Bourne Again Shell (Bash)

Bash ek scripting language hai, jiska use hum Unix-based OS ke saath communicate karne aur system ko commands dene ke liye karte hain.

May 2019 se, Windows **Windows Subsystem for Linux (WSL)** provide karta hai, jo humein Windows environment mein **Bash** use karne ki permission deta hai.

Is language par achhi command hona zaroori hai, taaki hum iske saath efficiently kaam kar sakein.

Scripting aur Programming languages ke beech main difference yeh hai ki scripting language ke code ko execute karne ke liye humein compile karne ki zaroorat nahi hoti, jabki programming languages mein code ko execute karne se pehle compile karna padta hai.

Penetration testers ke roop mein, humein kisi bhi Operating System ke saath kaam karna aana chahiye, chahe woh Windows ho ya Unix-based.

Hamari efficiency mainly systems ki knowledge par depend karti hai, khaaskar Privilege Escalation ke field mein.

Unix-based systems mein terminal use karna, data filter karna aur in processes ko automate karna seekhna bahut zaroori hai.

Khaaskar bade Unix-based enterprise networks mein, humein bahut saare data ke saath kaam karna padta hai.

Humein data ko properly sort aur filter karna hota hai, taaki hum possible security gaps aur important information ko jitni jaldi ho sake identify kar sakein.

Kai commands ko combine karna aur unke individual results ke saath kaam karna seekhna bhi zaroori hai.

Yahin par Scripting ka use hota hai, jo hamari speed aur efficiency ko badhati hai.

Programming language ki tarah, scripting language ka structure bhi lagbhag same hota hai, jise in parts mein divide kiya ja sakta hai:

- **Input** & **Output** – Input aur Output
- **Arguments**, **Variables** & **Arrays** – Arguments, Variables aur Arrays
- **Conditional execution** – Condition ke according commands execute karna
- **Arithmetic** – Mathematical calculations
- **Loops** – Ek hi kaam ko baar-baar repeat karna
- **Comparison operators** – Do values ki comparison karna

- **Functions** – Commands ke group ko ek naam dena, taaki unhe dobara use kar sakein.

Aksar kuch processes ko automate karna common hota hai, taaki humein unhe baar-baar manually repeat na karna pade, ya phir bahut saari information ko process aur filter karna aasaan ho sake.

Generally, ek script khud se process create nahi karti, balki use ek Interpreter execute karta hai, jo script ko run karta hai. Is case mein interpreter Bash hai.

Kisi script ko execute karne ke liye, humein interpreter specify karna hota hai aur use batana hota hai ki kaunsi script process karni hai.

Is tarah se command likhi jaati hai:

Script Execution - Examples

```
Toxicx01@htb[/htb]$ bash script.sh <optional arguments>
```

```
Toxicx01@htb[/htb]$ ./script.sh <optional arguments>
```

```
Toxicx01@htb[/htb]$ sh script.sh <optional arguments>
```

Aaiye, aisi hi ek script ko dekhte hain aur samajhte hain ki specific results paane ke liye ise kaise create kiya ja sakta hai.

Agar hum is script ko execute karte hain aur ek domain specify karte hain, toh humein pata chalta hai ki yeh script kaunsi information provide karti hai.

CIDR.sh

```
Toxicx01@htb[/htb]$ ./CIDR.sh inlanefreight.com

Discovered IP address(es):
165.22.119.202

Additional options available:
    1) Identify the corresponding network range of target domain.
    2) Ping discovered hosts.
    3) All checks.
    *) Exit.

Select your option: 3

NetRange for 165.22.119.202:
NetRange:       165.22.0.0 - 165.22.255.255
CIDR:           165.22.0.0/16

Pinging host(s):
165.22.119.202 is up.

1 out of 1 hosts are up.
```

Ab aaiye, us script ko detail mein dekhte hain aur har line ko sabse achhe tareeke se samajhte hain.

Agle sections mein, hum is script ke sabhi parts ko ek-ek karke dekhenge aur unka analysis karenge.

CIDR.sh

```
#!/bin/bash

# Check for given arguments
if [ $# -eq 0 ]
then
   echo -e "You need to specify the target domain.\n"
   echo -e "Usage:"
   echo -e "\t$0 <domain>"
   exit 1
else
   domain=$1
fi

# Identify Network range for the specified IP address(es)
function network_range {
```

```
    for ip in $ipaddr
    do
        netrange=$(whois $ip | grep "NetRange\|CIDR" | tee -a CIDR.txt)
        cidr=$(whois $ip | grep "CIDR" | awk '{print $2}')
        cidr_ips=$(prips $cidr)
        echo -e "\nNetRange for $ip:"
        echo -e "$netrange"
    done
}

# Ping discovered IP address(es)
function ping_host {
    hosts_up=0
    hosts_total=0

    echo -e "\nPinging host(s):"
    for host in $cidr_ips
    do
        stat=1
        while [ $stat -eq 1 ]
        do
            ping -c 2 $host > /dev/null 2>&1
            if [ $? -eq 0 ]
            then
                echo "$host is up."
                ((stat--))
                ((hosts_up++))
                ((hosts_total++))
            else
                echo "$host is down."
                ((stat--))
                ((hosts_total++))
            fi
        done
    done

    echo -e "\n$hosts_up out of $hosts_total hosts are up."
}

# Identify IP address of the specified domain
hosts=$(host $domain | grep "has address" | cut -d" " -f4 | tee discovered_hosts.txt)

echo -e "Discovered IP address:\n$hosts\n"
ipaddr=$(host $domain | grep "has address" | cut -d" " -f4 | tr "\n" " ")
```

```
# Available options
echo -e "Additional options available:"
echo -e "\t1) Identify the corresponding network range of target domain."
echo -e "\t2) Ping discovered hosts."
echo -e "\t3) All checks."
echo -e "\t*) Exit.\n"

read -p "Select your option: " opt

case $opt in
   "1") network_range ;;
   "2") ping_host ;;
   "3") network_range && ping_host ;;
   "*") exit 0 ;;
esac
```

Jaise ki hum dekh sakte hain, yahan humne script ke alag-alag parts par comments likhe hain, jinmein is script ko divide kiya ja sakta hai.

1. **Diye gaye arguments ko check karna.**
2. **Diye gaye IP address(es) ke liye network range identify karna.**
3. **Discover kiye gaye IP address(es) ko Ping karna.**
4. **Diye gaye domain ke IP address(es) identify karna.**
5. **Available options ko dikhana.**

---

1. Diye gaye Arguments ko Check karna

Script ke pehle part mein, humare paas ek **if-else** statement hai, jo check karta hai ki kya humne target company ko represent karne wala domain specify kiya hai.

---

2. Diye gaye IP Address(es) ke liye Network Range Identify karna

Yahan humne ek function create kiya hai, jo har IP address ke liye **whois** query perform karta hai.

Yeh function reserved network range wali line ko display karta hai aur use **CIDR.txt** file mein store karta hai.

---

3. Discover kiye gaye IP Address(es) ko Ping karna

Is additional function ka use yeh check karne ke liye kiya jaata hai ki mile hue hosts apne respective IP addresses ke through reachable hain ya nahi.

For-Loop ki help se, hum network range ke har IP address ko ping karte hain aur ping ke results ko count karte hain.

---

4. Diye gaye Domain ke IP Address(es) ko Identify karna

Script ke pehle step mein, hum us domain ka IPv4 address identify karte hain, jo humein return kiya gaya hai.

---

5. Available Options

Iske baad, hum decide karte hain ki infrastructure ke baare mein aur zyada information pata karne ke liye humein kaun-se functions use karne hain.

Conditional Execution

Conditional execution humein apni script ke flow ko control karne ki permission deta hai, jismein alag-alag conditions ke according code execute hota hai.

Yeh scripting ka ek bahut important component hai. Iske bina hum sirf ek command ke baad doosri command execute kar sakte hain.

Jab hum alag-alag conditions define karte hain, toh hum specify karte hain ki kisi particular value ke liye kaun-se functions ya code sections execute hone chahiye.

Agar koi specific condition match hoti hai, toh sirf usi condition ka code execute hota hai aur baaki conditions ke code skip kar diye jaate hain.

Jaise hi us code section ka execution complete hota hai, uske baad conditional execution ke bahar likhi hui commands execute hone lagti hain.

Aaiye, script ke pehle part ko dobara dekhte hain aur uska analysis karte hain.

Script.sh

```
#!/bin/bash

# Check for given argument
if [ $# -eq 0 ]
then
    echo -e "You need to specify the target domain.\n"
    echo -e "Usage:"
    echo -e "\t$0 <domain>"
    exit 1
else
    domain=$1
fi

<SNIP>
```

Summary mein, code ke is section mein hum neeche diye gaye components ka use karte hain:

- **#!/bin/bash** – Shebang.
- **if-else-fi** – Conditional execution (conditions ke according code execute karna).
- **echo** – Specific output ko print karna.
- **$# / $0 / $1** – Special variables.
- **domain** – Variables.

Conditional execution ki conditions ko variables (**$#, $0, $1, domain)**, values (**0**) aur strings ka use karke define kiya ja sakta hai, jaisa ki hum agle examples mein dekhenge.

In values ko **comparison operators** (**-eq**) ki help se compare kiya jaata hai, jiske baare mein hum agle section mein padhenge.

Shebang

Shebang line hamesha har script ke sabse upar hoti hai aur hamesha **#!** se start hoti hai.

Is line mein specified interpreter ka path (**/bin/bash**) hota hai, jiske through script execute ki jaati hai.

Hum Shebang ka use karke doosre interpreters bhi define kar sakte hain, jaise Python, Perl aur anya interpreters.

```
#!/usr/bin/env python
```

```
#!/usr/bin/env perl
```

If-Else-Fi

Programming ke sabse basic tasks mein se ek hai alag-alag conditions ko check karna aur unke according kaam karna.

Programming aur scripting languages mein conditions check karne ke usually do alag forms hote hain:

1. **if-else condition**
2. **case statements**

Pseudo-Code

```bash
if [ the number of given arguments equals 0 ]
then
    Print: "You need to specify the target domain."
    Print: "<empty line>"
    Print: "Usage:"
    Print: "   <name of the script> <domain>"
    Exit the script with an error
else
    The "domain" variable serves as the alias for the given argument
finish the if-condition
```

By default, **If-Else** condition mein sirf ek **If** ho sakta hai, jaisa ki agle example mein dikhaya gaya hai.

If-Only.sh

```
#!/bin/bash

value=$1

if [ $value -gt "10" ]
then
        echo "Given argument is greater than 10."
fi
```

If-Only.sh - Execution

```
Toxicx01@htb[/htb]$ bash if-only.sh 5
```

```
Toxicx01@htb[/htb]$ bash if-only.sh 12

Given argument is greater than 10.
```

Jab hum Elif ya Else add karte hain, toh hum alag-alag values ya conditions ko handle karne ke liye doosre options add karte hain.

Agar koi particular value pehli condition se match nahi karti, toh use baaki conditions handle kar leti hain.

If-Elif-Else.sh

```
#!/bin/bash

value=$1

if [ $value -gt "10" ]
then
    echo "Given argument is greater than 10."
elif [ $value -lt "10" ]
then
    echo "Given argument is less than 10."
else
    echo "Given argument is not a number."
fi
```

If-Elif-Else.sh - Execution

```
Toxicx01@htb[/htb]$ bash if-elif-else.sh 5

Given argument is less than 10.
```

```
Toxicx01@htb[/htb]$ bash if-elif-else.sh 12

Given argument is greater than 10.
```

```
Toxicx01@htb[/htb]$ bash if-elif-else.sh HTB

if-elif-else.sh: line 5: [: HTB: integer expression expected
if-elif-else.sh: line 8: [: HTB: integer expression expected
Given argument is not a number.
```

Hum apni script ko aur bada kar sakte hain aur usmein kai conditions specify kar sakte hain. Yeh kuch is tarah dikh sakta hai:

Several Conditions - Script.sh

```
#!/bin/bash

# Check for given argument
if [ $# -eq 0 ]
then
    echo -e "You need to specify the target domain.\n"
    echo -e "Usage:"
    echo -e "\t$0 <domain>"
    exit 1
elif [ $# -eq 1 ]
then
    domain=$1
else
    echo -e "Too many arguments given."
    exit 1
fi

<SNIP>
```

Yahan hum ek aur condition (**elif [<condition>];then**) define karte hain, jo ek line print karti hai (**echo -e "..."**) aur humein batati hai ki humne ek se zyada arguments diye hain. Iske baad program ko error ke saath band kar diya jata hai (**exit 1**)।

Exercise Script

```
#!/bin/bash
# Count number of characters in a variable:
#     echo $variable | wc -m

# Variable to encode
var="nef892na9s1p9asn2aJs71nIsm"

for counter in {1..40}
do
        var=$(echo $var | base64)
done
```

Q. Create an "If-Else" condition in the "For"-Loop of the "Exercise Script" that prints you the number of characters of the 35th generated value of the variable "var". Submit the number as the answer.

Ans

Arguments, Variables, and Arrays

Arguments

Bash scripts ka fayda ye hai ki hum bina unhe variables mein assign kiye ya unke liye alag se requirements set kiye, script mein maximum 9 arguments (**$0–$9**) pass kar sakte hain.

9 arguments isliye, kyunki pehla argument **$0** script ke liye reserved hota hai.

Jaisa ki hum yahan dekh sakte hain, kisi variable ko uski specified position par use karne ke liye variable ke naam se pehle dollar sign **($)** lagana zaroori hota hai.

Iske comparison mein assignment kuch is tarah dikhega:

```
Toxicx01@htb[/htb]$ ./script.sh ARG1 ARG2 ARG3 ... ARG9
       ASSIGNMENTS:       $0      $1   $2   $3 ...   $9
```

Iska matlab hai ki humne automatically corresponding arguments ko yahan pehle se define kiye gaye variables mein assign kar diya hai.

In variables ko Special Variables kaha jata hai.

Ye special variables placeholders ki tarah kaam karte hain.

Agar hum ab dobara code section ko dekhen, toh humein pata chalega ki kaun-se arguments kahan aur kaise use kiye gaye hain.

CIDR.sh

```
#!/bin/bash

# Check for given argument
if [ $# -eq 0 ]
then
    echo -e "You need to specify the target domain.\n"
    echo -e "Usage:"
    echo -e "\t$0 <domain>"
    exit 1
else
    domain=$1
fi

<SNIP>
```

Hum apni script ko execute (run) karne ke kai tarike use kar sakte hain.

Lekin script ko usmein define kiye gaye interpreter se execute karne se pehle, humein sabse pehle script ko execute karne ki permission (execution privileges) deni hogi.

CIDR.sh - Set Execution Privileges

```
Toxicx01@htb[/htb]$ chmod +x cidr.sh
```

CIDR.sh - Execution without Arguments

```
Toxicx01@htb[/htb]$ ./cidr.sh

You need to specify the target domain.

Usage:
    cidr.sh <domain>
```

CIDR.sh - Execution without Execution Permissions

```
Toxicx01@htb[/htb]$ bash cidr.sh

You need to specify the target domain.

Usage:
    cidr.sh <domain>
```

Special Variables

Special variables, Internal Field Separator (IFS) ka use karte hain, taaki ye identify kar sakein ki ek argument kahan khatam hota hai aur doosra argument kahan se shuru hota hai.

Bash scripting mein help karne ke liye kai tarah ke special variables provide karta hai.

Inmein se kuch variables ha**in:**

Special Variable	Description
$#	Ye variable script mein pass kiye gaye arguments ki total sankhya rakhta hai.
$@	Is variable ka use sabhi command-line arguments ki list ko retrieve karne ke liye kiya ja sakta hai.
$n	Har command-line argument ko uski position ke according alag-alag retrieve kar sakte hain. Example: pehla argument **$1** par milta hai.
$$	Ye currently execute ho rahe process ki Process ID (PID) hoti hai.

$?	Ye script ka exit status batata hai. Iska use pata karne ke liye hota hai ki koi command successful hui ya nahi. **0** ka matlab successful execution aur 1 ka matlab failure hota hai.

Upar diye gaye variables mein se, hamare **if-else** condition mein 3 special variables use kiye gaye hain.

Special Variable	Description
$#	Is case mein humein sirf ek variable chahiye, jismein value ko domain variable mein assign kiya jayega. Is variable ka use us target ko specify karne ke liye hota hai, jiske saath hum kaam karna chahte hain. Agar hum argument ke roop mein sirf ek FQDN provide karte hain, toh $# variable ki value 1 hogi.
$0	Is special variable mein execute ki gayi script ka naam assign hota hai, jise "**Usage:**" example mein dikhaya jata hai.
$1	Space se separate kiya gaya pehla argument is special variable mein assign hota hai.

Variables

Hum **if-else** loop ke end mein ye bhi dekhte hain ki hum first argument ki value ko **domain** naam ke variable mein assign karte hain.

Variables ko assign karte waqt dollar sign ($) ka use nahi kiya jata.

Dollar sign ka use sirf isliye hota hai taaki is variable ki corresponding value ko code ke doosre sections mein use kiya ja sake.

Variables assign karte waqt, variable ke naam aur value ke beech koi space nahi hona chahiye.

```
<SNIP>
else
    domain=$1
fi
<SNIP>
```

Dusri programming languages ke comparison mein, Bash mein variables ke types ke beech direct difference aur recognition nahi hota, jaise **"strings"**, **"integers"** aur **"boolean".**

Variables ke andar jo bhi content hota hai, Bash un sabhi ko string characters ke roop mein treat karta hai.

Bash arithmetic functions ko tab enable karta hai, jab variables mein sirf numbers assign kiye gaye hon.

Variables declare karte waqt ye dhyan rakhna important hai ki unmein **space** nahi hona chahiye.

Agar space diya gaya, toh actual variable name ko ek internal function ya command ke roop mein interpret kiya jayega.

Declaring a Variable - Error

```
Toxicx01@htb[/htb]$ variable = "this will result with an error."

command not found: variable
```

Declaring a Variable - Without an Error

```
Toxicx01@htb[/htb]$ variable="Declared without an error."
Toxicx01@htb[/htb]$ echo $variable

Declared without an error.
```

Arrays

Bash mein ek hi variable ko multiple values assign karne ka option bhi hota hai.

Ye tab useful ho sakta hai jab hum multiple domains ya IP addresses ko scan karna chahte hain.

In variables ko **arrays** kaha jata hai. Hum inka use kisi specific type ki values ke ordered sequence ko store aur process karne ke liye kar sakte hain.

Arrays har stored entry ko ek **index** se identify karte hain, jo **0** se start hota hai.

Jab humein array ke kisi component ko value assign karni hoti hai, toh hum bilkul waise hi karte hain jaise normal shell variables ke saath karte hain.

Bas humein square brackets **[ ]** ke andar field ka index specify karna hota hai.

Bash mein **arrays** ko declare karne ka tarika kuch is tarah hota hai:

Arrays.sh

```
#!/bin/bash

domains=(www.inlanefreight.com ftp.inlanefreight.com vpn.inlanefreight.com
www2.inlanefreight.com)

echo ${domains[0]}
```

Hum index ka use karke unhe individually bhi retrieve kar sakte hain. Iske liye variable ke saath corresponding index ko curly brackets {} ke andar likhte hain.

Curly brackets ka use Variable Expansion ke liye kiya jata hai.

```
Toxicx01@htb[/htb]$ ./Arrays.sh

www.inlanefreight.com
```

Ye dhyan rakhna important hai ki single quotes (' ... ') aur double quotes (" ... ") array ki individual values ko spaces ke basis par alag hone se rokte hain.

Iska matlab hai ki single ya double quotes ke andar maujood sabhi spaces ko ek hi value ka part maana jata hai aur us poori value ko array mein assign kiya jata hai.

Arrays.sh

```
#!/bin/bash

domains=("www.inlanefreight.com ftp.inlanefreight.com vpn.inlanefreight.com"
www2.inlanefreight.com)
echo ${domains[0]}
```

```
Toxicx01@htb[/htb]$ ./Arrays.sh

www.inlanefreight.com ftp.inlanefreight.com vpn.inlanefreight.com
```

Q. Submit the echo statement that would print "www2.inlanefreight.com" when running the last "Arrays.sh" script.

Ans.

Comparison Operators

Kisi bhi specific values ki ek-dusre ke saath comparison karne ke liye humein kuch elements ki zarurat hoti hai, jinhe Comparison Operators kaha jata hai.

Comparison Operators ka use ye pata lagane ke liye kiya jata hai ki define ki gayi values ek-dusre se kaise compare hoti hain.

In operators ko hum 4 categories mein divide karte hain:

1. String Operators – Text (words ya letters) ko compare karne ke liye.
2. Integer Operators – Numbers (integers) ko compare karne ke liye.
3. File Operators – Files aur directories ko check karne ke liye.
4. Boolean Operators – True aur False conditions ko compare karne ke liye.

String Operators

Jab hum strings (text/words/letters) ko compare karte hain, toh humein pata hota hai ki corresponding value mein hum kya check karna chahte hain.

Operator	Description
==	Agar dono strings equal (barabar) hain.
!=	Agar dono strings equal nahi hain.
<	ASCII alphabetical order ke according agar ek string doosri se chhoti hai.
>	ASCII alphabetical order ke according agar ek string doosri se badi hai.
-z	Agar string empty (khali) hai.
-n	Agar string empty nahi hai (usmein kuch value hai).

Yahan ye dhyan dena important hai ki hum diye gaye argument ke variable ($1) ko double-quotes ("$1") ke andar likhte hain.

Isse Bash ko pata chalta hai ki variable ke content ko ek String (text) ki tarah handle karna hai.

Agar hum double-quotes ka use nahi karenge, toh humein error mil sakta hai.

```bash
#!/bin/bash

# Check the given argument
if [ "$1" != "HackTheBox" ]
then
    echo -e "You need to give 'HackTheBox' as argument."
    exit 1

elif [ $# -gt 1 ]
then
    echo -e "Too many arguments given."
    exit 1

else
    domain=$1
    echo -e "Success!"
fi
```

String comparison operators (< / >) sirf double square brackets [[<condition>]] ke andar hi kaam karte hain.

Hum ASCII table ko Internet par ya terminal mein neeche diye gaye command ka use karke dekh sakte hain.

Hum iska ek example aage dekhenge.

```
Toxicx01@htb[/htb]$ man ascii
```

ASCII Table

Decimal	Hexadecial	Character	Description
0	00	NUL	End of a string
...	...	...	...
65	41	A	Capital A
66	42	B	Capital B
67	43	C	Capital C
68	44	D	Capital D
...	...	...	...
127	7F	DEL	Delete

ASCII ka full form **American Standard Code for Information Interchange** hai aur ye 7-bit character encoding ko represent karta hai.

Kyunki har bit ki 2 values ho sakti hain (0 ya 1), isliye ismein **128** alag-alag bit patterns hote hain.

Inhe decimal integers **0** se **127** tak ya hexadecimal values **00** se **7F** tak bhi represent kiya ja sakta hai.

Pehle 32 ASCII character codes ko Control Characters kaha jata hai, jo reserve kiye gaye hote hain.

Integer Operators

Integer numbers (poore numbers) ko compare karna hamare liye bahut useful ho sakta hai, agar humein pata ho ki hum kin values ki comparison karna chahte hain.

Iske according, hum agle steps aur commands define karte hain ki script ko corresponding value ke saath kaise handle karna chahiye.

Operator	Description
`-eq`	is equal to
`-ne`	is not equal to
`-lt`	is less than
`-le`	is less than or equal to
`-gt`	is greater than
`-ge`	is greater than or equal to

```bash
#!/bin/bash

# Check the given argument
if [ $# -lt 1 ]
then
    echo -e "Number of given arguments is less than 1"
    exit 1

elif [ $# -gt 1 ]
then
    echo -e "Number of given arguments is greater than 1"
    exit 1

else
    domain=$1
    echo -e "Number of given arguments equals 1"
fi
```

File Operators

File operators tab useful hote hain jab hum ye check karna chahte hain ki kisi file ya directory ki specific permissions hain ya nahi, ya phir woh file ya directory exist karti hai ya nahi.

Operator	Description
-e	Check karta hai ki file exist (maujood) karti hai ya nahi.
-f	Check karta hai ki given path ek file hai ya nahi.
-d	Check karta hai ki given path ek directory hai ya nahi.
-L	Check karta hai ki given path ek symbolic link hai ya nahi.

-N	Check karta hai ki file ko last read karne ke baad modify kiya gaya hai ya nahi.
-O	Check karta hai ki file ka owner current user hai ya nahi.
-G	Check karta hai ki file ki group ID current user ki group ID se match karti hai ya nahi.
-s	Check karta hai ki file ka size 0 se greater hai ya nahi.
-r	Check karta hai ki file mein read permission hai ya nahi.
-w	Check karta hai ki file mein write permission hai ya nahi.
-x	Check karta hai ki file mein execute permission hai ya nahi.

```bash
#!/bin/bash

# Check if the specified file exists
if [ -e "$1" ]
then
    echo -e "The file exists."
    exit 0

else
    echo -e "The file does not exist."
    exit 2
fi
```

Boolean and Logical Operators

Logical operators ka use karne par humein result ke roop mein Boolean value milti hai, jo **false** ya **true** hoti hai.

Bash humein double square brackets **[[<condition>]]** ka use karke strings ko compare karne ki possibility deta hai.

Ye Boolean values paane ke liye hum String Operators ka use kar sakte hain.

Comparison match hota hai ya nahi, uske according humein Boolean value **false** ya **true** milti hai.

```
#!/bin/bash

# Check the boolean value
if [[ -z $1 ]]
then
    echo -e "Boolean value: True (is null)"
    exit 1

elif [[ $# > 1 ]]
then
    echo -e "Boolean value: True (is greater than)"
    exit 1

else
    domain=$1
    echo -e "Boolean value: False (is equal to)"
fi
```

Logical Operators

Logical operators ki help se hum ek hi condition ke andar multiple conditions define kar sakte hain.

Iska matlab hai ki corresponding code execute hone se pehle, humne jitni bhi conditions define ki hain, un sabhi ka match hona zaroori hai.

Operator	Description
`!`	logical negotation NOT
`&&`	logical AND
`||`	logical OR

```bash
#!/bin/bash

# Check if the specified file exists and if we have read permissions
if [[ -e "$1" && -r "$1" ]]
then
    echo -e "We can read the file that has been specified."
    exit 0

elif [[ ! -e "$1" ]]
then
    echo -e "The specified file does not exist."
    exit 2

elif [[ -e "$1" && ! -r "$1" ]]
then
    echo -e "We don't have read permission for this file."
    exit 1

else
    echo -e "Error occured."
    exit 5
fi
```

Exercise Script

```bash
#!/bin/bash

var="8dm7KsjU28B7v621Jls"
value="ERmFRMVZ0U2paTlJYTkxDZz09Cg"

for i in {1..40}
do
        var=$(echo $var | base64)

        #<---- If condition here:
done
```

Q. Create an "If-Else" condition in the "For"-Loop that checks if the variable named "var" contains the contents of the variable named "value". Additionally, the variable "var" must contain more than 113,450 characters. If these conditions are met, the script must then print the last 20 characters of the variable "var". Submit these last 20 characters as the answer.

Ans.

Arithmetic

Bash mein humare paas 7 alag-alag **Arithmetic Operators** hote hain, jin ke saath hum kaam kar sakte hain.

Inka use alag-alag mathematical calculations karne ya kuch integers (poore numbers) ki values ko modify (badalne) karne ke liye kiya jata hai.

Arithmetic Operators

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus
`variable++`	Increase the value of the variable by 1
`variable--`	Decrease the value of the variable by 1

Hum in sabhi operators ko ek chhoti si script mein summarize (ek jagah par samjha) kar sakte hain:

Arithmetic.sh

```
#!/bin/bash

increase=1
decrease=1

echo "Addition: 10 + 10 = $((10 + 10))"
echo "Subtraction: 10 - 10 = $((10 - 10))"
echo "Multiplication: 10 * 10 = $((10 * 10))"
echo "Division: 10 / 10 = $((10 / 10))"
echo "Modulus: 10 % 4 = $((10 % 4))"

((increase++))
echo "Increase Variable: $increase"

((decrease--))
echo "Decrease Variable: $decrease"
```

Is script ka output kuch is tarah dikhta hai :-

Arithmetic.sh - Execution

```
Toxicx01@htb[/htb]$ ./Arithmetic.sh

Addition: 10 + 10 = 20
Subtraction: 10 - 10 = 0
Multiplication: 10 * 10 = 100
Division: 10 / 10 = 1
Modulus: 10 % 4 = 2
Increase Variable: 2
Decrease Variable: 0
```

Hum variable ki length bhi calculate kar sakte hain.

${#variable} function ka use karke variable ke har character ko count kiya jata hai aur humein variable mein total kitne characters hain, ye pata chalta hai.

VarLength.sh

```
#!/bin/bash

htb="HackTheBox"

echo ${#htb}
```

VarLength.sh

```
Toxicx01@htb[/htb]$ ./VarLength.sh

10
```

Agar hum apni **CIDR.sh** script ko dekhein, toh humein pata chalega ki humne **increase** aur **decrease** operators ka kai baar use kiya hai.

Isse ye ensure hota hai ki **while** loop, jiske baare mein hum baad mein padhenge, tab tak run karta rahe aur hosts ko ping karta rahe, jab tak variable **stat** ki value **1** hai.

Agar **ping** command **0** code ke saath end hoti hai (yaani ping successful hota hai), toh humein ek message milta hai ki **host is up.**

Iske saath hi **stat, hosts_up** aur **hosts_total** variables ki values bhi change ho jaati hain.

```
<SNIP>
    echo -e "\nPinging host(s):"
    for host in $cidr_ips
    do
        stat=1
        while [ $stat -eq 1 ]
        do
            ping -c 2 $host > /dev/null 2>&1
            if [ $? -eq 0 ]
            then
                echo "$host is up."
                ((stat--))
                ((hosts_up++))
                ((hosts_total++))
            else
                echo "$host is down."
                ((stat--))
                ((hosts_total++))
            fi
        done
    done
<SNIP>
```

Input Control

Humein apni bheji hui requests aur execute kiye gaye commands se results mil sakte hain. In results ko dekhkar humein manually decide karna hota hai ki aage kya karna hai.

Ek aur example hai: humne apni script mein alag-alag situations ke liye kai functions define kiye hain. Ab results ko manually check karne ke baad humein decide karna hoga ki unmein se kaunsa function execute hona chahiye.

Ye bhi possible hai ki kuch specific scans ya activities ko perform karne ki permission na ho.

Isliye, humein ye pata hona chahiye ki running script ko hamare instructions ka wait kaise karwaya jata hai.

Agar hum apni **CIDR.sh** script ko dobara dekhein, toh humein dikhega ki aage ke steps decide karne ke liye humne aisi hi ek call add ki hai.

CIDR.sh

```
# Available options
<SNIP>
echo -e "Additional options available:"
echo -e "\t1) Identify the corresponding network range of target domain."
echo -e "\t2) Ping discovered hosts."
echo -e "\t3) All checks."
echo -e "\t*) Exit.\n"

read -p "Select your option: " opt

case $opt in
    "1") network_range ;;
    "2") ping_host ;;
    "3") network_range && ping_host ;;
    "*") exit 0 ;;
esac
```

Pehli **echo** lines hamare liye available options ka ek display menu banati hain.

read command ki help se "**Select your option**:" wali line display hoti hai. Ismein additional option **-p** ensure karta hai ki hamara input usi line mein rahe.

Hamara input **opt** variable mein store hota hai. Iske baad hum **case** statement ka use karke corresponding functions execute karte hain, jiske baare mein hum baad mein padhenge.

Hum jo number enter karte hain, uske according **case** statement decide karta hai ki kaunse functions execute honge.

Output Control

Hum Linux Fundamentals module mein output redirection ke baare mein pehle hi seekh chuke hain.

Lekin redirection ke saath problem ye hai ki humein respective command ka output screen par nahi dikhta. Uska output directly appropriate file mein redirect ho jata hai.

Agar future mein hamari scripts zyada complicated ho jaati hain, toh unhe execute hone mein sirf kuch seconds se zyada time lag sakta hai.

Script ke results ka wait karte hue bina kuch kiye baithe rehne se bachne ke liye hum tee utility ka use kar sakte hain.

Ye ensure karta hai ki humein results turant screen par dikh jaayein aur saath hi woh corresponding files mein bhi store ho jaayein.

Hamari **CIDR.sh** script mein humne is utility ka do baar alag-alag tarikon se use kiya hai.

CIDR.sh

```
<SNIP>

# Identify Network range for the specified IP address(es)
function network_range {
    for ip in $ipaddr
    do
        netrange=$(whois $ip | grep "NetRange\|CIDR" | tee -a CIDR.txt)
        cidr=$(whois $ip | grep "CIDR" | awk '{print $2}')
        cidr_ips=$(prips $cidr)
        echo -e "\nNetRange for $ip:"
        echo -e "$netrange"
    done
}

<SNIP>

# Identify IP address of the specified domain
hosts=$(host $domain | grep "has address" | cut -d" " -f4 | tee
discovered_hosts.txt)

<SNIP>
```

Jab hum **tee** ka use karte hain, toh received output ko pipe (|) ke through tee command tak forward karte hain.

-a / **--append** parameter ensure karta hai ki specified file overwrite (purani information delete) nahi hoti, balki usmein naye results add ho jaate hain.

Saath hi, **tee** humein results screen par bhi dikhata hai aur unhe file mein bhi store karta hai.

```
Toxicx01@htb[/htb]$ cat discovered_hosts.txt CIDR.txt

165.22.119.202
NetRange:       165.22.0.0 - 165.22.255.255
CIDR:           165.22.0.0/16
```

Flow Control - Loops

Hamari scripts ke flow ko control karna bahut zaroori hai.

Hum pehle hi **if-else** conditions ke baare mein seekh chuke hain, jo Flow Control ka hi ek part hain.

Aakhirkar, hum chahte hain ki hamari script jaldi aur efficiently kaam kare. Iske liye hum kuch aur components ka use kar sakte hain, jo efficiency badhate hain aur bina errors ke processing karne mein help karte hain.

Har control structure ya toh ek **Branch** hota hai ya ek **Loop**.

Boolean values ke logical expressions aam taur par control structure ke execution ko control karte hain.

In control structures mein ye shamil hain:

- Branches:

- **If-Else** Conditions
- **Case** Statements

- Loops:

- **For** Loops
- **While** Loops
- **Until** Loops

For Loops

Chaliye **For** loops se shuru karte hain.

For loop har baar (iteration) mein sirf ek parameter ke liye execute hota hai. Ye parameter shell kisi list se leta hai, increment ke through calculate karta hai ya kisi doosre data source se leta hai.

For loop tab tak chalta hai, jab tak use corresponding data milta rehta hai.

Is type ke loop ko alag-alag tarikon se structure aur define kiya ja sakta hai.

Example:

For loops ka use aksar tab kiya jata hai, jab humein kisi array ki bahut saari alag-alag values ke saath kaam karna hota hai.

Iska use hum different hosts ya ports ko scan karne ke liye kar sakte hain.

Hum iska use known ports aur unki services ke liye specific commands execute karne mein bhi kar sakte hain, taaki hamara enumeration process jaldi complete ho sake.

Iska syntax kuch is tarah ho sakta hai:

Syntax - Examples

```
for variable in 1 2 3 4
do
    echo $variable
done
```

```
for ip in 10.10.10.170 10.10.10.174 10.10.10.175
do
    ping -c 1 $ip
done
```

```
for variable in file1 file2 file3
do
    echo $variable
done
```

Of course, hum in commands ko ek hi line mein bhi likh sakte hain. Aisi command kuch is tarah dikhegi:

```shellsession
Toxicx01@htb[/htb]$ for ip in 10.10.10.170 10.10.10.174;do ping -c 1 $ip;done

PING 10.10.10.170 (10.10.10.170): 56 data bytes
64 bytes from 10.10.10.170: icmp_seq=0 ttl=63 time=42.106 ms

--- 10.10.10.170 ping statistics ---
1 packets transmitted, 1 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 42.106/42.106/42.106/0.000 ms
PING 10.10.10.174 (10.10.10.174): 56 data bytes
64 bytes from 10.10.10.174: icmp_seq=0 ttl=63 time=45.700 ms

--- 10.10.10.174 ping statistics ---
1 packets transmitted, 1 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 45.700/45.700/45.700/0.000 ms
```

Chaliye, apni CIDR.sh script ko ek baar phir dekhte hain.

Humne is script mein kai For loops add kiye hain, lekin abhi hum iske chhote se code section par hi focus karte hain.

CIDR.sh

```bash
<SNIP>

# Identify Network range for the specified IP address(es)
function network_range {
    for ip in $ipaddr
    do
        netrange=$(whois $ip | grep "NetRange\|CIDR" | tee -a CIDR.txt)
        cidr=$(whois $ip | grep "CIDR" | awk '{print $2}')
        cidr_ips=$(prips $cidr)
        echo -e "\nNetRange for $ip:"
        echo -e "$netrange"
    done
}

<SNIP>
```

Pichhle example ki tarah, array **ipaddr** ke har IP address ke liye hum ek **whois** request bhejte hain.

Is request ke output ko **NetRange** aur **CIDR** ke liye filter kiya jata hai.

Isse humein pata lagane mein help milti hai ki hamara target kis address range mein located hai.

Hum is information ka use penetration test ke dauran additional hosts search karne ke liye kar sakte hain, agar client ne iski permission di ho.

Humein milne wale results screen par display kiye jaate hain aur saath hi **CIDR.txt** file mein store bhi kiye jaate hain.

While Loops

While loop conceptually bahut simple hai aur ye following principle par kaam karta hai:

- Jab tak koi condition satisfy (**true**) hoti hai, tab tak statement execute hota rehta hai.

Hum multiple loops ko combine bhi kar sakte hain aur unki execution ko different values ke saath merge kar sakte hain.

Ye dhyan rakhna important hai ki ek loop ke andar bahut saare loops ko combine karne se code samajhna mushkil ho sakta hai aur aisi errors aa sakti hain jinhe find karna aur follow karna difficult ho sakta hai.

Aisa combination hamari **CIDR.sh** script mein bhi dekhne ko mil sakta hai.

Think
To pick up a draggable item, press the space bar. While dragging, use the arrow keys to move the item. Press space again to drop the item in its new position, or press escape to cancel.

CIDR.sh

```
<SNIP>
        stat=1
        while [ $stat -eq 1 ]
        do
            ping -c 2 $host > /dev/null 2>&1
            if [ $? -eq 0 ]
            then
                echo "$host is up."
                ((stat--))
                ((hosts_up++))
                ((hosts_total++))
            else
                echo "$host is down."
                ((stat--))
                ((hosts_total++))
            fi
        done
<SNIP>
```

while loops bhi **if-else** ki tarah conditions ke saath kaam karte hain.

Ek **while** loop ko khud ko control karne ke liye kisi tarah ke counter ki zarurat hoti hai, jisse use pata chale ki uske andar diye gaye commands ko kab execute karna band karna hai.

Agar counter nahi hoga, toh loop lagataar chalta rahega, jisse endless loop (bina ruke chalne wala loop) ban jayega.

Aisa counter ek variable ho sakta hai, jise humne kisi specific value ke saath declare kiya ho, ya phir ek boolean value ho sakti hai.

while loops tab tak chalte hain jab tak boolean value **True** hoti hai.

Counter ke alawa, hum **break** command ka bhi use kar sakte hain. Jab loop **break** command tak pahunchta hai, toh yeh loop ko wahi rok deta hai, jaise ki neeche diye gaye example mein dikhaya gaya hai.

WhileBreaker.sh

```
#!/bin/bash

counter=0

while [ $counter -lt 10 ]
do
  # Increase $counter by 1
  ((counter++))
  echo "Counter: $counter"

  if [ $counter == 2 ]
  then
    continue
  elif [ $counter == 4 ]
  then
    break
  fi
done
```

WhileBreaker.sh

```
Toxicx01@htb[/htb]$ ./WhileBreaker.sh

Counter: 1
Counter: 2
Counter: 3
Counter: 4
```

Until Loops

until loop bhi hota hai, jo comparatively bahut kam use kiya jata hai.

Phir bhi, **until** loop bilkul **while** loop ki tarah kaam karta hai, lekin in dono mein ek difference hota hai:

- **until** loop ke andar ka code tab tak execute hota hai, jab tak di gayi condition False hoti hai.

Doosra tareeka yeh hai ki loop ko tab tak chalne diya jaye, jab tak desired value (jo value hum chahte hain) achieve nahi ho jaati.

Is kaam ke liye **until** loops bahut useful hote hain.

Yeh loop **while** loop ki tarah hi kaam karta hai, lekin jaisa ki pehle bataya gaya hai, yeh tab tak chalta hai jab tak boolean value **False** hoti hai.

Until.sh

```
#!/bin/bash

counter=0

until [ $counter -eq 10 ]
do
  # Increase $counter by 1
  ((counter++))
  echo "Counter: $counter"
done
```

Until.sh

```
Toxicx01@htb[/htb]$ ./Until.sh

Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
Counter: 8
Counter: 9
Counter: 10
```

Exercise Script

#!/bin/bash

 # Decrypt function

```
function decrypt {
   MzSaas7k=$(echo $hash | sed 's/988sn1/83unasa/g')
   Mzns7293sk=$(echo $MzSaas7k | sed 's/4d298d/9999/g')
   MzSaas7k=$(echo $Mzns7293sk | sed 's/3i8dqos82/873h4d/g')
   Mzns7293sk=$(echo $MzSaas7k | sed 's/4n9Ls/20X/g')
   MzSaas7k=$(echo $Mzns7293sk | sed 's/912oijs01/i7gg/g')
   Mzns7293sk=$(echo $MzSaas7k | sed 's/k32jx0aa/n391s/g')
   MzSaas7k=$(echo $Mzns7293sk | sed 's/nI72n/YzF1/g')
   Mzns7293sk=$(echo $MzSaas7k | sed 's/82ns71n/2d49/g')
   MzSaas7k=$(echo $Mzns7293sk | sed 's/JGcms1a/zIm12/g')
   Mzns7293sk=$(echo $MzSaas7k | sed 's/MS9/4Sls/g')
   MzSaas7k=$(echo $Mzns7293sk | sed 's/Ymxj00lms/Uso18/g')
   Mzns7293sk=$(echo $MzSaas7k | sed 's/sSi8Lm/Mit/g')
   MzSaas7k=$(echo $Mzns7293sk | sed 's/9su2n/43n92ka/g')
   Mzns7293sk=$(echo $MzSaas7k | sed 's/ggf3iunds/dn3i8/g')
   MzSaas7k=$(echo $Mzns7293sk | sed 's/uBz/TT0K/g')

   flag=$(echo $MzSaas7k | base64 -d | openssl enc -aes-128-cbc -a -d -salt -pass
pass:$salt)
}

# Variables
var="9M"
salt=""
hash="VTJGc2RHVmtYMTl2ZnYyNTdUeERVRnBtQWVGNmFWWVUySG1wTXNmRi9rQT0
K"

# Base64 Encoding Example:
#       user@htb$ echo "Some Text" | base64

# <- For-Loop here

# Check if $salt is empty
if [[ ! -z "$salt" ]]
then
   decrypt
   echo $flag
else
   exit 1
fi
```

Q. Create a "For" loop that encodes the variable "var" 28 times in "base64". The number of characters in the 28th hash is the value that must be assigned to the "salt" variable.
Ans.

Flow Control - Branches

Jaisa ki hum pehle hi dekh chuke hain, flow control mein branches ke andar **if-else** aur **case** statements shamil hote hain.

Hum **if-else** statements ke baare mein detail mein discuss kar chuke hain aur jaante hain ki yeh kaise kaam karte hain.

Ab hum **case** statements ko aur dhyan se samjhenge.

Case Statements

Case statements ko doosri programming languages, jaise C/C++ aur C# mein **switch-case** statements ke naam se bhi jaana jata hai.

if-else aur **switch-case** ke beech main difference yeh hai:

- **if-else** humein kisi bhi Boolean expression ko check karne ki permission deta hai.
- Jabki **switch-case** hamesha sirf variable ki exact value ko compare karta hai.

Isliye, **if-else** mein use hone wali conditions, jaise greater-than (>), **switch-case** mein allowed nahi hoti hain.

switch-case statements ka syntax kuch is tarah dikhta hai:

Syntax - Switch-Case

```
case <expression> in
    pattern_1 ) statements ;;
    pattern_2 ) statements ;;
    pattern_3 ) statements ;;
esac
```

switch-case ki definition **case** se start hoti hai. Iske baad variable ya value ko expression ke roop mein likha jata hai, jise pattern ke saath compare kiya jata hai.

Agar variable ya value expression se match ho jaati hai, toh parentheses ke baad diye gaye statements execute hote hain aur double semicolon (**;;**) ke saath end hote hain.

Hamari **CIDR.sh** script mein bhi humne aisa hi **case** statement use kiya hai.

Yahan humne apni script ke liye 4 alag-alag options define kiye hain. Ye options decide karte hain ki hamare decision ke baad script ko aage kaise proceed karna chahiye.

CIDR.sh

```
<SNIP>
# Available options
echo -e "Additional options available:"
echo -e "\t1) Identify the corresponding network range of target domain."
echo -e "\t2) Ping discovered hosts."
echo -e "\t3) All checks."
echo -e "\t*) Exit.\n"

read -p "Select your option: " opt

case $opt in
    "1") network_range ;;
    "2") ping_host ;;
    "3") network_range && ping_host ;;
    "*") exit 0 ;;
esac
<SNIP>
```

Pehle do options ke saath, yeh script alag-alag functions execute karti hai, jinhe humne pehle define kiya tha.

Teesre option ke saath, dono functions execute hote hain.

Aur agar koi doosra option select kiya jata hai, toh script terminate (band) ho jaati hai.

Functions

Jaise-jaise hamari scripts badi hoti jaati hain, waise-waise woh complicated aur confusing hoti jaati hain.

Agar hum script mein same commands ko baar-baar use karte hain, toh script ka size bhi usi hisaab se badhta jaata hai.

Aise cases mein **functions** ek solution hain, jo script ke size ko kam karne aur uski clarity ko kaafi improve karne mein help karte hain.

Hum multiple commands ko curly brackets (**{** ... **}**) ke beech ek block mein combine karte hain aur unhe **functions** ke through apne diye hue function name se call karte hain.

Ek baar function define ho jaane ke baad, hum use script ke andar dobara call aur use kar sakte hain.

Functions scripts aur programs ka ek important part hain, kyunki inka use baar-baar repeat hone wale commands ko script ya program ke different values aur phases ke liye execute karne mein hota hai.

Isliye humein baar-baar poora code repeat karne ki zarurat nahi hoti. Iske bajay hum ek single function bana sakte hain, jo specific commands ko execute karta hai.

Functions define karne se code ko padhna aur samajhna aasan ho jata hai aur code ka size bhi jitna ho sake utna chhota rehta hai.

Dhyan dene wali baat hai ki functions ko hamesha unke pehli baar call hone se pehle define karna chahiye, kyunki script bhi top-to-bottom process hoti hai.

Isliye function ki definition hamesha script ke beginning (shuruat) mein hoti hai.

Functions ko define karne ke do methods hain:

Method 1 - Functions

```
function name {
    <commands>
}
```

Method 2 - Functions

```
name() {
    <commands>
}
```

Hum apne liye function define karne ka woh method choose kar sakte hain jo humein sabse zyada aasan lagta hai.

Humari **CIDR.sh** script mein humne pehla method use kiya hai, kyunki **function** keyword ke saath ise padhna aur samajhna zyada aasan hota hai.

CIDR.sh

```bash
<SNIP>
# Identify Network range for the specified IP address(es)
function network_range {
    for ip in $ipaddr
    do
        netrange=$(whois $ip | grep "NetRange\|CIDR" | tee -a CIDR.txt)
        cidr=$(whois $ip | grep "CIDR" | awk '{print $2}')
        cidr_ips=$(prips $cidr)
        echo -e "\nNetRange for $ip:"
        echo -e "$netrange"
    done
}
<SNIP>
```

Function ko sirf uska diya gaya naam (name) likhkar call kiya jaata hai, jaisa ki humne case statement mein dekha hai.

```
<SNIP>
case $opt in
    "1") network_range ;;
    "2") ping_host ;;
    "3") network_range && ping_host ;;
    "*") exit 0 ;;
esac
```

Parameter Passing (Parameters Pass Karna)

Aise functions ko is tarah design karna chahiye ki unhe values ke ek fixed structure ya kam se kam ek fixed format ke saath use kiya ja sake.

Jaise ki humne apni **CIDR.sh** script mein pehle dekha tha, humne **network_range** function ke liye IP address ka format use kiya tha.

Parameters optional hote hain, isliye hum function ko bina parameters ke bhi call kar sakte hain.

Basic principle mein, function ko pass kiye gaye parameters bhi waise hi kaam karte hain jaise hum shell script ko parameters pass karte hain.

Ye hain **$1** se **$9** (**${n}**), ya **$variable**, jaisa ki hum pehle dekh chuke hain.

Har function ke apne alag parameters hote hain. Isliye ye doosre functions ya shell script ke parameters ke saath aapas mein conflict nahi karte.

Bash scripts aur doosri programming languages ke beech ek important difference ye hai ki define kiye gaye sabhi variables hamesha **globally** process hote hain, jab tak unhe **[local](https://www.tldp.org/LDP/abs/html/localvar.html)** declare na kiya gaya ho.

Iska matlab hai ki agar humne pehli baar kisi variable ko function ke andar define kiya hai, toh hum use apni main script mein bhi call kar sakte hain (function ke bahar).

Functions ko parameters pass karne ka tarika bhi wahi hota hai, jis tarah hum apni script ko arguments pass karte hain. Ye kuch is tarah dikhta hai:

PrintPars.sh

```
#!/bin/bash

function print_pars {
    echo $1 $2 $3
}

one="First parameter"
two="Second parameter"
three="Third parameter"

print_pars "$one" "$two" "$three"
```

```
Toxicx01@htb[/htb]$ ./PrintPars.sh

First parameter Second parameter Third parameter
```

Return Values (Return Values)

Jab hum koi naya process start karte hain, toh har **child process** (jaise ki execute ki gayi script ke andar ek **function**) apne khatam hone par ek **return code**, **parent process** (**bash shell**, jiske through humne script execute ki thi) ko return karta hai.

Ye return code parent process ko batata hai ki execution ka status kya tha.

Is information ka use ye pata lagane ke liye kiya jaata hai ki process successfully run hua ya usmein koi specific error aayi.

Is information ke basis par **parent process**s decide kar sakta hai ki program ko aage kaise execute karna hai.

Return Code	Description
`1`	General errors
`2`	Misuse of shell builtins
`126`	Command invoked cannot execute
`127`	Command not found
`128`	Invalid argument to exit
`128+n`	Fatal error signal "`n`"
`130`	Script terminated by Control-C
`255\*`	Exit status out of range

Kisi function se value wapas paane ke liye hum kai methods ka use kar sakte hain, jaise **return**, **echo**, ya kisi **variable** ka.

Agle example mein hum dekhenge ki **return code** ko read karne ke liye **$?** ka use kaise karte hain, function ko arguments kaise pass karte hain aur function ke result ko kisi variable mein kaise assign karte hain.

Return.sh

```
#!/bin/bash

function given_args {

	if [ $# -lt 1 ]
	then
		echo -e "Number of arguments: $#"
		return 1
	else
		echo -e "Number of arguments: $#"
		return 0
	fi
}

# No arguments given
given_args
echo -e "Function status code: $?\n"

# One argument given
given_args "argument"
echo -e "Function status code: $?\n"

# Pass the results of the funtion into a variable
content=$(given_args "argument")

echo -e "Content of the variable: \n\t$content"
```

Return.sh - Execution

```
Toxicx01@htb[/htb]$ ./Return.sh

Number of arguments: 0
Function status code: 1

Number of arguments: 1
Function status code: 0

Content of the variable:
    Number of arguments: 1
```

Debugging

Bash humein apne code mein errors ko find karne, track karne aur fix karne ka ek achha mauka deta hai.

Debugging shabd ke kai alag-alag meanings ho sakte hain. Lekin Bash debugging ka matlab hai apne code se errors (bugs) ko remove karna.

Debugging kai alag-alag tarikon se ki ja sakti hai.

Example ke liye, hum apne code ka use debugging ke liye kar sakte hain, taaki typing mistakes (typos) ko check kar sakein. Ya phir code analysis ke liye use kar sakte hain, taaki errors ko track kar sakein aur pata laga sakein ki specific errors kyun aa rahi hain.

Is process ka use programs mein vulnerabilities find karne ke liye bhi kiya jaata hai.

Example ke liye, hum alag-alag types ke inputs dekar errors create karne ki koshish kar sakte hain aur assembler ke through CPU mein un errors ko kaise handle kiya ja raha hai, ise track kar sakte hain.

Isse humein errors ko handle karne ke process ko manipulate karne ka tareeka mil sakta hai, jisse hum apna khud ka code insert kar sakein aur system ko us code ko execute karne ke liye force kar sakein.

Is topic ko doosre modules mein detail mein cover aur discuss kiya jaayega.

Bash humein **-x** (**xtrace**) aur **-v** options ka use karke apne code ko debug karne ki permission deta hai.

Ab hum apni **CIDR.sh** script ke saath ek example dekhte hain.

CIDR.sh - Debugging

```
Toxicx01@htb[/htb]$ bash -x CIDR.sh

+ '[' 0 -eq 0 ']'
+ echo -e 'You need to specify the target domain.\n'
You need to specify the target domain.

+ echo -e Usage:
Usage:
+ echo -e '\tCIDR.sh <domain>'
    CIDR.sh <domain>
+ exit 1
```

Yahan Bash humein clearly dikhata hai ki kaunsa function ya command execute hua aur kis value ke saath execute hua.

Iski pehchaan line ki shuruaat mein diye gaye plus sign **(+)** se hoti hai.

Agar hum kisi particular function ka poora code dekhna chahte hain, toh hum -v option set kar sakte hain, jo output ko aur zyada detail mein display karta hai.

CIDR.sh - Verbose Debugging

```
Toxicx01@htb[/htb]$ bash -x -v CIDR.sh

#!/bin/bash

# Check for given argument
if [ $# -eq 0 ]
then
    echo -e "You need to specify the target domain.\n"
    echo -e "Usage:"
    echo -e "\t$0 <domain>"
    exit 1
else
    domain=$1
fi
+ '[' 0 -eq 0 ']'
+ echo -e 'You need to specify the target domain.\n'
You need to specify the target domain.

+ echo -e Usage:
Usage:
+ echo -e '\tCIDR.sh <domain>'
    CIDR.sh <domain>
+ exit 1
```

Normal debugging ke comparison mein, yahan hum code ka poora section dekh sakte hain, jo ab tak process ho chuka hai. Uske baad hum un individual steps ko bhi dekh sakte hain, jo execute kiye gaye hain.