阿里云
IoT安全运营中心

设备开发指南

文档版本：20190915

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的”现状“、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含”阿里云”、Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

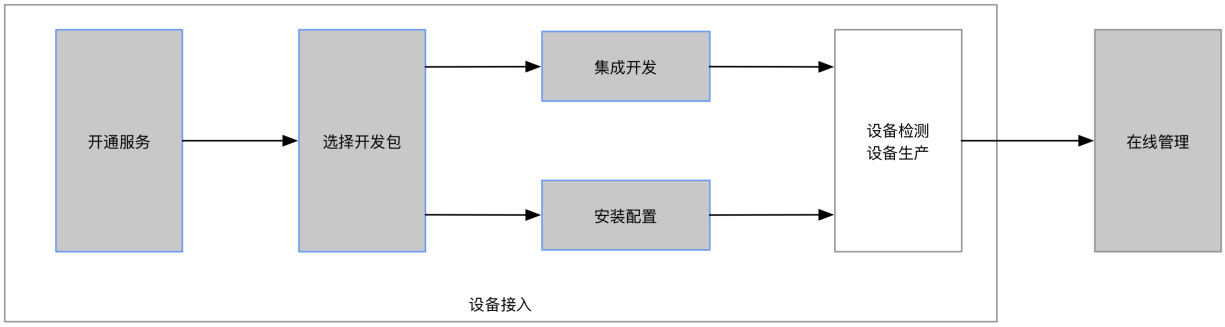
格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **禁止:** 重置操作将丢失用户配置数据。
	该类警示信息可能导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **警告:** 重启操作将导致业务中断，恢复业务所需时间约10分钟。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 **说明:** 您也可以通过按Ctrl + A选中全部文件。
>	多级菜单递进。	设置 > 网络 > 设置网络类型
粗体	表示按键、菜单、页面名称等UI元素。	单击 确定。
`courier` 字体	命令。	执行 `cd /d C:/windows` 命令，进入Windows系统文件夹。
*## *	表示参数、变量。	`bae log list --instanceid` *`Instance_ID`*
[]或者[a|b]	表示可选项，至多选择一个。	`ipconfig` *`[-all|-t]`*
{}或者{a|b}	表示必选项，至多选择一个。	`swich` *`{stand | slave}`*

目录

1 设备接入

物联网安全运营中心Link Security Operations Center（简称SOC）支持多种物联网设备操作系统，为设备提供不同等级的保护。

在Android和Linux设备上，可以使用设备保护服务Device Protection Service（简称 DPS），实时检测入侵，及时修复漏洞。在AliOS Things或其它RTOS设备上，可以使用设备取证服务Device Attestation Service（简称 DAS），持续掌握设备的安全状况。



开通服务

设备接入SOC，首先需要在阿里云注册账号，登录并开通SOC服务，在SOC的设备管理的安全基线页面内前往创建产品或设备。将产品接入平台的凭配置到设备端的应用中，并集成相应的DPS或DAS开发包，当设备上线时，就能在SOC中看见并管理设备。

选择开发包

设备上启用DPS或DAS服务，要先选择和设备系统相应的SDK开发包。

设备端	版本号	发布时间	使用环境	下载链接	版本说明
DAS	1.0.1	2018-12-25	RTOS	das-1.0.1-20181225.tgz	· 上报运行状态 · 定期检测系统完整性

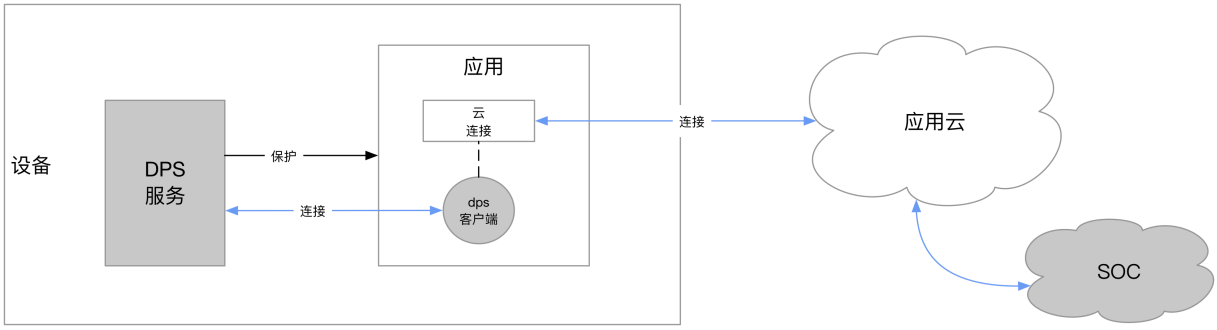
设备端	版本号	发布时间	使用环境	下载链接	版本说明
DPS	1.1.2（beta）	2019-07-19	Android 5/7/8（ARM/ARM64）	dps_sdk_1.1.2_android_20190719.tgz	· 上报运行状态 · 定期检测系统完整性 · 安全基线生成和管理 · 漏洞检测/修复 · 设备行为检测 · 风险检测/处置 · 安全运营托管
DPS	1.1.2（beta）	2019-07-20	Linux（x64/ARM64/ARMHF）	dps_sdk_1.1.2_linux_20190720.tgz	· 上报运行状态 · 定期检测系统完整性 · 安全基线生成和管理 · 漏洞检测和修复 · 设备行为检测 · 风险检测和处置 · 安全运营托管

注意:

- （beta）表示：未经充分测试验证存在未知风险。因此，beta版仅用于在实验环境测试或熟悉功能用，禁止用于生产环境。
- 获取支持Docker环境的DPS，请联系我们咨询。

集成开发

DPS/DAS运行时需要设备有上云通道和安全运营中心互动消息和控制，SDK中提供的DPS/DAS客户端API使设备应用将原有上云通分享给DPS使用，无需另外建立网络连接。详细说明请参见后面两篇SDK介绍文档。



DAS在适配AliOS Things之外的其它RTOS系统时，需要适配DAS中设备相关的基础功能函数。

安装配置

DPS本身是自包含服务，不需要额外的编译集成，并且有预置的上云通道。如果设备允许多个上云连接存在，则可以选择直接使用DPS开发包进行安装。按照SDK介绍文档内容配置具体的服务参数和上云参数后即可运行。

2 设备保护服务SDK

2.1 概览

设备保护服务Device Protection Service（以下简称为 DPS）， 是为Linux、Android等智能设备提供的安全保护服务。DPS不仅可以提供实时的入侵检测，还负责及时修复安全问题，使设备受到持续的安全防护。

下载SDK

版本号	发布日期	目标设备	下载
DPS 1.1.2（beta）	2019-07-19	Android 5/7/8（ARM/ARM64）	dps_sdk_1.1.2_android_20190719.tgz
DPS 1.1.2（beta）	2019-07-20	Linux（x64/ARM64/ARMHF）	dps_sdk_1.1.2_linux_20190720.tgz

注意:

（beta）表示：未经充分测试验证存在未知风险。因此，beta版仅用于在实验环境测试或熟悉功能用，禁止用于生产环境。

2.2 Linux系统集成

本章主要介绍在Linux系统上集成开发的操作。

开发环境

当前Linux X86-64 SDK需在有systemd为init的系统上开发。

开发包下载完成后，将其解压到目标设备开发环境所在机器上的任意目录。再根据目标设备开环境的具体要求，将SDK目录引入到设备的开发环境中。

编译设置

DPS开发包提供标准的Makefile编译脚本，以适配一般Linux设备的编译环境。其中差异化的配置可以在SDK包中的*config.mk*文件中指定。

- 硬件体系

```
ARCH := x86_64
```

DPS SDK目前支持Armhf，Arm64，x86_64三个目标平台架构。

- DPS_DATA

```
DPS_DATA := /data/dps
```

指定DPS的data目录，该目录主要用来存放基线规则，以及客户端在运行过程中产生的临时文件等。

缺省值为*/data/dps*，请确保*/data*目录或者相应分区可读、可写。

- DPS_PROFILE

```
DPS_PROFILE := standard
```

DPS类型，取值为standard、server、lite中的一种。当开发板的CPU性能较差时，可选择lite模式。

缺失值为standard。

- PROTECTED_PATH

```
PROTECTED_PATH := /home/dps/my_important_dir
```

scanner默认扫描常规系统目录。如果有用户需要自定义scanner额外的扫描路径，可在这里指定。

- SYSTEM_VERSION

```
SYSTEM_VERSION :=  version://textfile:/etc/YOUR_VERNO_FILE:.+
SYSTEM_VERSION :=  version://string:ver.1.0.0
```

若选择textfile比对方式，则指定厂商固件版本号寄存的文本文件路径，例如*/etc/YOUR_VERNO_FILE*，按照定义的正则表达式从该文件路径中解析版本号，例如`.+`正则表达式。如果选择string比对方式，那么直接使用接续字串当做固件版本号，例如`ver.1.0.0`。

- MANAGED_ID

```
MANAGED_ID :=  managedid://textfile:/etc/MANAGED_ID:.+
MANAGED_ID :=  managedid://string:15709823
```

MANAGED_ID同SYSTEM_VERSION，也可选择字串和档案正则表达方式。

- 客户端选项

  客户端主要用来连接云端，接收和发送云端指令。相关配置选项在SDK包的*config.mk*中指定。

  DPS连云提供了两种方式，一种是直接使用SDK包里的dps_client客户端程序，可通过设备证书直接上云；另一种是使用SDK包中提供的Makefile脚本编译*libclient.a*，然后将*libclient.a*集成到自己的项目中，通过库导出的接口连接云端。

  使用独立DPS客户端：

  1. 打开相应开关。

     ```
     CONFIGURE_STANDALONE_CLIENT := 1
     ```

     说明：

     配置CONFIGURE_STANDALONE_CLIENT为1，则使用SDK包中自带的独立DPS客户端，可通过设备证书信息直接与云端建立连接。

  2. 配置设备证书（ProductKey、DeviceName、DeviceSecret）。

     ```
     PRODUCT_KEY := ProductKey
     DEVICE_NAME := DeviceName
     DEVICE_SECRET := DeviceSecret
     ```

     说明：

     当且仅当CONFIGURE_STANDALONE_CLIENT为1时，才需要配置设备证书，否则配置不生效。

  编译客户端开发库：

  1. 打开相应开关。

     ```
     CONFIGURE_CLIENT_LIBRARY := 1
     CONFIGURE_IOTX_LIBRARY   := 0
     CONFIGURE_BUILD_CLIENT   := 0
     ```

     说明：

配置 CONFIGURE_CLIENT_LIBRARY 为1，则需要编译 DPS 客户端开发库 libclient.a ，供自有客户端项目集成使用。客户端API的具体使用方式请参见客户端API。

另外， CONFIGURE_IOTX_LIBRARY开关用来编译Linkkit开发库。该库主要用来最终编译演示版dps_client客户端程序之用，用户在实际项目中自行替换成最新的Linkkit开发库即可。客户端开发库*libclient.a*内部封装了Linkkit开发库导出的接口。

CONFIGURE_BUILD_CLIENT开关用来编译完整的演示版dps_client客户端程序，其主要功能是用来演示如何将客户端开发库*libclient.a*集成进自有的客户端项目中。

2. 指定ToolChain安装路径。

   目标平台toolchain安装路径在*toolchains/config.mk*文件中指定。

   ```
   TOOLCHAIN_INSTL_DIR := $(HOME)/your_toolchain_path_dir
   ```

3. 编译*libclient.a*。

   ```
   make libs
   ```

   如果上述配置都没有问题，那么最终编译出来的库会在会放在*output/lib_a/libclient.a*。

   最后，将编译出来的*.a*库集成到自己的工程中即可。客户端API的具体使用方式请参见客户端API。

- 其他组件选项

  ```
  CONFIGURE_SANDBOX_SERVICE   := 0
  ```

  DPS沙箱服务，主要用来限制进程的行为和权限，防止病毒作恶。具体使用方式参考DPS Sandbox相关文档。

- 安装路径

  SDK中Makefile的主要工作是将DPS各组件释放到目标image的rootfs根目录下，并自动创建相关文件夹。

  那么，上述所有配置完毕之后，就可以将DPS SDK解压并释放到指定的安装目录中了。安装目录可以在SDK包的*config.mk*中指定。

```
prefix ?= /device_root_path
```

  也可以在安装的时候通过参数形式指定。

```
make install prefix=/device_root_path
```

内核配置

开发者如果需要为目标设备编译Linux Kernel，请确保Linux Kernel版本在3.15及以上。并且需要检查在Kernel编译的defconfig选项中使能了以下项目。

- 开启AUDIT和NETLINKE

```
CONFIG_AUDIT_ARCH=y
CONFIG_AUDIT=y
CONFIG_HAVE_ARCH_AUDITSYSCALL=y
CONFIG_AUDITSYSCALL=y
CONFIG_AUDIT_WATCH=y
CONFIG_AUDIT_TREE=y
CONFIG_NF_CT_NETLINK=y
CONFIG_NF_CT_NETLINK_TIMEOUT=y
CONFIG_NETFILTER_NETLINK_GLUE_CT=y
```

- 开启NETFILTER和IPTABLES

```
CONFIG_NETFILTER=y
CONFIG_NETFILTER_NETLINK=y
CONFIG_NETFILTER_NETLINK_ACCT=y
CONFIG_NETFILTER_NETLINK_QUEUE=y
CONFIG_NETFILTER_NETLINK_LOG=y
CONFIG_IP_NF_IPTABLES=y
```

- 开启XTABLES及相关选项

```
CONFIG_NETFILTER_XTABLES=y

#
# Xtables combined modules
#
CONFIG_NETFILTER_XT_MARK=y
...

#
# Xtables targets
#
CONFIG_NETFILTER_XT_TARGET_AUDIT=y
...
```

```
#
# Xtables matches
#
CONFIG_NETFILTER_XT_MATCH_BPF=y
...
```

编译示例

DPS模块提供头文件和静态库供设备应用集成使用。AppSample为实现参考。

- `$(DPS_SDK)/client/inc/DPSClient.h`

  DPS client header file

- `$(DPS_SDK)/output/lib_a/libdpsclient.a`

  DPS client static library

- `$(DPS_SDK)/output/lib_a/libiotx.a`

  Link MQTT static library

应用Makefile示例：

```
CPPFLAGS += -I$(DPS_SDK)/client/inc
LDFLAGS += -L$(DPS_SDK)/output/libdpsclient.a
LDFLAGS += -L$(DPS_SDK)/output/libiotx.a
```

启动设置

正常情况下，DPS SDK通过Makefile install预装入image。操作系统启动的过程中会将DPS SDK拉起，无需额外配置。

systemd启动方式如下步骤所示。

1. 在编译目标系统Image之前，在目标系统的*/usr/lib/systemd/system/*目录下创建*dpsd.service*文件。

   ```
   [Unit]
   Description=dps deamon server
   # After=network.target

   [Service]
   ExecStart=/system/dps/bin/dpsd
   Type=simple
   User=root
   Group=root
   KillMode=process
   Restart=always
   RestartSec=10s

   [Install]
   WantedBy=multi-user.target
   ```

```
Alias=dpsd.service
```

2. 在*/etc/systemd/system/multi-user.target.wants/*目录下创建软连接，指向*/usr/lib/systemd/system/dpsd.service*文件。

2.3 Android系统集成

本章主要介绍在Android系统上集成开发的操作。

编译设置

1. 设置DPS-SDK。

   针对Android设备的编译环境，将SDK目录用软链接`ln -s`的方式放入Android源代码的*external*目录下，并修改*platform/device*目录的相应产品目录下的*device.mk*，增加如下一行：

   ```
   $(call inherit-product-if-exists,external/dps-sdk/dps_sdk.mk)
   ```

2. 部署安全策略。

   需要手动加入 DPS 对部分原生 Android 服务的 sepolicy 和 seccomp 补丁，具体方法为手动执行`external/dps-sdk/policy/update_sepolicy.sh [android_home]`和`external/dps-sdk/policy/update_seccomp_policy.sh [android_home]`。

   其中，`[android_home]`代表Android源码跟路径。

   如果在集成开发环境当中出现因为SELinux Policy阻碍编译以及在编译之后运行过程中产生SELinux permissive审计日志，请根据*external/dps-sdk/policy/sepolicy/*当中的`dps_file_contexts`和*dps.te*文件进行SELinux策略的补充和调整。

3. 编译镜像。

   准备就绪之后，对Android产品镜像进行正常编译，DPS就可以随产品镜像一同生成了。

   高阶的DPS配置可以在*external/dps-sdk/dps_sdk.mk*中进行具体的控制。

内核配置

开发者如果需要为目标设备编译Linux Kernel，请确保Linux Kernel版本在3.15及以上。并且需要检查在Kernel编译的defconfig选项中使能了以下项目。

- 开启AUDIT和NETLINKE

```
CONFIG_AUDIT_ARCH=y
CONFIG_AUDIT=y
CONFIG_HAVE_ARCH_AUDITSYSCALL=y
CONFIG_AUDITSYSCALL=y
CONFIG_AUDIT_WATCH=y
CONFIG_AUDIT_TREE=y
```

```
CONFIG_NF_CT_NETLINK=y
CONFIG_NF_CT_NETLINK_TIMEOUT=y
CONFIG_NETFILTER_NETLINK_GLUE_CT=y
```

- 开启NETFILTER和IPTABLES

```
CONFIG_NETFILTER=y
CONFIG_NETFILTER_NETLINK=y
CONFIG_NETFILTER_NETLINK_ACCT=y
CONFIG_NETFILTER_NETLINK_QUEUE=y
CONFIG_NETFILTER_NETLINK_LOG=y
CONFIG_IP_NF_IPTABLES=y
```

- 开启XTABLES及相关选项

```
CONFIG_NETFILTER_XTABLES=y

#
# Xtables combined modules
#
CONFIG_NETFILTER_XT_MARK=y
...

#
# Xtables targets
#
CONFIG_NETFILTER_XT_TARGET_AUDIT=y
...

#
# Xtables matches
#
CONFIG_NETFILTER_XT_MATCH_BPF=y
...
```

编译示例

DPS模块提供头文件和静态库供设备应用集成使用。AppSample为实现参考。

- `external/dps/client/inc/DPSClient.h`

  DPS client header file

- `libdspclient.a`

  DPS client static library

- `libiotx.a`

  Link MQTT static library

*Android.bp*示例：

```
cc_binary {
  name: "AppSample",
  include_dirs: [
    "external/dps/client/inc",
    ...
  ],
```

```
  srcs: [
    "AppSample.cpp",
    ...
  ],
  static_libs: [
    "libiotx",
    "libdpsclient",
    ...
  ],
}
```

启动设置

对于Android 7以上版本，目前DPS在Android设备上通过集成编译即可自启动，无需额外配置。

对于Android 5，需要在系统的*init.rc*或者厂商自定义的*init.<vendor>.rc*当中声名DPS服务。声名方式如下。

```
service dpsd /system/bin/dps/bin/dpsd
    class main
```

2.4 服务配置

本章主要介绍设备保护服务SDK的服务配置。

DPS服务的配置文件位于SDK的packages目录下，文件名为*configure.ini*，此配置文件保存DPS服务程序的基本配置，其中：

- DPS_DATA：DPS服务器程序存放数据的路径，请配置到操作系统的可读/写分区的文件夹路径下。
- SYSTEM_VERSION：用以配置设备固件版本。配置方法如下所示。
  - 如果设备厂商始终使用本地文件系统来追溯固件版本，那么请选择textfile比对方式。例如：

    ```
    SYSTEM_VERSION=version://textfile:/etc/YOUR_VERNO_FILE:.+
    ```

    其中*/etc/YOUR_VERNO_FILE*代表厂商固件版本号寄存的文本文件路径，`.+`代表正则表达式，用以从该文件路径中解析版本号。
  - 如果想使用字串当做固件版本号，请选择string比对方式。例如：

    ```
    SYSTEM_VERSION=version://string:ver.1.0.0
    ```

    其中`ver.1.0.0`即为版本号。

    注意：

    固件版本号会影响在目标设备进行固件升级之后需要重新取证等功能，请务必在固件开发过程中根据即将发布的固件版本修改此配置项，否则将对目标设备造成严重的后果。

- PROTECTED_PATH：目标设备取证扫描的自定义路径，在此路径之下的所有文件将被取证，建议配置成目标设备上可执行程序、共享库、配置文件等重要文件存放的目录。
- DPS_DEBUG，NUM_LOGS 和 LOG_FILE_SIZE：DPS调试相关选项。配置方法如下所示。
  - DPS_DEBUG表示DPS的调试开关，默认为关闭0，如果需要打开，请配置成1。
  - NUM_LOGS表示DPS调试开关在打开的情况下，DPS旧日志保存的数目，默认为3个。
  - LOG_FILE_SIZE表示DPS调试开关在打开的情况下，每个DPS日志文件的最大尺寸，默认为10 MB。

*configure.ini*示例：

```
[system]
DPS_DATA=/data/dps
DPS_PROFILE=server
DPS_DEBUG=1
NUM_LOGS=3
LOG_FILE_SIZE=10240

[attestation]
PROTECTED_PATH=

[device]
SYSTEM_VERSION=version://textfile:/etc/FW_VERSION:.+
MANAGED_ID=managedid://textfile:/etc/MANAGED_ID:.+
```

2.5 客户端API

本章主要介绍设备保护服务SDK的客户端API。

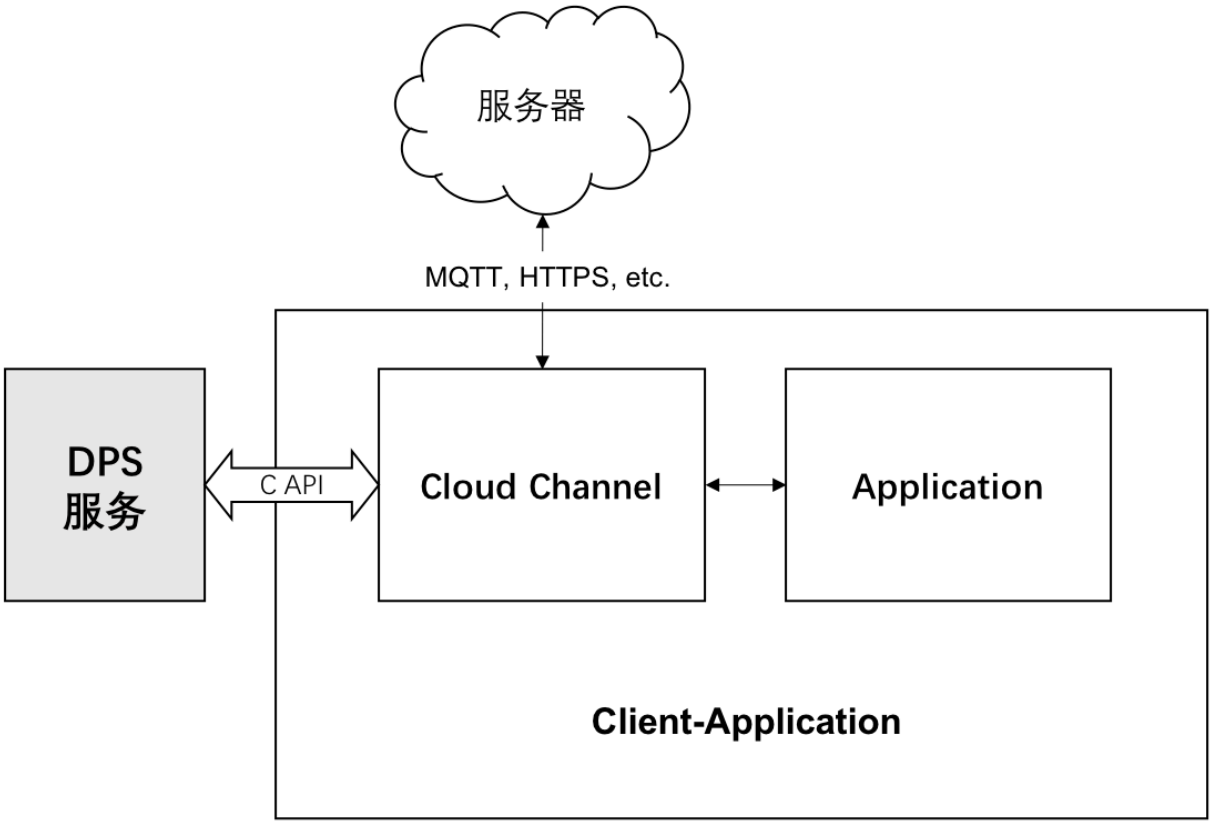
概述

DPS是自包含服务，不需要额外的开发和编译，SDK含预编译客户端可与安全运营中心互动消息和控制。此外，SDK依然提供客户端API使设备应用将自有上云通分享给DPS使用，无需另外建立网络连接。

客户端API有两种集成方法，C版本API和C++版本API。基础C版本API让DPS复用应用上云通道；C++版本API则抽象化DPS的接入方式，使其他应用能模仿DPS与应用对接，分享上云通道。

C版本API

C版本API架构图如下所示。



C版本API让DPS服务复用上云通道，上云通道为具备有和服务器连接能力的协议。C版本API可参考的SDK文件如下。

- *include/dps_core.h*：列出应用对接DPS所需接口。
- *example/client-sample/main.c*：使用*dps_core.h*配合Linkkit MQTT上云完成的应用范例。

应用透过dps_init初始化与DPS服务的通讯句柄。

```
void* dps_init(const char *product_key, const char *device_key);
```

应用透过dps_connection注册消息处理器，此处理器当DPS消息到来时被调用，您可在处理器中将DPS消息发送往服务器，或者存到队列等待后续发送。

```
typedef bool (*pub_handle)(const char *, const char *, int , void *);

void dps_connection(void *session, pub_handle pub_topic, void *context
);
```

当您确认上云通道一切就绪时，即可透过dps_on_connected告知DPS，此时DPS服务即可使用此通道将消息发往服务器端，抵达安全运营中心。

```
bool dps_on_connected( void *session);
```

当您发现有来自安全运营中心的消息，即可透过dps_on_message直接将消息传送给DPS服务。

注意:

此API需在dps_on_connected调用之后，确保DPS服务认为上云通道畅通。

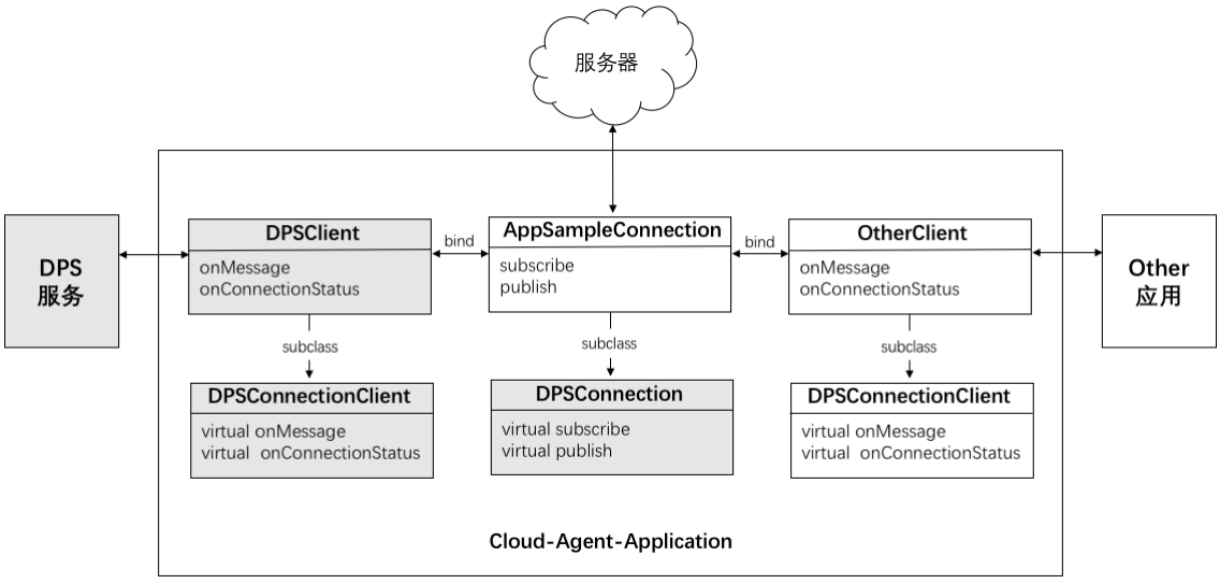
```
bool dps_on_message( void *session, const char *message, int length);
```

终止服务。

```
void dps_final(void *session)
```

C++版本API

C++版本API使各种应用能模仿DPS与应用对接的方式，共同分享上云通道。



C++版本API可参考的SDK文件如下。

- **include/DPSClient.h：以下做说明。**
- **example/cloud-agent/*：使用 DPSClient.h 配合 Linkkit MQTT 上云完成的应用范例。**

DPSConnection类：

DPS用来上云的消息通道抽象接口。应用需要继承并实现此接口，使DPS客户端可以复用应用的上云通道。

```
enum DPSConnectionStatus
{
  DPS_CONNECTION_DISCONNECTED = 0,
  DPS_CONNECTION_CONNECTED,
};

class DPSConnectionClient
{
public:
  virtual ~DPSConnectionClient();
```

```
  virtual bool onMessage( const char *message, int length) = 0;
  virtual bool onConnectionStatus(  DPSConnectionStatus status) = 0;
};

class DPSConnection
{
public:
    virtual ~DPSConnection();

    virtual bool subscribe(const char * topic) = 0;
    virtual bool publish(const char *topic, const char *event, int
length) = 0;
};
```

DPSClient类：

应用和DPS客户端交互的接口。

```
class DPSClient : public DPSConnectionClient
{
public:
  DPSClient( const char *product_key, const char *device_key);

  static DPSClient *create(const char *product_key, const char *
device_name);

  /* DPSConnectionClient implements */
  bool onMessage( const char *message, int length);
  bool onConnectionStatus( DPSConnectionStatus satus);
};
```

代码示例如下所示。

- *AppSampleConnection.h*

  ```
  #include "DPSClient.h"
  class AppSampleConnection : public DPSConnection {
  public:
    AppSampleConnection( DPSClient *client);

    bool subscribe(const char * topic);
    bool publish(const char *topic, const char *event, int length);

  private:
    DPSClient *mClient;
    enum ConnectionState {
      CONNECTED,
      CLOSED,
    } mState;
    ...
  };
  ```

- *AppSampleConnection.cpp*

  ```
  #include "AppSampleConnection.h"
  /**
   * DPS publishs events to the cloud.
   * /
  bool AppSampleConnection::subscribe(const char *topic)
  {
  ```

```
...
}

/**
 * DPS subscribes a event topic.
 * /
bool AppSampleConnection::publish(const char *topic, const char *
event, int length)
{
...
}
```

1. **创建DPSClient实例，并开始服务。**

```
#include "DPSClient.h"
const char *product_key = "sample";
const char *device_name = "test-1";
DPSClient *client = DPSClient.create(product_key, device_name);

AppSampleConnection *connection = new AppSampleConnection(client);
client.start( connection);

// MQTT messaging loop
...
```

2. **分发处理DPS消息。**

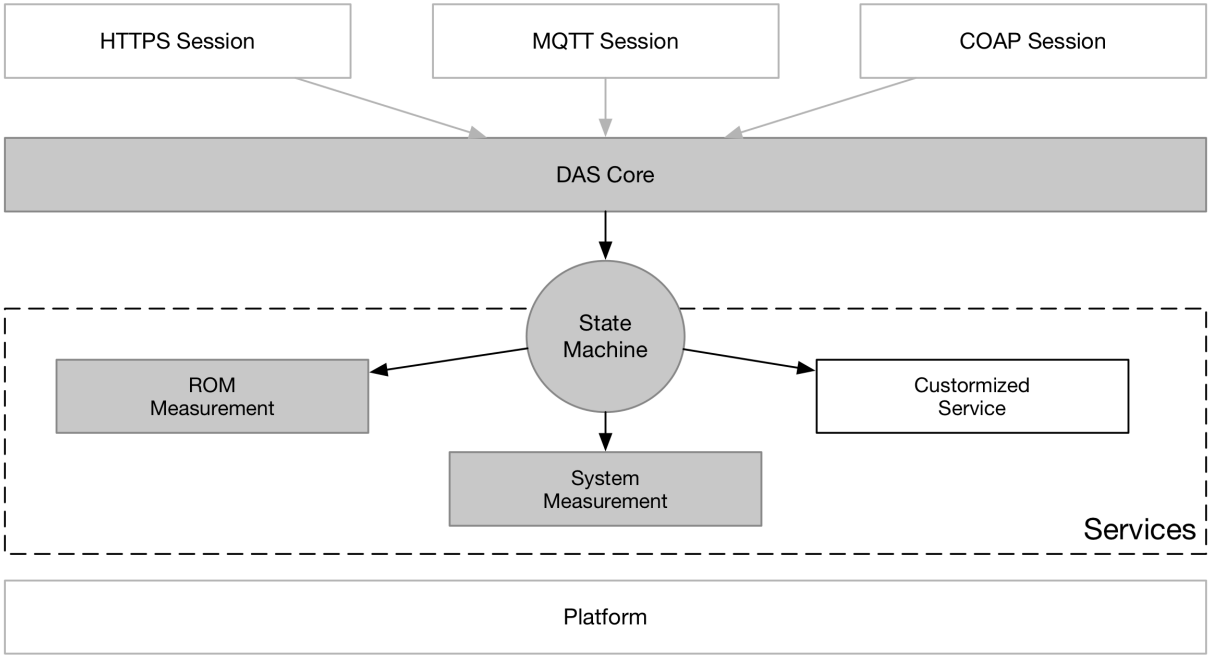
```
char message[DPS_MESSAGE_MAX_LENGTH];
...
// Application received a new message with length
...
mClient->onMessage(message, length);
```

3 设备取证服务SDK

3.1 概览

设备取证服务Device Attestation Service（以下简称为DAS），是为运行RTOS的物联网节点设备提供的安全审计服务。DAS会实时上报运行状态，定期检查系统的完整性，让管理者随时了解设备的安全状况，及时发现、排查和修复安全问题。

DAS分Core、Platform、Service 三个层次提供应用集成，平台移植和服务扩展功能，旨在方便各种设备利用SOC服务一站式保护物联网终端节点安全。



下载SDK

设备端	版本号	发布时间	使用环境	下载链接	版本说明
DAS	1.0.1	2018-12-25	RTOS	das-1.0.1-20181225.tgz	· 上报运行状态 · 定期检测系统完整性

开发环境

DAS是以开源代码方式提供，用户无需额外配置开发环境，只需要根据目标系统的开发要求将SDK解压到特定位置即可。

DAS同时是作为AliOS Things的一个安全组件发布的，代码位于`/security/das`目录下，开发者可以直接使用。

3.2 应用集成

在一般设备的实际开发中，由于资源受限，每个设备网络连接的数目是受限的。DAS充分考虑到这点，在核心层只定义了DAS相关数据的订阅和分发接口，设备可以根据实际的网络会话上对接DAS服务。

初始化核心服务

```
void* das_init(const char *product_name, const char *device_name);
```

设置上下行消息主题

```
const char* das_pub_topic(void *session, const char *topic);

const char* das_sub_topic(void *session, const char *topic);
```

说明:

如果不单独设置主题，DAS使用缺省主题。

配置网络连接

```
typedef int (*publish_handle_t)(const char *topic,
                const uint8_t *message, size_t msg_size,
                void *channel);

void das_connection(void *session,
          publish_handle_t publish_handle,
          void *channel);
```

更新连接状态

```
void das_on_connected(void *session);

void das_on_disconnected(void *session);

void das_on_message(void *session, const uint8_t *message, size_t
msg_size);
```

步进驱动取证服务

```
das_result_t das_stepping(void *session, uint64_t now);
```

说明:

如果适配定时器函数，则DAS服务缺省由定时器任务驱动。

终止核心服务

```
void das_final(void *session);
```

综合示例代码

本示例是基于阿里云Link Kit MQTT的代码片段，完整代码见DAS开发库MQTT示例。

```
#include "iot_import.h"

#define PRODUCT_KEY     "demo_das_product"
#define DEVICE_NAME   "demo_das_device_1"

static void *session = null;

static int _on_publish( const char *topic, uint8_t *msg, uint32_t size
,  void *mqtt)
{
  iotx_mqtt_topic_info_t topic_msg;
  topic_msg.qos = IOTX_MQTT_QOS1;
  topic_msg.payload = (void *)message;
  topic_msg.payload_len = length;
  return IOT_MQTT_Publish( mqtt, topic, &topic_msg);
}

static void on_message( void *handle, void *pclient, iotx_mqtt_
event_msg_pt msg)
{
  das_on_message( session, msg.payload, msg.payload_len);
}

int main( int argc, const char argv[][])
{
  const char *sub_topic;

  mqtt = IOT_MQTT_Construct( &mqtt_params);

  session = das_init( PRODUCT_KEY, DEVICE_NAME);

  das_connection(session, _on_publish, mqtt);

  das_on_connected(session);

  sub_topic = das_sub_topic(session, NULL);

  IOT_MQTT_Subscribe( mqtt,
      sub_topic, IOTX_MQTT_QOS1, on_message, session);

  while (IOT_MQTT_Yield(mqtt, 200) != IOT_MQTT_DISCONNECTED) {
    ...
    das_stepping(session, time(NULL));
    ...
  }

  das_on_disconnected(session);
  das_final(session);

  return 0;
```

```
}
```

3.3 平台、设备与服务适配

本章主要介绍设备取证服务的平台适配、设备适配和ROM度量服务适配。

平台适配

DAS仅使用到少数几个libc标准函数（strlen、memcpy、memset）和数据类型定义（size_t、uint8_t、uint32_t、uint64_t）。

如果设备平台没有相关实现或有不同的定义，可以通过在Makefile中设置CONFIG_DAS_PLATFORM_ALT替代头文件，然后在替代头文件中实现或重定义DAS所需的函数和类型。

Makefile示例如下：

```
CONFIG_DAS_PLATFORM_ALT = platform_alt.h
```

设备适配

设备适配的函数定义在*inc/das/hardware.h*文件中。

- 获取设备固件版本

```
size_t das_hal_firmware_version(char *buf, size_t size);
```

- 获取设备硬件标识

```
size_t das_hal_device_id(char *buf, size_t size);
```

ROM度量服务适配

作为DAS缺省的安全度量服务，ROM Service完成设备代码完整性的检查功能，需要设备适配ROM（txt段）的分段信息。

```
typedef struct _das_rom_bank {
  uint8_t* address;
  size_t size;
} das_rom_bank_t;

int das_hal_rom_info(das_rom_bank_t banks[DAS_ROM_BANK_NUMBER]);
```

ROM度量服务可以通过修改*inc/das/configure.h*中DAS_SERVICE_ROM_ENABLED的定义来禁用。

3.4 服务扩展

除了使用DAS自带的安全度量服务以外， 您还可以根据设备的特点和需要在DAS服务框架内构建自己的度量服务。

实现服务

在*src/service*下创建*service_skeleton.c*文件，并实现*inc/das/service.h*中的服务接口定义。

```
das_result_t skt_info (char *buffer, size_t size,
             das_service_state_t *state);
das_result_t skt_attest (char *path, size_t size, das_sum_context_t *
sum_context,                  das_service_state_t *state);
das_result_t skt_measure (das_sum_context_t *sum_context,
              das_mac_context_t *mac_context,
              das_service_state_t *state);

das_service_t skt_service = {
          .name = "skeleton",
          .info = skt_info,
          .attest = skt_attest,
          .measure = skt_measure,
        };
```

添加服务

在*src/service/service.c*文件中添加服务。

```
#if DPS_SERVICE_SKELETON_ENABLED
extern das_service_t skt_service;
#endif

das_service_t * das_service_table[DAS_MAX_SRV_NUM] = {
#ifdef DPS_SERVICE_SKELETON_ENABLED
     &skt_service,
#endif
  &sys_srv,
  NULL,
};
```

配置服务

配置Makefile使能服务。

```
CFLAGS += -DDPS_SERVICE_SKELETON_ENABLED
SRCS   += service/service_skeleton.c
```