Урок 5

**Скрипты.**

Итак, мы с вами все глубже и глубже погружаемся в среду Linux и уже представляем для чего и как использовать некоторые команды.

Мы сгенерировали ключи доступа на различные ресурсы и серверы, стали понимать как нам необходимо поступать, есть вдруг нужно будет зайти на сервак и переместить там файл.

Погнали в мир скриптов!

Для начала я напомню, что такое скрипты:

**Скрипт (англ. script - сценарий) - это небольшая программа, которая содержит последовательность действий, созданных для автоматического выполнения задачи.**

Скрипты пишутся не только на разных языках программирования, но и в терминале bash в Linux.

Делается это для упрощения работы и удобства.

Давайте попробуем разобраться на практике со скриптами.

<https://bellard.org/jslinux/index.html>

Пишем:

nano script.sh

нам сейчас неважно, где мы находимся, нам нужно понять сам принцип.

Почему мы не создаем файл при помощи команды touch? Да потому что он создастся пустым, а нам нужно сделать так, чтобы в нем была информация.

Так же напомню, что файл, который мы создадим при помощи редактора nano создастся только тогда, когда мы будем выходить из редактора с сохранением. А выходить с сохранением нужно ctrl+x

Зашли и открыли. И что нам теперь нужно? А нужно написать скрипт.

#!/bin/bash

Давайте разбираться по порядку.

#!

Эти два символа называют по-разному. Решетка\восклицательный знак, диез\восклицательный знак и прочее.

Но в нашем случае это ШЕБАНГ (тот случай, когда можно написать по-русски).

Что это? Говоря простым языком - эта штука показывает, чем же открывать наш файл.

Например во всем известном Windows ассоциации файлов с программами с которыми они работают осуществляется при помощи расширения.

В Linux чуть по-другому. Файл является исполняемым, если у него есть права на исполнение. То есть если ему присвоили права на исполнение.

А чем открывать этот файл мы передаем первой строкой в этом самом файле. И инструкция о том, чтобы система понимала, чем именно нужно открывать файл начинается с ШЕБАНГА.

Что мы видим в первой строчке? То, что система должна понять, что скрипт (или сценарий) будет запускаться при помощи bash, который находится в папке /bin.

**Bash нужен для приема команд пользователя и их отправки операционной системе для последующей обработки.**

Говоря простым языком - мы пишем команду, а bash нам возвращает выполненную команду или результат действия на экран. Или не возвращает и просто отрабатывает. И терминал ошибок не прощает, так что будьте внимательны.

А тут, в нашем конкретном случае мы пишем в начале файла, что нам необходимо запустить все, что дальше в файле bash-ем.

Это будет набор инструкций, где каждая инструкция будет идти с новой строки.

Вместо /bash после /bin может идти /pyton чтобы все открывалось в интерпретаторе пайтона. Либо java.

**Интерпрета́ция** — построчный анализ, обработка и выполнение исходного кода программы или запроса, в отличие от компиляции, где весь текст программы, перед запуском анализируется и транслируется в машинный или байт-код без её выполнения.

В принципе системе все-равно как и чем запускать код. Мы будем рассматривать только bash оболочку.

Почему стоит ее рассматривать? Bash есть практически во всех современных дистрибутивах. Кроме очень экстремальных, где каждый байт имеет значение. Но тогда там будет облегченная версия (кажется, куда уже меньше и проще, правда?).

Как это понять? Чем меньше букв в названии, тем более облегченная оболочка.

Но как узнать в какой оболочке мы работаем? Все просто. Если мы нажимаем на **TAB** и нам подсказывается слово, то будьте уверены - перед вами bash - полная версия.

Если вы нажимаете **TAB** и у вас нет подсказок, то перед вами облегченная версия оболочки. И оболочка может называться **ash.**

**cat /etc/os-release - проверить Версию Linux**

**Бывают настолько обрезанные дистрибутивы, что в них даже не показывает твой настоящий путь. Тоесть ты не видишь где ты находишься и тебе приходится постоянно перепроверяться командой pwd (которая укажет твой путь). Не говоря уже о подсказках через TAB.**

Но такие системы встречаются довольно редко и работать с ними некоторым не придется. Однако, мы с вами с ними будем сталкиваться в процессе обучения, когда будем говорить о “контейнерах”. Когда каждый скрипт может быть заперт в своем контейнере и контейнер видит себя как отдельную операционную систему. И там уже, для экономии места и ресурсов, иногда урезается **bash** и бывает самый простой пример оболочки **sh**.

Может возникнуть вопрос, а что если мы написали в файле **bash**, а его нету? Тогда система должна сама перенаправить на выполнение в оболочку **ash**.

Но это уже особенности определенных операционных систем.

Пока у нас есть **bash** и мы будем с ним работать.

И поэтому сценарии, с которыми мы будем работать так и называются **bash** скрипты.

Многие это даже называют программированием на bash, хотя это и не является настоящим программированием.

Ведь это набор инструкций с использованием переменных, которые вдруг нам могут понадобиться.

А теперь вернемся к учебному терминалу и попробуем написать парочку команд:

#!/bin/bash

echo Hello

date

Напомню, что команда **echo** выводит на экран текст, который мы напишем, а команда **date** - дату

Теперь выйдем и сохранимся:

**ctrl+x**

И проверим командой **ls**, что файл сохранился.

Вот, мы видим свой файл:

localhost:~# ls

bench.py hello.c hello.js readme.txt script.sh

И чтобы убедиться, что у нас файл содержит команды давайте введем команду:

localhost:~# cat script.sh

#!/bin/bash

echo Hello

date

Теперь введем команду:

localhost:~# ls -la

total 44

drwxr-xr-x 5 root root 263 Jan 9 2021 .

drwxrwxrwx 21 root root 461 Feb 15 14:45 ..

-rw------- 1 root root 40 Feb 15 14:45 .ash\_history

drwx------ 3 root root 61 Jul 6 2020 .cache

drwx------ 5 root root 124 Jul 6 2020 .mozilla

drwxr-xr-x 4 root root 202 Jul 6 2020 .wine

-rw-r--r-- 1 root root 114 Jul 5 2020 bench.py

-rw-r--r-- 1 root root 76 Jul 3 2020 hello.c

-rw-r--r-- 1 root root 22 Jun 26 2020 hello.js

-rw-r--r-- 1 root root 151 Jul 5 2020 readme.txt

-rw-r--r-- 1 root root 29 Feb 15 14:43 script.sh

И что мы видим? Наш файл, который мы создали не более чем текстовик. И его можно только прочитать и записать. И то, только суперадмину. Что нам делать? Ааааа! Паника!

Это еще не скрипт, а простой файл.

Что нам нужно сделать, чтобы он стал полноценным скриптом?

Что думаете, что нужно сделать?

Помните, мы затрагивали тему прав на файлы? Что есть некая команда chmod, которая позволяет раздавать права направо и налево?

**Итак, мы хотим добавить владельцу возможность исполнения.**

localhost:~# chmod u+x script.sh

Что это значит?

**chmod - change modification**

**u - user**

**+x - дать права на выполнение.**

**вместо U может быть:**

владельцу (u), группе (g), остальным (o) или всем (a). Можно указать как одну, так сразу несколько категорий.

Вторая группа означает запрещение, разрешение или назначение права и всегда состоит из одного символа: -, + или =.

В третьей группе перечисляется изменяемые права: чтение (r), запись (w), исполнение (x). Можно указывать сразу несколько прав.

Теперь проверим:

ls -la

total 44

drwxr-xr-x 5 root root 263 Jan 9 2021 .

drwxrwxrwx 21 root root 461 Feb 15 14:53 ..

-rw------- 1 root root 68 Feb 15 14:55 .ash\_history

drwx------ 3 root root 61 Jul 6 2020 .cache

drwx------ 5 root root 124 Jul 6 2020 .mozilla

drwxr-xr-x 4 root root 202 Jul 6 2020 .wine

-rw-r--r-- 1 root root 114 Jul 5 2020 bench.py

-rw-r--r-- 1 root root 76 Jul 3 2020 hello.c

-rw-r--r-- 1 root root 22 Jun 26 2020 hello.js

-rw-r--r-- 1 root root 151 Jul 5 2020 readme.txt

-rwxr--r-- 1 root root 29 Feb 15 14:43 script.sh

И у нас у владельца есть право на исполнение.

Это мы можем видеть по зеленому цвету файла. Если цвет белый - то это значит, что перед нами текстовик.

**Вообще, что касается цвета файлов:**

* **зеленый**: Исполняемые файлы.
* **белый**: Обычный файл.
* **синий**: Каталоги или папки.
* **Небесный**: Символическая ссылка.
* **красный**: Сжатые файлы (.tar, .gz, .zip, .rpm).
* **фуксин или фиолетовый**: Файлы изображений (.jpg, gif, bmp, png, tif)

## **Установка прав с помощью чисел**

Способ использования команды chmod, когда предоставляемые права выражаются одной восьмеричной цифрой для каждой категории пользователей, несколько проще с точки зрения записи. Однако надо помнить, что обозначают цифры:

7 – разрешены чтение, запись, исполнение

6 – разрешены чтение и запись

5 – разрешены чтение и исполнение

4 – разрешено только чтение

0 – ничего не разрешено

В первом аргументе chmod указываются три цифры: первая обозначает права владельца, вторая – группы, третья – остальных.

Пример с цифрами:

chmod 744 script.sh

localhost:~# ls -la

total 44

drwxr-xr-x 5 root root 263 Jan 9 2021 .

drwxrwxrwx 21 root root 461 Feb 15 15:04 ..

-rw------- 1 root root 96 Feb 15 15:04 .ash\_history

drwx------ 3 root root 61 Jul 6 2020 .cache

drwx------ 5 root root 124 Jul 6 2020 .mozilla

drwxr-xr-x 4 root root 202 Jul 6 2020 .wine

-rw-r--r-- 1 root root 114 Jul 5 2020 bench.py

-rw-r--r-- 1 root root 76 Jul 3 2020 hello.c

-rw-r--r-- 1 root root 22 Jun 26 2020 hello.js

-rw-r--r-- 1 root root 151 Jul 5 2020 readme.txt

-rwxr--r-- 1 root root 29 Feb 15 14:43 script.sh

Разница заключается в том, что при помощи цифр мы передает то состояние, которое мы хотим. То есть у владельца - все возможное, у группы чтение исполнение. Мы сразу передаем то состояние, которое мы хотим видеть у файла.

Если мы будем делать первым способом, то нам нужно несколько раз указывать, что и кому мы хотим дать. **Второй способ с цифрами чуть быстрее.**

Есть еще один трюк, который не совсем безопасный, но очень быстрый.

localhost:~# chmod +x script.sh

Этим мы дадим всем права на запуск файла.

Теперь вернемся к нашему скрипту, который написан в файлике.

Есть 3 способа запустить наш скрипт.

1. Если мы находимся в той же папке, что и наш скрипт, то:

localhost:~# ./script.sh

тут мы указываем, где взять в нашей папке то, что запустить.

Если мы введем название скрипта (файла), просто в терминал, то нас скажет, что файл не найден. Ведь его не ввели в переменные.

echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

Помните мы говорили про переменные?

Так вот,запуская через точку слеш мы говорим системе, что нам необходимо в этой же папке взять наш файл и запустить.

1. Запустить через полный путь:

localhost:~# pwd

/root

localhost:~# /root/script.sh

Hello

Tue Feb 15 15:17:46 UTC 2022

1. запустить через bash:

localhost:~# bash script.sh

Hello

Tue Feb 15 15:18:57 UTC 2022

Но третий не совсем правильный. Скорее запасной. Все дело в том, что мы делаем бесполезную ссылку на то, чем мы этот скрипт запускаем.

localhost:~# cat script.sh

#!/bin/bash

echo Hello

date

Помните, вначале мы когда писали скрипт, мы говорили системе, чем нужно запускать скрипт?

И если мы в терминале пишем **bash**, то тем самым исключаем нашу первую строчку из текстового файла. А значит выполнили лишнее действие и потратили больше времени. Не совсем правильно.

И в этом виде интерпретатор будет относиться к текстовому файлу не как к скрипту, а как к простому текстовому файлу с двумя командами.

Однако есть и бонус в этой записи. Если, например, в нашем файле нету строки #!/bin/bash, то скрипт все равно отработает. И еще, если мы принудительно запускаем bash-ом, то файл может быть и не исполняемым. То есть есть свои плюсы.

Но пользоваться этим нужно осторожно и понимая когда и зачем.

Этим можно сэкономить время, но это не является полноценным скриптом.

Окей, запустили, теперь давайте мы модифицируем наш скрипт.

Зайдем еще раз в наш файл:

nano script.sh

#!/bin/bash

CITY=Berlin

echo Hello from $CITY

date

Что я сейчас сделал?

Сохранимся и выйдем.

И запустим наш скрипт:

localhost:~# ./script.sh

Hello from Berlin

Tue Feb 15 15:31:45 UTC 2022

Помните, когда мы говорили про переменные, мы делали так:

export MYVAR=hello

export NAME=Ivan

Но в скрипте мы пишем проще:

CITY=Berlin

Но вызываем мы переменную через значок доллара. Напомню, что переменные пишутся большими буквами, но это джентельменское соглашение - не более. Вы можете назначать их и маленькими буквами, но если хотите, чтобы ваш код был более понятен для остальных - пишите большими.

И не забывацте, где нужно ставить пробелы, а где нет. Например, тут после CITY=Berlin пробел ставить нельзя.

Можно попробовать, кстати.

Но если вам уж очень нужен пробел, тогда его нужно поставить в кавычки:

CITY=" Berlin"

echo Hello from $CITY

date

И все заработает.

Двигаемся дальше.

**Циклы**.

Если вы загуглите использование циклов в bash, то вам вывалится огромное количество вариантов.

Давайте я покажу самый просто из вариантов и если вы его запомните - уже будет здорово.

Вы можете использовать разные стили написания. Например стиль Си. Ведь, если вы помните, все, что мы сейчас вводим и пишем выросло из языка Си.

Но есть чисто BASH стиль.

Что такое циклы? Это повторяющиеся действия.

Любой уважающий себя язык программирования должен содержать **циклы**. **Цикл** - это такая последовательность, которая позволяет выполнять определенный участок кода необходимое количество раз. С помощью **циклов** вы можете очень сильно сократить количество строк кода, которые необходимо написать для однотипных операций.

То, что мы сейчас рассмотрим, нормально работает как с массивами, так же и со списками файлов в директории.

#!/bin/bash

CITY=" Berlin"

echo Hello from $CITY

date

for run in {1..10}

do

echo $run

done

Что мы делаем. Мы объявляем переменную с маленькими буквами run, вначале мы ее нигде не заводили и ничего не присваивали.

Мы прям сразу на ходу ее заводим, называем ее run и это {1..10} это стандартная конструкция минимального-максимального значения массива.

Где мы говорим, что мы заводим новую переменную, которая называется run и она принимает значение от 1 до 10.

Если мы пишем от 1 до 10, то с шагом 1 будет выполнятся 10 раз. И начало может быть не обязательно с 1(спойлер на домашку).

Тут мы говорим с какого начинаем и каким заканчиваем. И не обязательно это 1-10.

Еще раз - эта запись нам говорит, что массив будет с приростов в единицу от и до.

То есть у нас есть переменная run, она может быть от 1 до 10 и будет крутиться с шагом в единичку.

Все, что между do и done будет повторяться количество раз, которое указано в {1..10}

**В данном случае 10 раз.**

Давайте выйдем с сохранением и запустим:

localhost:~# ./script.sh

Hello from Berlin

Tue Feb 15 16:00:47 UTC 2022

1

2

3

4

5

6

7

8

9

10

localhost:~# cat script.sh

#!/bin/bash

CITY=" Berlin"

echo Hello from $CITY

date

for run in {1..10}

do

echo $run

done

У нас есть переменная run, к которой мы здесь же и обращаемся, говоря “а выведи нам значение”.

И тут у нас первая итерация:

Итерация ([лат.](https://ru.wikipedia.org/wiki/%2525D0%25259B%2525D0%2525B0%2525D1%252582%2525D0%2525B8%2525D0%2525BD%2525D1%252581%2525D0%2525BA%2525D0%2525B8%2525D0%2525B9_%2525D1%25258F%2525D0%2525B7%2525D1%25258B%2525D0%2525BA) [iteratio](https://ru.wiktionary.org/wiki/iteratio%2523%2525D0%25259B%2525D0%2525B0%2525D1%252582%2525D0%2525B8%2525D0%2525BD%2525D1%252581%2525D0%2525BA%2525D0%2525B8%2525D0%2525B9) «повторение») — повторение какого-либо действия.

run = 1 и мы говорим - выведи его.

Оболочка выводит. Смотрит на конечное число, сколько раз нужно повторить этот цикл и делает еще раз. Только run уже равно 2 и так далее.

После чего мы закончили.

Но у нас отработало все настолько быстро, что мы не успели ничего заметить. А как же драма? Давайте ее добавим! (=

localhost:~# nano script.sh

#!/bin/bash

CITY=" Berlin"

echo Hello from $CITY

date

for run in {1..10}

do

echo $run

sleep 1

done

Как мы видим, я добавил sleep 1, а это означает, что система будет ждать 1 секунду после выполнения одного действия.

Обратите внимание, как теперь выполняется наш код. Прям как в фильмов про хакеров.

Так же тут нормально работают дробные значения. Можно указать менее секунды.

**Если вдруг необходимо прервать процесс** - мы нажимает ctrl+c

создадим файл, допишем в него дату и перенесем его в папку opt/3/ c переименовыванием. Так же напишем, что при удачном переносе нам на экран выведется Done!:

#!/bin/bash

touch /tmp/2.txt

date >> /tmp/2.txt

mkdir -p /opt/3

mv /tmp/2.txt /opt/3/newfile.txt

echo "Done!"

*#!/bin/bash*

*echo -n "Enter your name: "*

*read name*

*echo "Hello $name, welcome to my program."*

*Обратите внимание на то, что команда echo, которая выводит приглашение, вызывается с ключом -n. Это приводит к тому, что в конце приглашения не выводится знак перевода строки, что позволяет пользователю скрипта вводить данные там же, где расположено приглашение, а не на следующей строке.*

*давайте не будем голословными, а попробуем убрать ключ -n и снова запустим файл.*

#!/bin/bash

echo Наш первый скрипт:

echo Вы:

whoami

echo Система стартовала:

uptime

echo Сегодня:

date

echo bye

**Процессы.**

Что такое процессы в терминале?

Любая программа, запущенная в ОС будет процессом. Иногда эти процессы виснут, ведут себя странно и их нужно отслеживать.

Как же посмотреть список процессов?

Я сейчас зайду на наш учебный сервер:

**localhost:~# chmod 744 script.sh**

и введу команду **top**

**top** (table of processes) — консольная команда, которая выводит список работающих в системе процессов и информацию о них. По умолчанию она в реальном времени сортирует их по нагрузке на процессор.

так же мы можем посмотреть количество процессов через команду

**ps -ef**

Вот значение основных колонок в выводе утилиты:

* **UID** - имя пользователя, от имени которого работает процесс;
* **PID** - идентификатор пользователя;
* **PPID** - идентификатор родительского процесса пользователя;
* **C** - расходование ресурсов процессора, в процентах;
* **SZ** - размер процесса;
* **RSS** - реальный размер процесса в памяти;
* **PSR** - ядро процессора, на котором выполняется процесс;
* **STIME** - время, когда процесс был запущен;
* **TTY** - если процесс привязан к терминалу, то здесь будет выведен его номер;
* **TIME** - общее время выполнения процесса (user + system);
* **CMD** - команда, которой был запущен процесс, если программа не может прочитать аргументы процесса, он будет выведен в квадратных скобках;

Чтобы посмотреть список процессов в виде дерева, и понимать какой процесс имеет какие дочерние процессы, выполните команду:

**ps -efH**

Для просмотра списка процессов с потоками используйте опцию -L:

**ps -efL**

Здесь появятся ещё две дополнительных колонки:

* **LWP - Это сокращение от LightWeight Process. Идентификатор потока;**
* **NLWP - количество потоков у этого процесса.**

### **УТИЛИТА TOP**

Программа позволяет интерактивно просматривать список запущенных процессов Linux. Чтобы вывести список процессов Linux выполните команду:

**top**

Колонки, которые выводит программа очень похожи на ps:

* **PID -** идентификатор процесса;
* **USER -** имя пользователя, от имени которого выполняется процесс;
* **PR -** приоритет планировщика, установленный для процесса;
* **NI -** рекомендуемый приоритет процесса. Это значение можно менять, может не совпадать с реальным приоритетом планировщика;
* **VIRT -** всё, что находится в памяти, используется или зарезервировано для использования;
* **RES -** всё, что находится в оперативной памяти и относится к процессу. Расшифровывается как Resident Memory Size, указывается в килобайтах;
* **SHR -** часть памяти из RES, которую занимают ресурсы, доступные для использования другим процессам. Расшифровывается - Shared Memory Size.
* **S -** состояние процесса: D - ожидает завершения операции, R - запущен, S - спит, T - остановлен, t - остановлен отладчиком, Z - зомби;
* **%CPU -** процент использования ресурсов процессора;
* **%MEM -** процент использования ресурсов оперативной памяти на основе колонки RES;
* **TIME -** общее процессорное время, которое процесс использовал с момента запуска;
* **COMMAND** - команда, с помощью которой был запущен процесс.
* **Проце́сс-зо́мби, зо́мби** ([англ.](https://ru.wikipedia.org/wiki/%2525D0%252590%2525D0%2525BD%2525D0%2525B3%2525D0%2525BB%2525D0%2525B8%2525D0%2525B9%2525D1%252581%2525D0%2525BA%2525D0%2525B8%2525D0%2525B9_%2525D1%25258F%2525D0%2525B7%2525D1%25258B%2525D0%2525BA) zombie process, [англ.](https://ru.wikipedia.org/wiki/%2525D0%252590%2525D0%2525BD%2525D0%2525B3%2525D0%2525BB%2525D0%2525B8%2525D0%2525B9%2525D1%252581%2525D0%2525BA%2525D0%2525B8%2525D0%2525B9_%2525D1%25258F%2525D0%2525B7%2525D1%25258B%2525D0%2525BA) defunct process) — дочерний процесс в [Unix](https://ru.wikipedia.org/wiki/Unix)-системе, завершивший своё выполнение, но ещё присутствующий в списке процессов [операционной системы](https://ru.wikipedia.org/wiki/%2525D0%25259E%2525D0%2525BF%2525D0%2525B5%2525D1%252580%2525D0%2525B0%2525D1%252586%2525D0%2525B8%2525D0%2525BE%2525D0%2525BD%2525D0%2525BD%2525D0%2525B0%2525D1%25258F_%2525D1%252581%2525D0%2525B8%2525D1%252581%2525D1%252582%2525D0%2525B5%2525D0%2525BC%2525D0%2525B0), чтобы дать родительскому процессу считать код завершения.

**Для того чтобы сделать вывод программы цветным, нажмите Z**

Чтобы вывести дерево процессов Linux нажмите сочетание клавиш **Shift+V:**

Для отображения потоков нажмите **Shift + H:**

Выйдем из TOP нажмем **ctrl+z**

и введем:

**ps -ef | wc -l**

Так мы посмотрим сколько процессов запущено в цифрах.

И мы можем отфильтроваться по слову ssh:

**ps -ef | grep ssh**

Тут мы видим наш процесс, который был запущен нашей системой:

**root 4744 1 0 Feb01 ? 00:00:03 /usr/sbin/sshd -D**

Это значит, что система ожидает, когда к ней подключатся другие пользователи по ssh.

И мы видим идентификатор. После слова root.

Далее первый процесс запускает второй:

**root 4744 1 0 Feb01 ? 00:00:03 /usr/sbin/sshd -D**

**root 19727 4744 0 12:08 ? 00:00:00 sshd: ec2-user [priv]**

И тут мы видим, что идентификаторы процесса совпадают.

Итак. Теперь о наших процессах. Если мы сами что-то запускаем на сервере и мы не является администраторами, то мы можем завершать процессы командой **kill**

Всем сделать аккаунт на:

**github.com**

Пароль держите при себе, желательно на рабочем столе.

Статьи:

<https://habr.com/ru/post/423049/> - процессы в Linux

<https://habr.com/ru/company/ruvds/blog/325522/> - скрипты

**Homework:**

Создайте скрипт sleeper.sh, который будет 10 раз с интервалом в 5 секунд писать дату в формате HH:MM:SS и количество процессов одним числом.

Уменьшите или уберите временной интервал (который нам дает sleep), используя vi или nano, закомментировав строку или поменяв значение sleep

С помощью скрипта запишите в файл информацию об операционной системе, но отфильтруйте информацию так, чтобы осталось только имя (NAME=Alpine Linux).

Выполните прошлое действие, но так, чтобы слово NAME= не осталось, а было только имя в чистом виде (Alpine)

С помощью скрипта создайте 50 файлов с расширением txt и именами от 50.txt до 100.txt

Вот что вам понадобится для реализации всего этого:

Если удобно, то вот однострочник, создающий простейший скрипт, который надо будет редактировать:

**echo -e `#!/bin/bash\n date\n echo** “**it works!**“**` > /tmp/script.sh**

**a вот расширенный вариант и то, что может использоваться:**

for run in {1..10}

do

#Comment section

#here are your commands and actions

done

Другие команды:

sleep

touch

ps -ef

date +“%H:%M:%S”

cat /etc/os-release

grep

awk ‘{print$НОМЕР\_СТОЛБЦА}’