14. Hints and Tips

참고: 더 많은 힌트와 팁은 **https://www.shellscript.sh/tips** 에 자주 게시됩니다. 더 흥미롭고 최신의 힌트가 있는지 확인해 보세요. **CGI** 스크립팅과 같이 다소 학술적인 내용도 있습니다.

유닉스에는 텍스트를 조작하는 유틸리티가 가득하며, 이 튜토리얼의 이 섹션에서는 그 중 몇 가지 강력한 유틸리티에 대해 설명합니다. 여기서 중요한 점은 유닉스에서는 거의 모든 것이 텍스트라는 점입니다. 여러분이 생각할 수 있는 거의 모든 것이 텍스트 파일이나 명령줄 인터페이스(CLI)로 제어됩니다. 셸 스크립트를 사용하여 자동화할 수 없는 유일한 것은 GUI 전용 유틸리티나 기능입니다. 그리고 유닉스에서는 그 수가 그리 많지 않습니다!

*닉스를 사용하면 "모든 것이 파일이다"라는 말을 들어보셨을 것입니다. - 사실입니다.

여기에는 몇 가지 하위 섹션이 있습니다... 다음은 일반적인 조언, 힌트 및 팁입니다.

CGI Scripting

CGI 프로그래밍을 할 때 주의해야 할 몇 가지 추가 변수와 그 과정에서 얻은 몇 가지 팁이 있습니다. 셸은 CGI 프로그래밍에 적합한 언어가 아닌 것처럼 보일 수 있지만, 작성 속도가 빠르고 디버깅이 간단합니다. 따라서 CGI 스크립트를 위한 이상적인 프로토타이핑 언어이며, 단순하거나 거의 사용하지 않는 CGI 스크립트를 영구적으로 사용하기에도 좋습니다. fortune.cgi를 호출하는 cookie.cgi

Exit Codes

종료 코드는 0에서 255 사이의 숫자로, 모든 Unix 명령이 상위 프로세스로 제어권을 반환할 때 반환되는 숫자입니다. 다른 숫자를 사용할 수도 있지만, 이러한 숫자는 256 모듈로 처리되므로 종료 -10은 종료 246에 해당하고 종료 257은 종료 1에 해당합니다.

셸 스크립트 내에서 이러한 변수를 사용하여 실행된 명령의 성공 또는 실패에 따라 실행 흐름을 변경할 수 있습니다. 이에 대해서는 10장, "변수 - 2부"에서 간략하게 소개했습니다. 여기서는 종료 코드의 해석에 대해 좀 더 자세히 살펴보겠습니다.

성공 ***(Success)***은 일반적으로 종료 0으로 표시되며, 실패 ***(Failure)***는 일반적으로 0이 아닌 종료 코드로 표시됩니다. 이 값은 실패의 다양한 이유를 나타낼 수 있습니다. 예를 들어 GNU grep은 성공하면 0을, 일치하는 항목이 없으면 1을, 기타 오류(구문 오류, 존재하지 않는 입력 파일 등)가 있으면 2를 반환합니다.

오류 상태를 확인하는 세 가지 방법을 살펴보고 각 방법의 장단점에 대해 논의해 보겠습니다.

첫째, 간단한 접근 방식입니다:

```
#!/bin/sh
# First attempt at checking return codes
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
if [ "$?" -ne "0" ]; then
  echo "Sorry, cannot find user ${1} in /etc/passwd"
  exit 1
fi
NAME=`grep "^${1}:" /etc/passwd|cut -d":" -f5`
HOMEDIR=`grep "^${1}:" /etc/passwd|cut -d":" -f6`

echo "USERNAME: $USERNAME"
echo "NAME: $NAME"
echo "HOMEDIR: $HOMEDIR"
```

이 스크립트는 /etc/passwd에 유효한 사용자 아이디를 입력하면 정상적으로 작동합니다. 그러나 잘못된 코드를 입력하면 처음에 예상했던 대로 작동하지 않고 계속 실행되어 표시만 됩니다:

```
USERNAME:
NAME:
HOMEDIR:
```

왜 그럴까요? 앞서 언급했듯이 $? 변수는 마지막으로 실행된 명령의 반환 코드로 설정됩니다. 이 경우, 그것은 cut입니다. 제가 테스트하고 문서를 읽으면서 알 수 있는 한, cut은 보고하는 것처럼 느껴질 정도로 아무런 문제가 없었습니다. 빈 문자열이 입력되면 빈 문자열인 입력의 첫 번째 필드를 반환하는 작업을 수행했습니다. 이제 어떻게 해야 할까요? 여기에 오류가 있으면 grep은 잘라내기가 아니라 오류를 보고합니다. 따라서 cut이 아닌 grep의 반환 코드를 테스트해야 합니다.

```
#!/bin/sh
# Second attempt at checking return codes
grep "^${1}:" /etc/passwd > /dev/null 2>&1
if [ "$?" -ne "0" ]; then
  echo "Sorry, cannot find user ${1} in /etc/passwd"
  exit 1
fi
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
NAME=`grep "^${1}:" /etc/passwd|cut -d":" -f5`
HOMEDIR=`grep "^${1}:" /etc/passwd|cut -d":" -f6`
```

```
echo "USERNAME: $USERNAME"
echo "NAME: $NAME"
echo "HOMEDIR: $HOMEDIR"
```

이렇게 하면 코드가 약간 길어지기는 하지만 문제가 해결됩니다. 이것이 교과서에 나와 있는 기본적인 방법이지만 셸 스크립트에서 오류를 검사하는 방법에 대해 알아야 할 전부는 아닙니다. 이 방법은 특정 명령 시퀀스에 가장 적합하지 않을 수도 있고 유지 관리가 불가능할 수도 있습니다. 아래에서는 두 가지 대체 접근 방식을 살펴보겠습니다.

두 번째 접근 방식으로, 코드를 4줄짜리 테스트로 가득 채우는 대신 테스트를 별도의 함수에 넣음으로써 이 문제를 어느 정도 해결할 수 있습니다:

```
#!/bin/sh
# A Tidier approach

check_errs()
{
  # Function. Parameter 1 is the return code
  # Para. 2 is text to display on failure.
  if [ "${1}" -ne "0" ]; then
    echo "ERROR # ${1} : ${2}"
    # as a bonus, make our script exit with the right error code. exit ${1}
  fi
}

### main script starts here ###

grep "^${1}:" /etc/passwd > /dev/null 2>&1
check_errs $? "User ${1} not found in /etc/passwd"
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
check_errs $? "Cut returned an error"
echo "USERNAME: $USERNAME"
check_errs $? "echo returned an error - very strange!"
```

이를 통해 3개의 개별 테스트를 작성할 필요 없이 사용자 정의 오류 메시지를 사용하여 오류를 3번 테스트할 수 있습니다. 테스트 루틴을 한 번만 작성하면 원하는 만큼 호출할 수 있으므로 프로그래머가 거의 비용을 들이지 않고도 더욱 지능적인 스크립트를 만들 수 있습니다. Perl 프로그래머라면 이 기능이 Perl의 die 명령과 유사하다는 것을 알 수 있습니다.

세 번째 접근 방식으로는 더 간단하고 조잡한 방법을 살펴보겠습니다. 저는 리눅스 커널을 구축할 때 이 방법을 사용하는 편인데, 잘 진행되면 그냥 넘어가면 되지만 문제가 발생하면 운영자가 지능적인 작업(즉, 스크립트로는 할 수 없는 작업!)을 해야 하는 간단한 자동화 방식입니다:

```
#!/bin/sh
cd /usr/src/linux && \
     make dep && make bzImage && make modules && \
     make modules_install && \
     cp arch/i386/boot/bzImage /boot/my-new-kernel && \ cp System.map /boot && \
     echo "Your new kernel awaits, m'lord."
```

이 스크립트는 리눅스 커널 빌드와 관련된 다양한 작업(시간이 꽤 걸릴 수 있음)을 실행하고 && 연산자를 사용하여 성공 여부를 확인합니다. 이 작업을 수행하려면 if가 필요합니다:

```
#!/bin/sh
cd /usr/src/linux
if [ "$?" -eq "0" ]; then
  make dep
    if [ "$?" -eq "0" ]; then
      make bzImage
      if [ "$?" -eq "0" ]; then
        make modules
        if [ "$?" -eq "0" ]; then
          make modules_install
          if [ "$?" -eq "0" ]; then
            cp arch/i386/boot/bzImage /boot/my-new-kernel
            if [ "$?" -eq "0" ]; then
              cp System.map /boot/
              if [ "$?" -eq "0" ]; then
                echo "Your new kernel awaits, m'lord."
              fi
            fi
          fi
        fi
      fi
    fi
  fi
fi
```

... 개인적으로 따라가기가 꽤 어렵다고 생각합니다.

연산자 && 및 ||는 셸에서 AND 및 OR 테스트에 해당하는 연산자입니다. 위와 같이 함께 사용할 수도 있고, 또는:

```
#!/bin/sh
cp /foo /bar && echo Success || echo Failed
```

이 코드는 다음과 같이 echo합니다 .

```
Success
```

또는

```
Failed
```

이것은 cp 명령이 성공하느냐 실패하느냐에 따라 다릅니다 . 다음 구조를 자세히 보세요 :

```
command && command-to-execute-on-success \
  || command-to-execute-on-failure
```

각 부분에는 하나의 명령만 포함될 수 있습니다 . 이 방법은 간단한 성공 /실패 시나리오에는 편리하지만 , 에코 명령 자체의 상태를 확인하려는 경우 어떤 &&와 ||가 어떤 명령에 적용되는지 금방 헷갈리기 쉽습니다 . 또한 유지 관리도 매우 어렵습니다 . 따라서 이 구조는 간단한 명령 시퀀싱에만 사용하는 것이 좋습니다 .

이전 버전에서는 cp 명령의 성공 또는 실패 여부에 따라 서브셸을 사용하여 여러 명령을 실행할 수 있다고 제안한 적이 있습니다 :

```
cp /foo /bar && \
  ( echo Success ; echo Success part II; ) || \
  ( echo Failed ; echo Failed part II )
```

그러나 실제로 Marcel은 이것이 제대로 작동하지 않는다는 것을 발견했습니다 . 서브셸의 구문은 다음과 같습니다 :

```
( command1 ; command2; command3 )
```

서브셸의 반환 코드는 최종 명령 (이 예에서는 command3)의 반환 코드입니다 . 이 반환 코드는 전체 명령에 영향을 미칩니다 . 따라서 이 스크립트의 출력은 다음과 같습니다 :

```
cp /foo /bar && \
  ( echo Success ; echo Success part II; /bin/false ) ||\
  ( echo Failed ; echo Failed part II )
```

cp가 성공했기 때문에 성공 부분을 실행하고 , /bin/false가 실패를 반환하기 때문에 실패 부분도 실행한다는 것입니다 :

```
Success
Success part II
Failed
Failed part II
```

따라서 다른 조건의 상태에 따라 여러 명령을 실행해야 하는 경우 표준 if, then, else 구문을 사용하는 것이 더 좋고 훨씬 명확합니다.

Simple Expect Replacement

다음은 expect를 간단하게 대체할 수 있는 방법입니다. 많은 분들이 이 작업을 수행하는 방법에 대해 문의해 주셨고, 제가 Sun Microsystems의 Explorer 유틸리티에 사용되는 힌트 및 팁에서 보여드린 예제에서 영감을 받아 기대의 매우 간단한 버전을 소개해 드립니다.

expect.txt의 구문은 매우 간단합니다:

```
S command E[delay] expected_text
```

따라서 명령은 "S"(Send의 경우)로 시작하여 표시되고, 예상 결과는 "E"로 표시됩니다. 일부 명령은 완료하는 데 시간이 걸릴 수 있으므로 결과를 예상하기 전에 지연 시간을 지정할 수 있습니다: "E10 $"는 10초 동안 기다린 후 달러 프롬프트를 표시합니다. 예상 텍스트를 찾지 못하면 스크립트는 1초를 기다린 후 다시 시도하고, 2초, 3초를 기다린 후 예상 텍스트가 발견되거나 MAX_WAITS에 정의된 최대값에 도달할 때까지 기다립니다. 지연은 선택 사항이므로 "E $"는 즉시 프롬프트가 표시될 것으로 예상됩니다.

MAX_WAITS=5인 경우 최대 지연 시간은 5초가 아니라 1+2+3+4+5=15초가 됩니다.

```
#!/bin/sh
# expect.sh | telnet > file1
host=127.0.0.1
port=23
file=file1
MAX_WAITS=5

echo open ${host} ${port}

while read l
do
c=`echo ${l}|cut -c1`
 if [ "${c}" = "E" ]; then
   expected=`echo ${l}|cut -d" " -f2-`
```

```
   delay=`echo ${l}|cut -d" " -f1|cut -c2-`
   if [ -z "${delay}" ]; then
      sleep ${delay}
   fi
   res=1
   i=0
   while [ "${res}" -ne "0" ]
   do
     tail -1 "${file}" 2>/dev/null | grep "${expected}" > /dev/null
     res=$?
     sleep $i
     i=`expr $i + 1`
     if [ "${i}" -gt "${MAX_WAITS}" ]; then
        echo "ERROR : Waiting for ${expected}" >> ${file}
        exit 1
      fi
    done
  else
    echo ${l} |cut -d" " -f2-
  fi
done < expect.txt
```

이렇게 실행됩니다 :

```
$ expect.sh | telnet > file1
```

이렇게 하면 세션의 기록이 포함된 file1이라는 파일이 생성됩니다 . 이 경우 로그인 프로세스 , /tmp의 ls, cal의 출력이 됩니다 . 예를 들어 :

```
telnet> Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

declan login: steve
Password:
Last login: Thu May 30 23:52:50 +0100 2002 on pts/3 from localhost.
No mail.
steve:~$ ls /tmp
API.txt                  cgihtml-1.69.tar.gz            orbit-root
cal
```

```
a.txt                   cmd.txt                      orbit-steve
apache_1.3.23.tar.gz    defaults.cgi                 parser.c
b.txt                   diary.c                      patchdiag.xref
background.jpg          drops.jpg                    sh-thd-1013541438
blocks.jpg              fortune-mod-9708.tar.gz      stone-dark.jpg
blue3.jpg               grey2.jpg                    water.jpg
c.txt                   jpsock.131.1249
steve:~$ cal
      May 2002
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
steve:~$ exit
logout
```

Trap

Trap은 간단하지만 매우 유용한 유틸리티입니다. 현재 디렉터리의 모든 파일에서 FOO를 BAR로 바꾸는 이 간단한 스크립트와 같이 스크립트가 임시 파일을 생성하는 경우, 스크립트가 종료될 때 /tmp는 깨끗해집니다. 하지만 스크립트가 도중에 중단되면 /tmp에 파일이 남아있을 수 있습니다:

```
#!/bin/sh

trap cleanup 1 2 3 6

cleanup()
{
  echo "Caught Signal ... cleaning up."
  rm -rf /tmp/temp_*.$$
  echo "Done cleanup ... quitting."
  exit 1
}

### main script
for i in *
```

```
do
  sed s/FOO/BAR/g $i > /tmp/temp_${i}.$$ && mv /tmp/temp_${i}.$$ $i
done
```

trap 문은 스크립트에서 signal 1, 2, 3 또는 6에 대해 cleanup()을 실행하도록 지시합니다. 가장 일반적인 것 (CTRL-C)은 signal 2(SIGINT)입니다. 이것은 매우 흥미로운 용도로도 사용될 수 있습니다:

```
#!/bin/sh

trap 'increment' 2

increment()
{
  echo "Caught SIGINT ..."
  X=`expr ${X} + 500`
  if [ "${X}" -gt "2000" ]
  then
    echo "Okay, I'll quit ..."
    exit 1
  fi
}

### main script
X=0
while :
do
  echo "X=$X"
  X=`expr ${X} + 1`
  sleep 1
done
```

위의 스크립트는 꽤 재미있습니다. CTRL-C를 잡아서 종료하지 않고 실행 방식만 변경합니다. 이것이 어떤 긍정적인 효과와 부정적인 효과를 가져올 수 있는지는 독자의 몫으로 남겨두겠습니다.) 이 특정 예제에서는 4번의 인터럽트 (또는 2000초) 후에 종료하도록 허용합니다. 처리할 기회를 얻지 못한 채 kill -9 <PID>에 의해 모든 것이 종료된다는 점에 유의하세요.

다음은 몇 가지 일반적인 인터럽트를 정리한 표입니다:

Number	SIG	뜻
0	0	셸에서 종료할 때
1	SIGHUP	깔끔한 정리

2	SIGINT	인터럽트
3	SIGQUIT	종료 (Quit)
6	SIGABRT	취소 (Abort)
9	SIGKILL	바로 Kill함 (덫에 걸리지 않음)
14	SIGALRM	알람 시계
15	SIGTERM	종료 (Terminate)

스크립트가 자체적으로 신호를 무시하는 환경 (예 : **nohup** 제어 상태)에서 시작된 경우 스크립트는 해당 신호도 무시합니다 .

echo: -n vs \c

이미 눈치채셨겠지만 , echo 문을 사용하면 명령의 끝에 줄 바꿈이 추가됩니다 . 이에 대한 수정 사항이 있습니다 ... 아니 , 더 정확하게는 두 가지 수정 사항이 있습니다 .

일부 Unix 시스템에서는 **echo -n message**를 사용하여 echo에 줄 바꿈을 추가하지 말라고 지시하고 , 다른 시스템에서는 **echo message \c**를 사용하여 동일한 작업을 수행합니다 :

```
echo -n "Enter your name:"
read name
echo "Hello, $name"
```

이는 일부 시스템에서 작동하며 다음과 같이 표시됩니다 :

```
Enter your name: Steve
Hello, Steve
```

그러나 다른 시스템에서는 다음과 같이 코드를 작성해야 합니다 :

```
echo "Enter your name: \c"
read name
echo "Hello, $name"
```

그러면 해당 시스템에서 동일한 결과를 얻을 수 있습니다 .

골치 아픈 일이죠 . 다음은 두 가지 모두에서 작동하는 해결 방법입니다 :

```
if [ "`echo -n`" = "-n" ]; then
    n=""
```

```
    c="\c"
else
    n="-n"
    c=""
fi

echo $n Enter your name: $c
read name
echo "Hello, $name"
```

echo -n이 제대로 해석되지 않으면 -n이라는 텍스트만 echo 하게되며 , 이 경우 $n은 빈 문자열로 설정되고 $c는 \c로 설정됩니다 . 그렇지 않으면 그 반대의 결과가 나오므로 $n은 -n으로 설정되고 $c는 빈 문자열로 설정됩니다 .

위에서 이미 간단하지만 효과적인 cut 명령의 사용법을 보여드렸습니다 . 여기서는 좀 더 일반적으로 사용되는 외부 프로그램 몇 가지를 예로 들어 설명하겠습니다 .

grep은 셸 스크립트 프로그래머에게 매우 유용한 유틸리티입니다 . grep의 예는 다음과 같습니다 :

```
#!/bin/sh
steves=`grep -i steve /etc/passwd | cut -d: -f1`
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: $steves"
```

일치하는 항목이 하나만 있으면 이 스크립트는 정상적으로 보입니다 . 그러나 /etc/passwd에 "steve"라는 단어가 포함된 두 줄이 있으면 대화형 셸이 표시됩니다 :

```
$> grep -i steve /etc/passwd
steve:x:5062:509:Steve Parker:/home/steve:/bin/bash
fred:x:5068:512:Fred Stevens:/home/fred:/bin/bash
$> grep -i steve /etc/passwd |cut -d: -f1
steve
fred
```

하지만 스크립트가 표시됩니다 :

```
Entries are: steve fred
```

결과를 변수에 넣음으로써 NEWLINE을 공백으로 변경했습니다 . sh 매뉴얼 페이지에서는 $IFS의 첫 번째 문자가 이 용도로 사용된다고 알려줍니다 . IFS는 기본적으로 <space><tab><cr>입니다 . 아마도 우리는 NEWLINE을 유지하고 싶었을 것입니다 : 공백을 NEWLINEs.... 으로 만들면 더 보기 좋을 것입니다 . 이것은 tr 을 위한 작업입니다 :

```
#!/bin/sh
steves=`grep -i steve /etc/passwd | cut -d: -f1`
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: "
echo "$steves" | tr ' ' '\012'
```

tr은 공백을 8진수 문자 012(NEWLINE)로 변환한다는 점에 유의하세요. tr의 또 다른 일반적인 용도는 범위를 사용하는 것입니다. 예를 들어 텍스트를 대문자 또는 소문자로 변환할 수 있습니다:

```
#!/bin/sh
steves=`grep -i steve /etc/passwd | cut -d: -f1`
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: "
echo "$steves" | tr ' ' '\012' | tr '[a-z]' '[A-Z]'
```

여기서는 [a-z]를 [A-Z]로 번역했습니다. a-z 범위에는 A-Z와 정확히 같은 수의 값이 있다는 점에 유의하세요. 그러면 ASCII 범위 a-z에 속하는 모든 문자를 A-Z로 번역할 수 있습니다. 즉, 소문자를 대문자로 변환하는 것입니다. tr은 실제로 이보다 더 똑똑합니다. tr [:lower:] [:upper:]이 더 잘 작동하고 가독성도 더 높을 수 있습니다. 또한 이식성이 떨어지므로 모든 tr이 이 작업을 수행할 수 있는 것은 아닙니다.

Cheating

속임수를 쓸 수 없는 사람들

치팅은 잘못된 것이 아닙니다! 셸이 잘하지 못하는 것도 있습니다. 두 가지 유용한 도구는 sed와 awk입니다. 이 두 가지 유틸리티는 그 자체로 미니 프로그래밍 언어로 사용할 수 있는 매우 강력한 유틸리티이지만, 셸 스크립트에서는 매우 간단하고 특정한 이유로 사용되는 경우가 많습니다.

이는 시스템에서 대용량 실행 파일(sed의 경우 52k, awk의 경우 110k)을 로드해야 한다는 것을 의미하며, 이는 끔찍한 일이지만, 훌륭한 작업자가 도구를 탓하지 않는 이유는 훌륭한 작업자는 처음부터 올바른 도구를 사용하기 때문입니다. 그래서 이 두 가지 도구를 아주 간단하게 소개해드리겠습니다.

Cheating with awk

텍스트 파일에 있는 문자, 줄, 단어의 개수를 계산하는 wc를 예로 들어 보겠습니다. 출력은 다음과 같습니다:

```
$ wc hex2env.c
          102     189     2306     hex2env.c
```

줄 수를 변수로 가져오고 싶다면 다음을 사용하면 됩니다:

```
NO_LINES=`wc -l file`
```

를 입력하면 전체 줄을 읽을 수 있습니다. 출력에 공백이 추가되어 있기 때문에 숫자 102를 문자열에 안정적으로 넣을 수 없습니다. 대신, awk가 C의 scanf와 유사하게 작동한다는 사실을 이용해 원치 않는 공백을 제거합니다. 이를 $1 $2 $3 등의 변수에 넣습니다. 그래서 이 구조를 사용합니다:

```
NO_LINES=`wc -l file | awk '{ print $1 }'`
```

이제 NO_LINES 변수는 102입니다.

Cheating with sed

또 다른 편리한 유틸리티는 stream editor인 sed입니다. Perl은 정규식 처리에 매우 능숙하지만 셸은 그렇지 않습니다. 따라서 예를 들어 sed를 호출하여 s/from/to/g 구문을 빠르게 사용할 수 있습니다:

```
sed s/eth0/eth1/g file1 >  file2
```

는 파일 1에 있는 eth0의 모든 인스턴스를 파일 2에 있는 eth1로 변경합니다. 단일 문자만 변경하는 경우, 크기가 작고 로드 속도가 빠른 tr을 사용할 수 있습니다. tr이 할 수 없는 또 다른 일은 파일에서 문자를 제거하는 것입니다:

```
echo ${SOMETHING} | sed s/"bad word"//g
```

이렇게 하면 ${SOMETHING} 변수에서 "'bad word'"라는 문구가 제거됩니다. 이렇게 말하고 싶을 수도 있습니다.
"하지만 grep은 그렇게 할 수 있습니다!"라고 말하고 싶을 수도 있습니다. - grep은 전체 줄만 처리합니다. 파일을 생각해 봅시다:

```
This line is okay.
This line contains a bad word. Treat with care.
This line is fine, too.
```

grep은 두 번째 줄 전체를 제거하여 두 줄짜리 파일만 남기고, sed는 파일을 읽기로 변경합니다:

```
This line is okay.
This line contains a . Treat with care.
This line is fine, too.
```

Telnet hint

이것은 제가 Sun의 Explorer 유틸리티에서 배운 유용한 기술입니다. 텔넷은 더 이상 서버에서 사용되지 않지만 터미널 집중 장치 등과 같은 일부 네트워크 장치에서는 여전히 사용됩니다. 이와 같은 스크립트를 직접 만들거나 명령줄에서 실행할 수 있습니다:

```
$ ./telnet1.sh | telnet
```

몇몇 사람들이 이 문제에 대해 물어본 적이 있는데, 저는 꽤 복잡하고 부피가 큰 코드 모음인 기대 코드를 추천하는 경향이 있습니다. 이 코드는 시스템 간에 이식성이 높아야 합니다 (egrep가 있는 한). 시스템에서 작동하지 않는 경우 -q 스위치와 함께 GNU grep을 사용하거나 전용 grep을 사용하여 /dev/null로 직접 실행해 보세요. 그래도 기대 설치보다는 훨씬 쉽습니다.

```
#!/bin/sh
host=127.0.0.1
port=23
login=steve
passwd=hellothere
cmd="ls /tmp"

echo open ${host} ${port}
sleep 1
echo ${login}
sleep 1
echo ${passwd}
sleep 1
echo ${cmd}
sleep 1
echo exit
```

하지만 Sun은 몇 가지 영리한 오류 검사 코드를 추가했습니다 (현재 셸 또는 셸 스크립트에서 변수를 설정하고 내보내어 읽을 수 있는 파일에 비밀번호가 저장되는 것을 방지할 수 있습니다):

```
$ ./telnet2.sh | telnet > file1
```

```
#!/bin/sh
# telnet2.sh | telnet > FILE1
host=127.0.0.1
port=23
login=steve
passwd=hellothere
cmd="ls /tmp"
timeout=3
file=file1
prompt="$"

echo open ${host} ${port}
```

```
sleep 1
tout=${timeout}
while [ "${tout}" -ge 0 ]
do
    if tail -1 "${file}" 2>/dev/null | \
      egrep -e "login:" > /dev/null
    then
        echo "${login}"
        sleep 1
        tout=-5
        continue
    else
        sleep 1
        tout=`expr ${tout} - 1`
    fi
done

if [ "${tout}" -ne "-5" ]; then
  exit 1
fi

tout=${timeout}
while [ "${tout}" -ge 0 ]
do
    if tail -1 "${file}" 2>/dev/null | \
       egrep -e "Password:" > /dev/null
    then
        echo "${passwd}"
        sleep 1
        tout=-5
        continue
    else
      if tail -1 "${file}" 2>/dev/null | \
     egrep -e "${prompt}" > /dev/null
      then
        tout=-5
      else
        sleep 1
        tout=`expr ${tout} - 1`
      fi
```

```
    fi
done

if [ "${tout}" -ne "-5" ]; then
  exit 1
fi

> ${file}

echo ${cmd}
sleep 1
echo exit
```

이 버전에서는 출력이 file1에 저장되며, 이 파일은 실제로 스크립트에서 진행 상황을 확인하는 데 사용됩니다. "> ${file}"을 추가하여 파일로 수신되는 출력이 로그인 프로세스가 아닌 명령의 출력만 되도록 했습니다.

Revision #4
Created 17 January 2024 01:41:52 by Enigma
Updated 17 January 2024 03:47:21 by Enigma