContext ,                    TSS_OBJECT_TYPE_ENCDATA,
TSS_ENCDATA_BIND,&hEncData);
//将策略分配给对象
result = Tspi_Policy_AssignToObject(hPolicy ,hObject) ;
result = Tspi_Policy_AssignToObject(hPolicy ,hEncData) ;
result = Tspi_Policy_AssignToObject(hPolicy, hTPM);
```



策略类

- 策略对象的秘密可以基于使用次数或使用时间设置为过期。 比如,创建一个使用10s 后过期的策略:


```
Tspi_SetAttribUint32(hPolicy , TSS_TSPATTRIB_POLICY_SECRET_LIFETIME ,
TSS_TSPATTRIB_POLSECRET_LIFETIME_TIMER ,10) ;
```

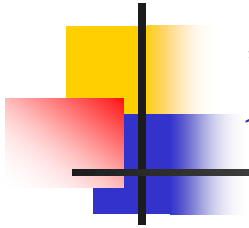
- 创建一个使用一次后过期的策略

```
Tspi_SetAttribUint32(hPolicy , TSS_TSPATTRIB_POLICY_SECRET_LIFETIME ,
TSS_TSPATTRIB_POLSECRET_LIFETIME_COUNTER ,1) ;
```

策略类

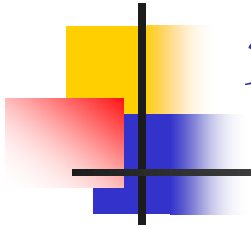
表 7.11　秘密类型的创建过程

秘密模式	秘密的创建过程	
	TSS 1.1	TSS 1.2
TSS_SECRET_MODE_NONE	无	
TSS_SECRET_MODE_PLAIN	Secret＝SHA1(传给 Tspi_Policy_SetSecret 的秘密数据)	
TSS_SECRET_MODE_SHA1	Secret＝传给 Tspi_Policy_SetSecret 的秘密数据	
TSS_SECRET_MODE_POPUP	Secret＝SHA1(由对话框中获得的 UTF16 编码的秘密数据＋\0\0\)	Secret＝SHA1(由对话框中获得的 UTF16 编码的秘密数据)
TSS_SECRET_MODE_CALLBACK	无	



策略类

- TSS_SECRET_MODE_CALLBACK 模式在下述情况下被应用程序使用：
  - 应用程序不愿（向TSS软件栈）泄露秘密；
  - 秘密由其它机制（如生物识别装置--biometric device）所获取；
  - 秘密由其它安全token（如smart card）所保护；

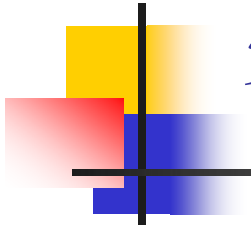


密钥类

- 密钥对象通常是RSA 公私密钥对，用来表示TPM 密钥
- 通过调用 Tspi_Context_CreateObject，密钥对象可以像其他对象一样被创建
- 调用TSS 的永久存储函数 ,如Tspi_Context_GetRegisteredKeyByUUID获取已有密钥。
- 为了创建1024 位的绑定密钥对，至少要调用两个API ,它创建密钥对象的过程如下:

密钥类

```
TSS_HCONTEXT hContext ;
TSS_HTPM hTPM ;
TSS_FLAG initFlags ;
TSS_HKEY hKey ;
TSS_HKEY hSRK ;
TSS_RESULT result ;
TSS_HPOLICY srkUsagePolicy ,keyUsagePolicy ;
initFlags = TSS_KEY_TYPE_SIGNING |TSS_KEY_SIZE_2048 |TSS_KEY_VOLATILE |TSS_KEY_AUTHORIZATION
|TSS_KEY_NOT_MIGRATABLE ;
//创建上下文
result = Tspi_Context_Create(&hContext) ;
//连接上下文
result = Tspi_Context_Connect(hContext ,get_server(GLOBALSERVER)) ;
//创建对象
result = Tspi_Context_CreateObject(hContext ,TSS_OBJECT_TYPE_RSAKEY , initFlags, &hKey);
//根据 UUID 加载密钥
result = Tspi_Context_LoadKeyByUUID(hContext ,          TSS_PS_TYPE_SYSTEM ,SRK_UUID,          &hSRK);
//获得策略对象
result = Tspi_GetPolicyObject(hSRK ,TSS_POLICY_USAGE ,&srkUsagePolicy) ;
//设置秘密
result = Tspi_Policy_SetSecret(srkUsagePolicy ,          TESTSUITE_SRK_SECRET_MODE ,
TESTSUITE_SRK_SECRET_LEN ,          TESTSUITE_SRK_SECRET);
//创建策略对象
result = Tspi_Context_CreateObject(hContext ,TSS_OBJECT_TYPE_POLICY ,TSS_POLICY_USAGE ,
&keyUsagePolicy) ;
//设置秘密
 result = Tspi_Policy_SetSecret(keyUsagePolicy ,          TESTSUITE_KEY_SECRET_MODE ,
TESTSUITE_KEY_SECRET_LEN ,          TESTSUITE_KEY_SECRET);
//将策略对象分配给密钥
result = Tspi_Policy_AssignToObject(keyUsagePolicy ,hKey) ;
//创建密钥
result = Tspi_Key_CreateKey(hKey, hSRK, 0);
```



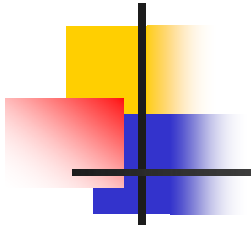
密钥类

- 调用Tspi_Context_CreateObject
  - 根据请求的属性,在TSP库中创建一个软件密钥对象结构。
- 调用Tspi_Key_CreateKey
  - 将软件密钥发送给TPM。TPM将依据软件密钥的属性产生密钥对,并将该密钥对返回给TSP。
- TSP 将生成的密钥对添加到它的软件对象中
  - 注意, 在调用Tspi_Key_CreateKey 之前要将hParentKey加载到TPM 中 。

密钥类

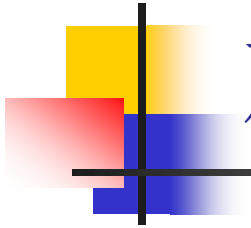
- 密钥对象是以数据块的形式从 TSS 中导出的---数据块就是被 TPM 使用的字节流格式。使用Tspi_GetAttribData 可以从数据块中取得密钥：

```
BYTE* keyBlob ;
UINT32 keyBlobLen ;
Tspi_GetAttribData(hKey ,TSS_TSPATTRIB_KEY_BLOB , TSS_TSPATTRIB_KEYBLOB_BLOB ,
& keyBlobLen ,& keyBlob) ;
```



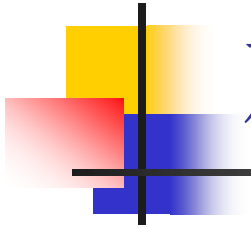
- 调用成功返回后, KeyBlob 将指向 KeyBlobLen 大小的分配的内存区域,保存着序列化的 TCPA_KEY 的结构。还可以设置其他的标志, 以使 Tspi_GetAttribData只获得密钥对的公钥部分(TCPA_PUBKEY数据块) :

```
BYTE* pubkeyBlob ;
UINT32 pubkeyBlobLen ;
Tspi_GetAttribData(hKey ,TSS_TSPATTRIB_KEY_BLOB ,
TSS_TSPATTRIB_KEYBLOB_PUBLIC_KEY , & pubkeyBlobLen ,& pubkeyBlob) ;
// 只 获 得 R S A 模 块
BYTE* modulus ;
UINT32 modulusLen ;
Tspi_GetAttribData(hKey ,TSS_TSPATTRIB_RSAKEY_INFO ,
TSS_TSPATTRIB_KEYINFO_RSA_MODULUS , &modulusLen ,&modulus) ;
```



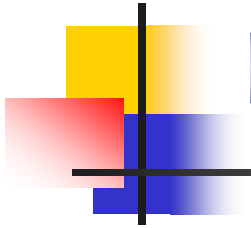
加密数据类

- 加密数据对象通常用来在密封或绑定操作中保存被密封的和被绑定的数据。
- 分别调用Tspi_SetAttribData 和Tspi_GetAttribData可以插入和提取数据。在绑定和密封操作时,被加密的数据将自动被TSS 插入加密数据对象中。



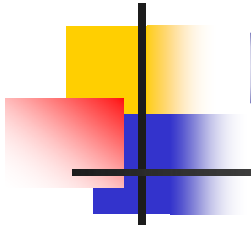
加密数据类

```
TSS_HENCDATA hEncData;
BYTE *blob, * data = “data”;
UINT32 blobLen;
Tspi_Context_CreateObject(hContext,TSS_OBJECT_TYPE_ENCDATA,
TSS_ENCDATA_BIND,&hEncData);
//调用 TSS 绑定数据,在 hEncData 中存储加密的数据块
Tspi_Data_Bind(hEncData,hKey,strlen(data),data);
// 获取加密的数据块,并将其存储在其他地方
Tspi_GetAttribData(hEncData,TSS_TSPATTRIB_ENCDATA_BLOB,
TSS_TSPATTRIB_ENCDATABLOB_BLOB, &blobLen, &blob);
```



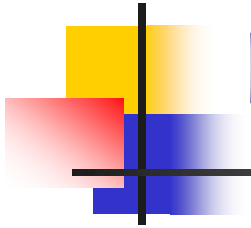
哈希类

- 哈希对象是用来保存哈希值，计算数据的哈希值,用密钥签名和认证哈希值。
- TSS 原本只支持使用SHA1 算法散列数据,但是它能签名和认证任意类型的哈希值。
- 调用Tspi_Hash_UpdateHashValue 可以创建使用SHA1 算法的数据的哈希值



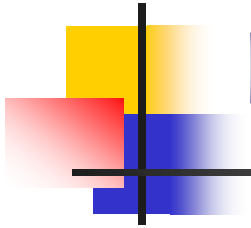
哈希类

```
TSS_HCONTEXT hContext ;
TSS_HHASH hHash ;
TSS_RESULT result ;
//创建上下文
result = Tspi_Context_Create(&hContext) ;
//连接上下文
result = Tspi_Context_Connect(hContext ,get_server(GLOBALSERVER)) ;
result =
Tspi_Context_CreateObject(hContext ,TSS_OBJECT_TYPE_HASH ,
TSS_HASH_SHA1,&hHash);
//更新 Hash 值
result = Tspi_Hash_UpdateHashValue(hHash ,24 , ''Jepense,dancjesuis'');
```



哈希类

- 虽然TPM 本身支持SHA1 算法,但是上例的SHA1 哈希是由TSS 实现的,而不是由 TPM实现。
- TPM 的 SHA1 哈希操作主要提供给在操作系统还未完全启动时使用,如 BIOS 加载。如果在任何时候加入TPM的SHA1操作,那么代价会相当大。
- 需要注意的是,调用Tspi_Hash_ UpdateHashValue意味着TSS 知道如何用与TSP哈希对象相关联的哈希算法来计算散列值
  - 因为TSS 1.1 规范和TSS 1.2 规范仅支持SHA1 算法,所以如果调用 Tspi_Hash_UpdateHashValue 时使用其他类型的算法 ,将会返回 TSS_E_INTERNAL_ERROR。

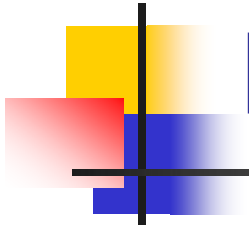


哈希类

- 如果需要签名或认证使用其他哈希算法 ,比如md5或SHA256 的散列值, 就不能使用TSS 创建哈希值
  - **需要支持其他哈希算法的外部库**
- 一旦使用要求的算法创建了哈希值 ,就可以调用TSS_HASH_OTHER 和 Tspi_Hash_SetHashValue 在TSS 对象中设置Hash 值
- 一旦一个哈希对象以类型 TSS_HASH_OTHER 被创建, 它不能再调用Tspi_Hash_ UpdateHashValue
  - Tspi_Hash_UpdateHashValue 只能操作 SHA1 哈希对象

哈希类

```
TSS_HHASH hHash ;
BYTE* digest = /* 哈希值 */ ;
UINT32 digestLen= /* 哈希值长度 */;
/ / 创建非 SHA1 的哈希对象
Tspi_Context_CreateObject(hContext ,TSS_OBJECT_TYPE_HASH ,
TSS_HASH_OTHER ,& hHash) ;
/ /在哈希对象中设置摘要值
Tspi_Hash_SetHashValue(hHash ,digestLen ,digest) ;
```



PCR合成类

- PCR 合成对象代表了一系列的哈希值及其合成摘要值。
- 以下两个操作将创建 PCR 合成对象：
  - 仅需要对PCR 的索引号操作,如 Tspi_TPM_Quote ；
  - 需要对PCR的索引号和PCR 的值操作,如 Tspi_Data_Seal。

PCR合成类

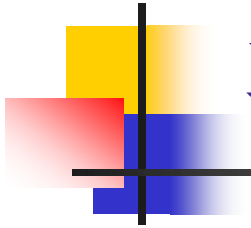
- 进行引用(Quote)操作时,首先 TPM 需要知道签名哪些 PCR ,然后TPM 需要使用特殊的密钥签名这一系列PCR 值。在这种情况下,使用 Tspi _PcrComposite_SelectPcrIndex

```
TSS_HCONTEXT   hContext ;
TSS_HPCRS   hPcrComposite ;
TSS_RESULT   result ;
/ /创建上下文
 result = Tspi_Context_Create(&hContext) ;
/ /创建 PCR 对象
result = Tspi_Context_CreateObject(hContext ,TSS_OBJECT_TYPE_PCRS ,
TSS_PCRS_STRUCT_INFO_LONG ,&hPcrComposite) ;
/ /选择 PCR 索引号
result = Tspi_PcrComposite_SelectPcrIndexEx(hPcrComposite ,8 ,
TSS_PCRS_DIRECTION_RELEASE) ;
```

PCR合成类

- 需要指定 PCR 的值并将其设置进一个对象的操作如下:

```
TSS_HCONTEXT   hContext ;
TSS_HPCRS   hPcrComposite ;
BYTE rgbPcrValue[20] =
{ 0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 } ;
TSS_RESULT   result ;
//创建上下文
result = Tspi_Context_Create(&hContext) ;
//创建 PCR 对象
result = Tspi_Context_CreateObject(hContext,
TSS_OBJECT_TYPE_PCRS, 0, &hPcrComposite);
//对指定 PCR 设置
result =
Tspi_PcrComposite_SetPcrValue(hPcrComposite ,8 ,20 ,rgbPcrVal
ue) ;
```



非易失性数据类

- 非易失性数据对象表示 TPM 自身内部的闪存区域(NVRAM)。NVRAM 是TPM 1.2规范的特征,它取代了TPM1.1规范中不够灵活的数据完整寄存器(DIM)。
- NV 对象定义空间的流程:

非易失性数据类

```
TSS_HCONTEXT hContext   = NULL_HCONTEXT ;
TSS_HNVSTORE hNVStore        = 0 ; / / NULL_HNVSTORE
TSS_HPOLICY   hPolicy         = NULL_HPOLICY ;
TSS_HTPM   hTPM          =NULL_HTPM ;
TSS_RESULT   result ;
/ / 创建上下文对象
result = Tspi_Context_Create(&hContext) ;
/ / 连接上下文
result = Tspi_Context_Connect(hContext ,NULL) ;
/ / 创建 TPM NV 对象
result = Tspi_Context_CreateObject (hContext ,TSS_OBJECT_TYPE_NV, 0 ,&hNVStore) ;
# ifdef NV_LOCKED / / 如果 NV_LOCKED
/ / 获得 TPM 对象
result = Tspi_Context_GetTpmObject (hContext ,& hTPM) ;
result = Tspi_GetPolicyObject(hTPM ,TSS_POLICY_USAGE ,&hPolicy) ;
/ / 设置密码
result = Tspi_Policy_SetSecret(hPolicy ,TESTSUITE_OWNER_SECRET_MODE ,
TESTSUITE_OWNER_SECRET_LEN ,TESTSUITE_OWNER_SECRET) ;
# endif
/ / 设置需要被定义的索引号
result = Tspi_SetAttribUint32 (hNVStore,TSS_TSPATTRIB_NV_INDEX,0 ,0x00011101 ) ;
/ / 设置对该索引号的许可
result = Tspi_SetAttribUint32 (hNVStore,TSS_TSPATTRIB_NV_PERMISSIONS,0 ,0x2000) ;
/ / 设置需要被定义的数据的大小
result = Tspi_SetAttribUint32 (hNVStore,TSS_TSPATTRIB_NV_DATASIZE ,0 ,0xa ) ;
/ / 定义 NV 空间
result = Tspi_NV_DefineSpace(hNVStore ,0 ,0) ;
```

可迁移数据类

- 可迁移数据类与被认证的可迁移密钥(CMK)一起使用,在传输数据块和属性时，用来存储可迁移数据块。
- 使用Tspi_SetAttribData 和 Tspi_GetAttribData 可以和其他对象一样获取和设置可迁移数据类的属性 。

直接匿名认证类

- 有三种不同类型的类可以用来表示在 DAA 协议中使用密钥和证书 ,包括 TSS_HDAA_CREDENTIAL、TSS_HDAA_ISSUER_KEY 和 TSS_HDAA_ARA_KEY。
- 由DAA密钥和证书使用的密钥数据类型可以由 Tspi_SetAttribData 和 Tspi_GetAttribData设置和获取类型的属性。

TSS 返回码

- Tspi API 的返回值是32 位的无符号整型值 ,由TSS_RESULT表示。这些值都经过编码, 指出TSS 哪一层出错、错误值,以及一些操作系统的出错信息。
- TSS_RESULT高16位存储与操作系统相关的信息,紧接着的4位定义出错的TSS 层次(TSP 、TCS 、TDDL 或TPM) ,最低的12位表示真正的错误码。

TSS 返回码

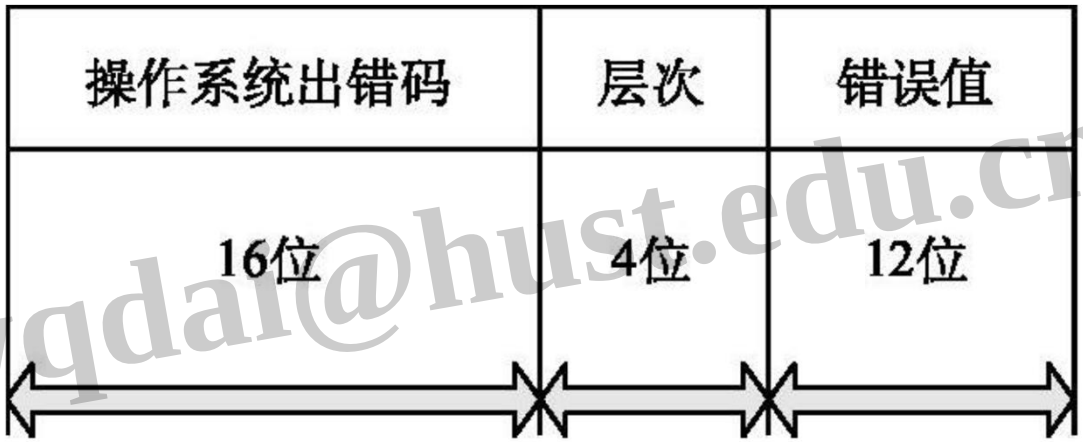


图 7.3　TSS_RESULT 的布局

持久性密钥存储

- TSS 提供的独立于TPM 的一个服务就是密钥存储。
- 在TCS 层,密钥被存储(注册)在磁盘上一块称为系统永久存储区的区域。
- 由于TPM 在密钥注册的过程中不实施任何强制决策,所以任何应用程序都可以无限制地访问系统永久存储区域。
- 客户存储区域由TSP 库管理,并且对于访问它的用户程序是私有的。
- 尽管客户存储区域是TSS 指定的 ,但是它通常保存在主属进程(即TSP 的主进程)可以访问的地方,如用户的主目录。

持久性密钥存储

- 用户和系统存储区域都使用TSS_UUID 结构在存储区内寻址。每一个密钥都在其永久存储区域内有唯一的UUID。
- 如果一个应用程序使用已存在的UUID存储一个密钥,API 将返回 TSS_E_ALREADY_REGISTERED。
- TCG已经预先定义了一些UUID 值来访问已知的密钥, 如TPM 的存储根密钥SRK。
- 为了在系统永久存储区中存储密钥 ,可以调用Tspi_Context_RegisterKey。应用程序在存储密钥时需要传递UUID信息,对于密钥的父密钥也同样。

持久性密钥存储

- 在客户永久存储区域内存储存储根密钥的子密钥 ,可以参考如下的步骤:

```
TSS_UUID SRK_UUID = TSS_UUID_SRK ;
TSS_UUID keyUUID = /* 唯一的 UUID 值 */ ;
Tspi_Context_RegisterKey(hContext ,hKey ,TSS_PS_TYPE_USER ,
keyUUID ,TSS_PS_TYPE_SYSTEM , SRK_UUID) ;

/ / 然后从存储区加载密钥
TSS_HKEY hKey ;
TSS_UUID keyUUID = /* UUID 值 */ ;
/ / 创建该密钥的应用程序句柄 ,并加载进 T PM
Tspi_Context_LoadKeyByUUID(hContext ,TSS_PS_TYPE_USER ,
keyUUID ,&hKey) ;
```

持久性密钥存储

- 使用TSS 的永久存储函数对于任何应用程序来说都是保存和加载密钥的简单方法。通过TSS 函数可以很方便地管理大量的密钥管理,也可以查询TSS密钥体系的描述信息,如调用Tspi_Context_GetRegisteredKeysByUUID

```
TSS_KM_KEYINFO* info ;
UINT32 infoLen ;
Tspi_Context_GetRegisteredKeysByUUID(hContext ,
TSS_PS_TYPE_SYSTEM ,NULL , &infoLen ,&info) ;
```

- 执行上述的调用将返回一个TSS_KM_KEYINFO 结构的数组
  - 包括每个密钥及其父密钥的UUID ,还包括该密钥是否需要授权、该密钥是否被加载,以及一个可选的由TSS设置的厂商数据项等内容

设置回调函数

- Tspi允许应用程序为一些不同的操作设置回调函数,如授权过程和身份密钥生成过程。回调函数可以有效防止应用程序将私密数据泄漏给TSS 。


```
TSS_CALLBACK cb ;
cb.callback = my_callback_func ;
cb.appData = my_data;
cb.alg = TSS_ALG_3DES;
Tspi_SetAttribData(hPolicy ,TSS_TSPATTRIB_POLICY_CALLB
ACK_HMAC , 0 ,sizeof(TSS_CALLBACK) ,&cb) ;
```

TSS确认数据结构

- Tspi 需要允许应用程序给某些 API 提供数据以供它们在操作中使用。比如,当Tspi_TPM_GetPubEndorsementKey 被调用时,TSS将20个字节的随机数传递给TPM ,这样TPM 能将其包含入哈希值内。TPM 计算:

  checksum = SHA1(TSS_Nonce||TCPA_PUBKEY(EK))

- 并将计算得的校验和数据传递给TSS。当TSS从TCS 中收到返回的数据时, 它使用返回的EK的TCPA_PUBKEY结构计算出相同的哈希值,并将其与由TPM 返回的校验和相比较。这使 TSS 可以认证由TPM 返回的数据的完整性

TSS确认数据结构

- 如果应用程序不能信任 TSS, 在这种情况下为了允许应用程序将数据传递给TSS 使用,可以使用 TSS_VALIDATION 结构
- 为了将 TSS 选择的随机数传递给TPM ,可以通过设置外部数据参数(应用程序所提供的) 以创建 TSS_VALIDATION 结构,并将其传递给 Tspi_TPM_GetPubEndorsementKey :

```
TSS_VALIDATION vData ;
vData.rgbExternalData = /* 20 位 的 随 机 数 */ ;
vData.ulExternalDataLength = 20 ;

Tspi_TPM_GetPubEndorsementKey(hTPM ,FALSE ,&vData ,&hPubEK) ;
```

TSS确认数据结构

- 在该API调用之后,该结构的数据和确认数据域都将被 TSS 填充。 rgbData 域将包含提供的随机数和从 TPM 返回的数据。 rgbValidationData 域包含 TPM 返回的校验和的值。然后,应用程序自己负责认证：

vData.rgbValidationData = SHA1(vData.rgbData)