中软国际公司内部技术规范

Shell语言安全编程规范

中软国际科技服务有限公司

修订声明

参考:华为 Shell 语言安全编码规范。

0.

日期	修订版本	修订描述	作者
2017-07-28	1.0	整理创建初稿	陈丽佳
2017-08-02	1.1	统一整理格式，并修正部分错误	陈丽佳

目录

概述

前言

《Shell 语言安全编码规范》针对 bash 语言编程中的数据校验、加密与解密、脚本执行、目录&文件操作等方面，描述可能导致安全漏洞或风险的常见编码错误。该规范基于业界最佳实践，并总结了公司内部的编程实践。该规范旨在减少 SQL 注入、敏感信息泄露、环境变量攻击、临时文件攻击、社会工程学等方面的安全问题发生。

使用对象

本规范的读者及使用对象主要为使用 bash 脚本的研发人员和测试人员等。

适用范围

该规范适用于基于 bash 脚本开发的最终交付外部客户的产品。

术语定义

规则：编程时必须遵守的约定。
建议：编程时必须加以考虑的约定。
说明：某个规则的具体解释。
错误示例：对此规则/建议从反面给出例子。
推荐做法：对此规则/建议从正面给出例子或建议。
例外情况：相应的规则不适用的场景。
信任边界：位于信任边界之内的所有组件都是被系统本身直接控制的。所有来自不受控的外部系统的连接与数据，包括客户端与第三方系统，都应该被认为是不可信的，要先在边界处对其校验，才能允许进一步与本系统交互。
非信任代码：非产品包中的代码，如通过网络下载到本地虚拟机中加载并执行的代码。

1. 通用原则

规则 1.1 校验跨信任边界传递的不可信数据

说明：软件最为普遍的缺陷就是没有对来自外部环境（系统/程序信任域之外）的数据进行合法性校验，校验缺失可导致系统的各种缺陷，比如 DoS、SQL 注入等，所以对于来自外部环境的输入，默认为不可信，需执行合法性风险，校验必须在信任边界以内进行（如对于 Web 应用，需在服务端做校验）。

不可信数据可能来自（包括但不局限于）：

- 用户输入
- 外部接口输入，如网络数据

数据校验可以参考如下方法（包括但不局限于）：

- 脚本的入参一定要校验其合法性

1、调用跨信任边界的入口脚本时，参数必须用双引号引起来，避免解析异常甚至引起注入攻击。

2、必须校验入口参数个数及“白名单”方式净化参数。

- “白名单”方式净化

该方法作为正向校验，包括输入数据的白名单字符范围校验和业务合法性校验。

1、白名单字符范围校验就是删除、编码或替换系统不能接受的字符/数据，只接受某个确定的数据列表或已知好的数据，确保数据属于一个可接受的数据集合。

2、业务合法性校验需要根据具体业务场景做合法校验。

比如如果系统期望接收一个电话号码，则可以剔除掉输入数据中所有的非数字字符，所以“(555)123-1234”，“555.123.1234”与“555\";DROP TABLE USER;--123.1234”这三个数据处理后，结果都是“5551231234”。

如下代码确保参数“name”只接受字母、数字以及下划线且必须以字母开头。

```
function fn_check_name()
{
    local name=$1

    # 增加参数校验
    if [[ ! "${name}" =~ ^[A-Za-z][A-Za-z0-9_]*$ ]]
    then
        # 出错处理
        return 1
    else
        # 业务处理
        return 0
    fi
}
```

- “黑名单”方式净化

该方法为确保输入数据“安全”，必须实时维护一个系统不可接受的数据列表（比如字符或模式）,即“黑名单”，否则列表中的数据会很快过时，系统安全性会降低。因为大部分输入域有其特殊的业务或格式要求，所以“黑名单”通常也不全面。相对为应对当前和未来所有攻击方式而使用一个复杂、迟钝的“黑名单”校验程序，针对正确输入的“正向”校验显得更加简单、高效。

规则 1.2 禁止直接使用不可信数据来拼接 SQL 语句

说明：SQL 注入是指原始 SQL 操作被动态更改成一个与程序预期完全不同的操作。执行更改后 SQL 可能导致信息泄露或数据被篡改。防止 SQL 注入主要需对不可信数据进行校验。推荐对不可信数据做如下校验以防止 SQL 注入：

- 对于字符型字段，需对输入数据做转义

  1、 将单引号替换为斜杠+单引号。

  2、 如果字符型输入数据作为 SQL 语句中 like 部分，需参考附录 A 做转义

  3、 系统根据具体业务场景，确定需要校验的特殊字符，特殊字符可以参考附录 A 和附录 B。

- 对于整型字段，需对输入数据进行类型检测，拒绝传入不合法数字。

错误示例 1：该例中直接使用输入数据拼接 SQL 语句，没有对输入数据做校验，存在 SQL 注入风险。当 username 输入值为：tom' OR '1' = '1，SQL 语句变成：SELECT * FROM db_user WHERE username='tom' OR '1'='1' AND password=<PASSWORD>，便绕开了对口令的验证。

```
function fn_get_dbuser()
{
    local username=$1
    read -s pvalue

    local sql="SELECT * FROM db_user WHERE username = '${username}' AND
password = '${pvalue}'"    # 没有校验输入参数就直接拼装成 SQL 并使用
    # 业务逻辑处理
}
```

推荐做法：

```
function fn_get_dbuser()
{
    local username=$1
    read -s pvalue

    # 增加参数校验
    if [[ ! "${username}" =~ ^[A-Za-z][A-Za-z0-9_]*$ ]]
    then
        # 出错处理
```

```
    fi
    # 增加参数校验
    echo "${pvalue}" | grep "^.\{8,30\}$" | grep "[a-z]" | grep "[A-Z]"
| grep "[0-9]" | grep "[~@#$%^&*()_=+]" &> /dev/null
    if [$? -ne 0 ]
    then
        # 出错处理
    fi
    local sql="SELECT * FROM db_user WHERE username = '${username}' AND
password = '${pvalue}'"

    # 业务逻辑处理
}
```

规则 1.4 禁止在 shell 脚本中保留敏感信息

说明：敏感信息说明见规则 1.3，shell 脚本在产品发布版本中以文本方式存在，如果脚本中包含敏感信息，可能会被恶意利用造成安全隐患。比如脚本注释中有员工个人信息，虽然不违反安全红线，但带有员工个人信息的注释可能会泄露具体的开发人员信息，从而引入社会工程学方面的风险，因此要从脚本注释中删除员工个人信息。
脚本中的敏感信息包括（但不限于）：

- 密钥、密码明文和密文：如果脚本中存在这类信息，一旦泄露，与其相关的数据的机密性将受到严重影响，因此不能在脚本中体现出来。
- 公司内部信息：如代码作者的工号、姓名、联系方式（手机号、Email、地址）等

错误示例 1：如下代码中，硬编码了默认的密码，在执行时会导致密码这类敏感信息很容易暴露出来。

```
#!/bin/bash

#业务逻辑

echo -n "please input your password: "
read -s pvalue
if [ -z "${pvalue}" ]
then
    pvalue="Changeme_123"
fi

#业务逻辑
```

推荐做法： shell 脚本里禁止硬编码密码等敏感信息，需要密码时通过手工输入

```
#!/bin/bash

#业务逻辑
```

```
while true
do
    echo -n "please input your password : "
    read -s pvalue
    [ -z "${pvalue}" ] && continue || break
done
```

```
#业务逻辑
```

例外情况：在系统初始安装的 shell 脚本中可以使用默认密码密文，但需要在调测及管理员操作指南里提醒用户及时修改缺省密码。

错误示例 2：该例中，发布版本的 shell 脚本中包含了公司内部信息，可能导致社会工程学方面的风险。

```
################################################################
#   FUNCTION   : Log
#   CREATED BY  : XXXXX（作者的工号，姓名等）
################################################################
function Log()
{
……
}
```

推荐做法：脚本中清除开发人员个人信息

```
################################################################
#   FUNCTION   : Log
################################################################
function Log()
{
……
}
```

规则 1.3 禁止在界面或日志中输出敏感信息

说明：敏感信息的范围应该基于应用场景以及产品威胁分析的结果来确定。典型的敏感信息包括口令、银行账号、个人信息、通讯记录、密钥、License 等，如果 shell 脚本中往界面或日志中输出敏感信息，都可能会有助于攻击者尝试发起进一步的攻击，因此 shell 脚本中禁止往界面或日志中输出敏感信息，避免被攻击者所获取并利用。

敏感信息举例（包括但不限于）：

- 口令和密钥：口令包括口令明文和口令密文
- 环境变量：命令 set、env、declare、 export 以及 typeset 会把操作系统的环境变量显示出来，其中可能存在关键信息

- 产品相关信息：产品 License，日志中如 “password”、“passwd”等关键字信息

对敏感信息建议采取以下措施：

- 不输出到界面或日志中
- 若特殊原因必须要输出到界面或日志，则用“*”代替，且不表达显示敏感信息长度，比如固定使用 1 或多个“*”。

错误示例 1：该例中，功能实现时为方便定位问题，在日志和界面上输出了详细信息，其中包括数据库密码，存在被利用的的风险。

```
#!/bin/bash
LOG_FILE=/tmp/tmp.log

function func()
{
    local db_user=$1
    local db_instance_name=$2
    local db_pvalue=$3

    echo "db user <${db_user}>, db instance name <${db_instance_name}>,
db password <${db_pvalue}>" | tee -a ${LOG_FILE}

    # ……
}
```

推荐做法：不将敏感信息输出到界面及日志中，避免被利用。

```
#!/bin/bash
LOG_FILE=/tmp/tmp.log

function func()
{
    local db_user=$1
    local db_instance_name=$2
    local db_pvalue=$3

    echo "db user <${db_user}>, db instance name <${db_instance_name}>"
| tee -a ${LOG_FILE}

    # ……
}
```

错误示例 2：如下示例中，如果环境变量 plvalue 为关键信息

```
# env
……
plvalue=Changeme_123
……
```

脚本内容如下，会将环境变量输出至日志及界面，存在信息泄露风险

```
#!/bin/bash
LOG_FILE=/tmp/tmp.log
function func()
{
    env | tee -a ${LOG_FILE}
}
```

推荐做法：禁止在 shell 脚本中把 set、env、declare、 export 及 typeset 命令的回显打印到日志及界面上。

例外情况：可以将命令执行结果赋值给一个变量，但需保证该变量值不输出到日志和界面。

如下所示。

```
#!/bin/bash
function func()
{
    V_env=$(env)
}
```

规则 1.5 禁止在 PATH 环境变量中使用相对路径

说明：环境变量攻击是用来操控脚本行为的最常见方式，通过修改环境变量攻击者可以改变一个依赖环境变量值的脚本的执行过程或执行结果。其中最常被修改的环境变量是 PATH ，该变量值的路径集决定用户在执行不包含完整路径的单独命令时，查找该命令的路径及路径查找先后顺序。

假设 PATH 变量内容为/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin，正常情况下命令会顺序在/usr/local/sbin、/usr/local/bin、/usr/sbin、/usr/bin、/sbin、/bin 中查找并执行。如果 PATH 环境变量内容设置不当，存在如“.”的相对路径，攻击者可以在当前目录下创建与系统命令同名的恶意程序或脚本，当用户执行该命令时，就会触发执行该恶意程序，导致恶意攻击。**防护措施：**在脚本中修改 PATH 环境变量时必须使用绝对路径。

错误示例：创建名为 ls 的脚本，内容如下：

```
#!/bin/bash
echo "I'm not ls command"
```

建立脚本 target.sh，内容如下：

```
#!/bin/bash
PATH=.:${PATH}
ls -l
```

执行脚本 target.sh，结果为非预期的“ls -l”结果，如下所示：

```
I'm not ls command
```

由于该脚本以当前用户权限执行，脚本中调用的命令也会拥有该权限，如果是 root 用户，安全隐患更大。

推荐做法： 修改 PATH 环境变量时，使用绝对路径可以防止如上问题，如下所示。

```
#!/bin/bash
PATH=/bin:${PATH}
ls -l
```

规则 1.6 正确使用经过验证的安全的标准加密算法

说明：禁止使用私有算法或者弱加密算法（比如 DES，SHA1 等）。应该使用经过验证的、安全的、公开的加密算法。

规则 1.7 禁止用户降级直接调用脚本

说明：禁止使用不合法用户调用脚本，否则可能导致权限提升，比如：不能使用 root 用户调用普通用户的脚本，避免存在权限提升问题。

错误示例：脚本 check_pad.sh 的执行用户应为“ossuser”，但如下调用脚本中却使用 root 用户直接调用，可能存在权限提升的隐患。

```
#!/bin/bash
#......
bash /tmp/check_pad.sh
#......
```

推荐做法：切换使用正确的用户“ossuser”执行 check_pad.sh，可以避免此类问题。

```
#!/bin/bash
#......
su - ossuser -c "bash /tmp/check_pad.sh"
#......
```

2. 脚本执行

规则 2.1 禁止使用调试模式执行 shell 脚本

说明：调试模式执行 shell 脚本，虽然能比较方便的查看脚本执行过程以及数据信息，但也会暴露相关的数据，包括可能的敏感信息，存在被利用的风险。
调试模式有两种：

1、 命令行调用shell脚本时使用选项“-x”，如“bash -x test.sh”

2、 在shell脚本中使用命令“set -x”打开调试模式

防护措施：禁止使用调试模式执行涉及敏感信息的脚本
错误示例 1：如下脚本中，使用选项“-x”调用 check_mirror.sh

```
#!/bin/bash
#......
bash -x check_mirror.sh
#......
```

执行结果如下，结果信息中可能会显露敏感信息

```
+ INSTALL_DIR=/opt/installtmp    #如果该变量包含敏感信息，就会明文显示出来。
+ LOGDIR=/opt/installtmp/installstep
+ '[' '!' -d /opt/installtmp/installstep ']'
+ LOG=/opt/installtmp/installstep/check_mirror.log
+ '[' -f /opt/installtmp/installstep/check_mirror.log ']'
+ rm -rf /opt/installtmp/installstep/check_mirror.log
+ /usr/bin/touch /opt/installtmp/installstep/check_mirror.log
+ TMPDIR=/opt/installtmp/tmp
+ '[' '!' -d /opt/installtmp/tmp ']'
```

推荐做法：不使用“-x”选项调用脚本，

```
#!/bin/bash
#......
bash check_mirror.sh
#......
```

错误示例 2：在 shell 脚本中使用命令“set -x”，运行结果和打开选项“-x”一样

```
#!/bin/bash
set -x
function func()
{
    #......
}
set +x
```

推荐做法：脚本中禁止使用“set -x” 选项

```
#!/bin/bash
function func()
```

```
{
    #......
}
```

规则 2.2 用户执行非查询类 shell 脚本必须记录操作日志

说明：日志是问题溯源的重要依据，产品中非查询类的对外接口脚本，且在产品配套资料中有明确说明，需在这类脚本中实现操作日志记录功能，且需设置日志访问权限，普通用户无权进行编辑和删除。
另外对于产品安全模块的查询类操作，也需要记录操作日志。

建议 2.3 检查脚本执行用户的合法性

说明：在跨信任边界的入口脚本中，应在入口处校验执行脚本的用户的合法性，如果不做校验，只要用户有权限就可以执行，在执行脚本过程中调用的命令等就会拥有该用户的权限，可能存在越权隐患或生成的文件属主不正确而导致功能异常。
错误示例：如下脚本中没有检查执行该脚本的用户的合法性，存在执行用户非原业务设计的用户执行该脚本，存在越权隐患。

```
#!/bin/bash
CURRENT_USER=`id -un`
echo "the current user is ${CURRENT_USER}"

#业务逻辑
```

推荐做法：检查执行用户是否是 ossuser，不是就报错退出

```
#!/bin/bash

# 判断当前用户是否为 ossuser
if [ "`id -un`" != "ossuser" ]
then
#出错处理
    exit 1
fi

#业务逻辑
```

3. 目录&文件操作

规则 3.1 创建文件时必须设置合理的访问权限

说明：文件访问权限依赖于文件系统，而文件系统一般都会提供控制访问权限功能。如果程序在创建文件时没有对文件的访问权限做足够的限制，攻击者可能在程序修改此文件的访问权限之前对其进行读取或者修改，所以一定要在创建文件时就为其指定访问权限，以防止未授权的文件访问，可以参考规则 3.2 中的例子。
文件权限设置参考如下：

- 产品自建目录及文件，所有目录和可执行文件权限控制不能大于0750。
- 敏感文件权限控制不能大于0600。
- 敏感可执行文件权限控制不能大于0700。
- 日志文件权限不能大于640。

其中敏感文件应该基于系统场景以及威胁分析的结果来确定，和规则 1.3 中的敏感信息原则一致。

规则 3.2 确保临时文件目录名称随机且只有创建者能访问

说明：shell 脚本中通常会将临时文件存放在/tmp 目录下，而/tmp 默认对所有人可读写，如果将一些敏感信息保存在/tmp 下的临时文件，该临时文件容易被监控、查看及攻击，导致临时文件攻击。

防护措施：针对该类临时文件攻击，可以采取如下策略

1、 创建临时文件时使用 umask 设置初始权限，取消同组和其他用户的读取/写入/执行权限。

2、 使用命令 mktemp 创建临时文件目录，且只有创建者可以访问。

错误示例：下例中，脚本内容如下

```
function func()
{
    rm -f /tmp/tempfile.$$
    echo "test" > /tmp/tempfile.$$
}
```

执行如上脚本，会在/tmp 目录下生成文件：tempfile.1251，虽然攻击者不知道进程 id 是多少，但只要创建足够多的 tempfile 文件，如 tempfile.1000，tempfile.1001 …，也可以猜测出该临时文件的文件名。

推荐做法 1：用 mktemp 创建只有脚本作者能访问的临时目录

```
function func()
```

```
{
    : ${TMPDIR:=/tmp}
    local temp_dir=""
    local save_mask=$(umask)
    umask 077
    temp_dir=$(mktemp -d "${TMPDIR}/XXXXXXXXXXXXXXXXXX")
    if [ $? -ne 0 ]
    then
        #出错处理
        umask "${save_mask}"
        return 1
    fi
    umask "${save_mask}"

    #业务逻辑
}
```

推荐做法 2：如果不支持命令 mktemp，则用 mkdir 创建只有脚本作者能访问的临时目录

```
function func()
{
    local temp_dir="/tmp/tmp_${RANDOM}_$$"
    mkdir -m 700 "${temp_dir}"
    if [ $? -ne 0 ]
    then
        #出错处理"
        return 1
    fi

    #业务逻辑
}
```

规则 3.3 及时删除使用完的临时文件

说明：脚本执行过程中会产生的临时文件，在使用后必须删除，避免遗留的临时文件中存储了产品关键信息，可能被利用。

推荐做法：使用 mktemp 创建只有脚本作者能访问的临时目录，并用 trap 命令在脚本退出时删除整个临时目录，如下脚本所示

```
#!/bin/bash
```

```
: ${TMPDIR:=/tmp}
trap '[ -n "${temp_dir}" ] && rm -rf "${temp_dir}"' EXIT

save_mask=$(umask)
umask 077
temp_dir=$(mktemp -d "${TMPDIR}/XXXXXXXXXXXXXXXXXX")
if [ $? -ne 0 ]
then
    #出错处理
    umask "${save_mask}"
    exit 1
fi
umask "${save_mask}"

#业务逻辑
```

建议 3.1 谨慎使用 rm 命令批量删除文件或目录

说明：使用 rm -f 删除文件或者 rm -rf 删除目录时，易出现文件或目录误删除，导致系统发生严重事故。在使用 rm 删除文件时，务必严格判断删除条件，比如删除文件时优先使用绝对路径，如果路径包含变量时充分判断异常情况。

错误示例 1：如下脚本所示，如果参数“file_path”内容包含空格，可能会导致误删其他文件；如果${file_path}实际上不是文件，而是一个链接或者其他类型时，也会存在误删。

```
#!/bin/bash
function fn_del_file()
{
    local file_path=$1
    rm -f ${file_path}
    echo "Ok: delete file successfully"
    return 0
}
```

推荐做法：对参数增加校验，确保安全删除文件

```
#!/bin/bash
function fn_del_file()
{
    local file_path=$1

    # 判断变量是否为空
    if [ -n "${file_path}" ]
    then
        # 判断是否是文件
        if [ -f "${file_path}" ]
        then
            rm -f "${file_path}"
```

```
            echo "Ok: delete file successfully"
            return 0
        else
            echo "Warn: the file is not exist"
            return 0
        fi
    else
        echo "Error: parameter invalid"
        return 1
    fi
}
```

错误示例2：如下脚本为删除指定目录下的所有文件。如果变量${dir_path}为空或者为系统根盘时，会直接删除系统根目录下的文件，导致系统事故；如果变量${ dir_path } 包含空格，导致用户未能删除想删除的目录，也可能会误删除其他目录。

```
#!/bin/bash
function fn_del_dir()
{
    local dir_path=$1
    rm -rf ${dir_path}/*
    echo "Ok: delete directory successfully"
    return 0
}
```

推荐做法：判断${dir_path}是否是目录，再判断${dir_path}是否为系统根目录，避免误删除系统根目录；对${dir_path}使用双引号解决带空格的问题，避免${dir_path}存在空格而导致误删除其他目录。

```
#!/bin/bash
function fn_del_dir()
{
    local dir_path=$1

    # 判断变量不为空且不是系统根盘
    if [ -n "${dir_path}" ] && [[ ! "${dir_path}" =~ "^/+$" ]]
    then
        # 判断是否是目录
        if [ -d "${dir_path}" ]
        then
            rm -rf "${dir_path}"/*
            echo "Ok: delete directory successfully"
            return 0
        else
            echo "Warn: the directory is not exist"
            return 0
        fi
```

```
    else
        echo "Error: parameter invalid"
        return 1
    fi
}
```

4. 其他

规则 4.1 禁止用注释方式使无用的功能代码失效

说明： shell 作为解释性语言，如果发布版本的脚本中存在没有调用的功能代码，即使用注释使其失效，在实际运行环境上可以方便的去掉注释使代码生效，存在安全风险，根据网络安全红线要求，必须删除 shell 脚本中不使用的模块、函数、变量等无效代码。

错误示例：该例中，开发阶段为方便调试，编写了免鉴权登录功能，正式发布版本中，只注释掉功能对应代码，该代码可直接删除注释重新启用，导致非法获得登录系统的权限。

```
# 免鉴权登录函数
#function fn login without auth()
#{
    #……
#}

function fn login()
{
    local username=$1
    local pvalue=$2

    #if [ "${login type}" = "NoAuth" ]
    #then
        #fn login without auth
    #fi

    if [ -n "${username}" -a -n "${pvalue}" ]
    then
        fn login with auth "${username}" "${pvalue}"
        if [ $? -ne 0 ]
        then
            #出错处理
            return 1
        else
            echo "Ok: login successfully"
            return 0
        fi
    else
        #出错处理
        return 1
    fi
}
```

推荐做法：直接删除无效代码，消除可能的安全风险。

```
function fn login()
{
    local username=$1
    local pvalue=$2

    # 参数校验
    if [ -n "${username}" -a -n "${pvalue}" ]
    then
        fn login with auth "${username}" "${pvalue}"
        if [ $? -ne 0 ]
        then
            #出错处理
            return 1
        else
            echo "Ok: login successfully"
            return 0
        fi
    else
        #出错处理
```

```
        return 1
    fi
}
```

规则 4.2 用重定向标准输入方式处理程序调用 shell 脚本传入敏感信息的场景

说明：当程序调用 shell 脚本时，如果参数中包含密码等敏感信息，在 shell 运行过程中普通用户使用 ps -ef 可以显示该脚本的进程信息，存在敏感信息泄露的风险。

错误示例：如下脚本 test.sh，涉及使用数据库密码作为参数调用脚本。

```
#!/bin/bash
DB_USER=$1
DB_INSTANCE_NAME=$2
DB_PVALUE=$3

# connect DB code here

sleep 30  # 测试该场景才使用该代码
unset DB_PVALUE
exit 0
```

执行脚本

```
#./test.sh sa DBSVR1 Changeme123
```

使用其他用户，运行命令“ps -ef | grep test.sh”，可以显示参数列表中的密码

```
# ps -ef | grep test.sh
root     11089 13238  0 18:48 pts/0    00:00:00 /bin/bash / test.sh sa
DBSVR1 Changeme123
root     11339  9399  0 18:48 pts/1    00:00:00 grep test.sh
```

推荐做法：存在敏感信息做参数的场景，使用重定向标准输入方式调用脚本

```
#!/bin/bash
DB_USER=$1
DB_INSTANCE_NAME=$2
read -s DB_PVALUE

# connect DB code here

unset DB_PVALUE
exit 0
```

执行脚本

```
# ./test.sh sa DBSVR1 <<EOF
> Changeme_123
> EOF
```

查看进程信息，结果显示查看不到密码

```
# ps -ef | grep test.sh
root     20563 13238  0 18:51 pts/0    00:00:00 /bin/bash /tmp/test.sh sa
DBSVR1
```

```
root     20815  9399  0 18:52 pts/1    00:00:00 grep test.sh
```

建议 4.1 检查函数入参有效性及合法性

说明：shell 脚本中函数入参一般是固定的类型，如文件路径、操作类型等，其取值范围一般可预计，所以函数中需根据业务逻辑对外部传入参数内容进行校验，确保该参数的有效性和合法性。

防护措施：必须校验公共函数入参的合法性，脚本中的内部函数入参不强制要求。

错误示例 1：如下代码，函数根据入参删除/opt/pub/software/tmp 目录下的指定子目录，如果传入参数为空，会误删除/opt/pub/software/tmp 整个目录。

```
function fn_del_dir()
{
    local dir_name=$1
    local dir_path="/opt/pub/software/tmp/${dir_name}"
    rm -rf "${dir_path}"
    echo "Ok: delete directory successfully"
    return 0
}
```

推荐做法：校验入参是否为空以及由入参拼接起来的路径的存在性。

```
function fn_del_dir()
{
    local dir_name=$1
    local dir_path="/opt/pub/software/tmp/${dir_name}"
    if [ -n "${dir_name}" ]
    then
        if [ -d "${dir_path}" ]
        then
            rm -rf "${dir_path}"
            echo "Ok: delete directory successfully"
            return 0
        else
            echo "Warn: the directory is not exist"
            return 0
        fi
    else
        echo "Error: parameter invalid"
        return 1
    fi
}
```

附录 A

下表中总结了常用数据库中与 SQL 注入攻击相关的特殊字符：

数据库	特殊字符	描述	转义序列
Oracle	%	百分比：任何包括 0 或更多字符的字符串	/% escape '/'
	_	下划线：任意单个字符	/_ escape '/'
	/	斜线：转义字符	// escape '/'
	'	单引号	''
MySQL	'	单引号	\'
	"	双引号	\"
	\	反斜杠	\\
	%	百分比：任何包括 0 或更多字符的字符串	\%
	_	下划线：任意单个字符	_
DB2	'	单引号	''
	;	冒号	.
SQL Server	'	单引号	''
	[	左中括号：转义字符	[[]
	_	下划线：任意单个字符	[_]
	%	百分比：任何包括 0 或更多字符的字符串	[%]
	^	插入符号：不包括以下字符	[^]

附录 B

下表中总结了 shell 脚本中常用的与命令注入相关的特殊字符：

类型	举例	常见注入模式和结果
管道	|	| shell_command -执行命令并返回命令输出信息
内联	;	; shell_command -执行命令并返回命令输出信息
	&	& shell_command -执行命令并返回命令输出信息

逻辑运算符	$	$(shell_command) -执行命令
	&&	&& shell_command -执行命令并返回命令输出信息
	||	|| shell_command -执行命令并返回命令输出信息
重定向运算符	>	> target_file -使用前面命令的输出信息写入目标文件
	>>	>> target_file -将前面命令的输出信息附加到目标文件
	<	< target_file -将目标文件的内容发送到前面的命令