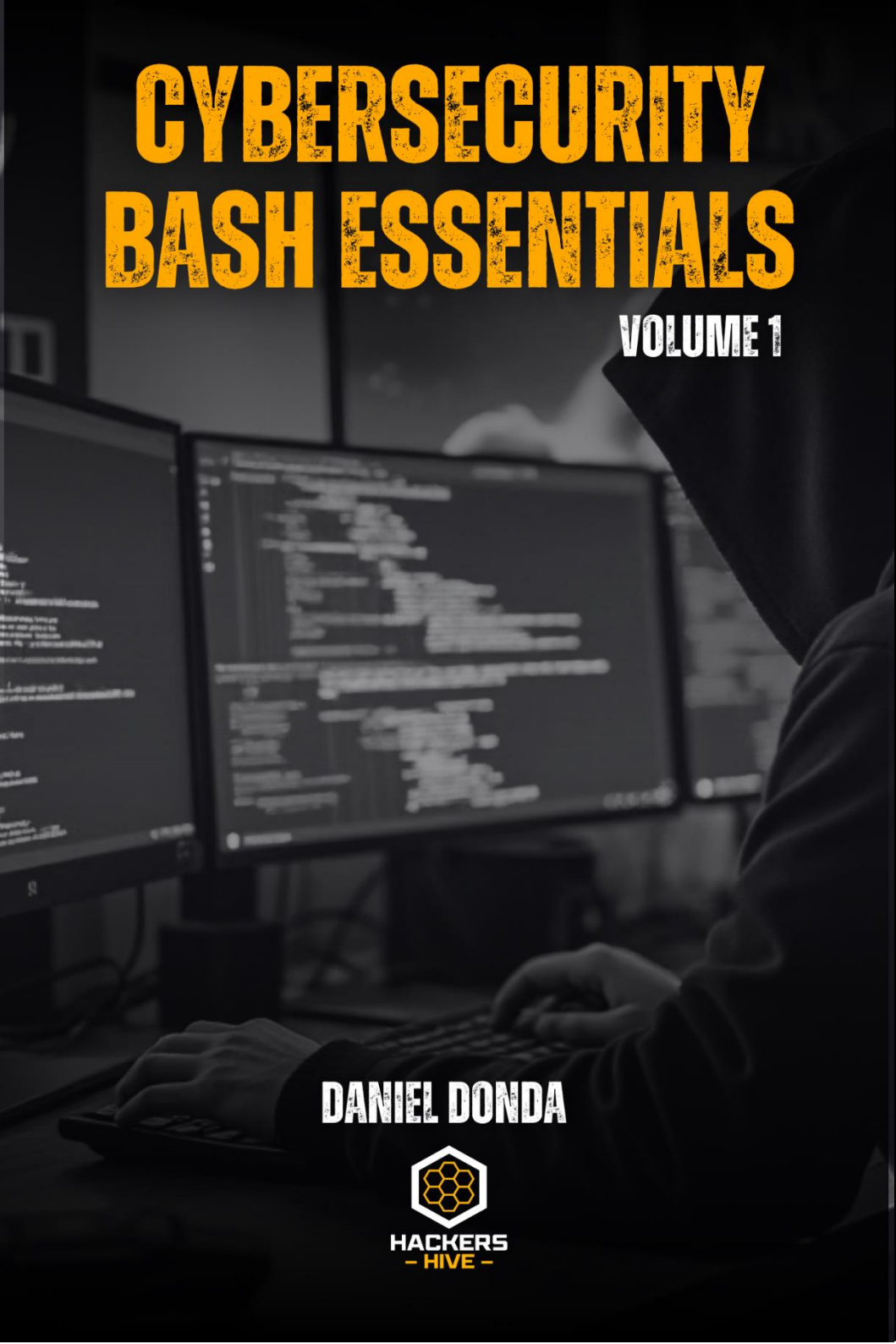
CYBERSECURITY BASH ESSENTIALS

VOLUME 1



DANIEL DONDA

HACKERS HIVE

CYBERSECURITY BASH ESSENTIALS

...

Volume 1

Daniel Donda
Hackers Hive

Primeira Edição – dezembro de 2024

Sumário

Introdução

Esse material é parte do treinamento Cybersecurity Bash Essentials. Durante o treinamento nós temos a oportunidade de ver na prática a aplicação de vários dos conceitos citados aqui neste material, obviamente este ebook contém recursos até mais detalhados para fins de pesquisa e estudos.

Você pode baixar os scripts utilizados no treinamento em nosso repositório no GitHub e então praticar em seu ambiente de laboratório.

É muito importante ressaltar que este ebook não contém todos os ensinamentos aplicados durante o curso este book, Ele foi criado para conter apenas os princípios fundamentais necessários para começar a trabalhar com Shell Script. O curso aborda além desses conceitos scripts para uso por profissionais de cibersegurança.

O que é o Shell ?

O Shell é um programa que permite a interação entre o usuário e o sistema operacional. Ele interpreta os comandos digitados pelo usuário e os envia para o núcleo (kernel) do sistema, onde serão executados. Em sistemas baseados em Unix, como Linux e macOS, o shell é a interface padrão para operar o sistema em um ambiente de linha de comando.

Existem diferentes tipos de shells, cada um com características próprias.

- Sh (Bourne Shell): Shell mais antigo e predecessor do Bash.
- Bash (Bourne Again Shell) - Amplamente utilizado em sistemas Linux, é conhecido por sua simplicidade e compatibilidade com diversos sistemas.
- Zsh (Z Shell): Conhecido por ser altamente customizável e possui funcionalidades avançadas.
- Ksh (Korn Shell): Popular em sistemas Unix e considerado robusto para scripts complexos.
- Tcsh (Tenex C Shell): Baseado no C Shell, com adições que melhoram a usabilidade e edição de linha.
- Fish (Friendly Interactive Shell): Um shell moderno que busca ser mais amigável ao usuário.

O que é Bash?

Definição: Bash (Bourne Again SHell) é um interpretador de comandos amplamente utilizado em sistemas Unix e Linux. Ele permite que os usuários interajam diretamente com o sistema operacional através de comandos de texto.

Características principais:

- **Interface de linha de comando (CLI)**: Permite interação direta com o sistema, ideal para automação e manipulação de arquivos.
- **Scripting poderoso**: Bash é uma linguagem de script, o que permite automatizar tarefas repetitivas.
- **Portabilidade**: Scripts Bash podem ser usados em diferentes sistemas Unix-like, como Linux e macOS.

Tem diferenças entre o Bash e ZShell ?

Sim, há algumas diferenças quando se cria scripts para rodar no **Zsh** em comparação com o **Bash**. Em geral, ambos compartilham muitos comandos e sintaxes, mas existem particularidades no Zsh que podem afetar o comportamento dos scripts. Aqui estão algumas diferenças importantes:

- **Sintaxe de arrays** - No Zsh, a indexação de arrays começa em **1** por padrão, enquanto no Bash começa em **0**. Isso pode causar resultados inesperados se você não adaptar o código para o Zsh.
- **Redirecionamento de saída** - O Zsh usa um redirecionamento &| para combinar a saída padrão e de erro em um único fluxo. No Bash, o equivalente é &>.
- **Substituição de variáveis** - O Zsh permite a expansão de variáveis mais avançada e algumas substituições (${(U)variable} para maiúsculas, por exemplo), que não funcionam no Bash.
- **Trabalhando com globbing** - O Zsh possui um sistema de globbing mais avançado, permitindo seleções mais refinadas de arquivos e diretórios. No Zsh, você pode usar glob qualifiers como *.log(.om[1,5]) para selecionar

os 5 arquivos .log mais antigos, algo que o Bash não suporta diretamente.

- **Compatibilidade com Bash** - Para garantir que scripts funcionem em ambos, muitas pessoas usam #!/bin/bash ou #!/bin/zsh no cabeçalho e evitam recursos específicos de um shell. Scripts complexos podem precisar de ajustes caso usem funcionalidades específicas de cada shell.

O Shell Script é uma ferramenta essencial para administradores de sistemas e profissionais de cibersegurança. Ao dominar scripts, você pode automatizar tarefas repetitivas, gerenciar configurações e reagir rapidamente a incidentes de segurança. Este capítulo apresenta os conceitos fundamentais do Shell Script e como ele pode ajudar a otimizar sua rotina no contexto da segurança da informação.

O que é um Shell Script?

Um Shell Script é um arquivo de texto contendo uma sequência de comandos que são executados de forma automatizada pelo interpretador do Shell. Ele é amplamente utilizado para simplificar processos complexos, automatizar tarefas administrativas e até mesmo implementar soluções criativas para desafios de cibersegurança, como a análise de logs ou a varredura de redes.

No contexto da cibersegurança, o Shell Script pode ser usado para:
- Analisar grandes volumes de dados em busca de anomalias.
- Automatizar testes de segurança em redes e sistemas.
- Criar relatórios rápidos e eficientes.

Consultado o shell do seu sistema

Para verificar qual Shell está ativo no momento, você pode consultar o terminal usando a variável de ambiente correspondente. Execute o comando abaixo para descobrir o Shell em uso

```
echo $SHELL
```

Este comando exibe o caminho completo do Shell configurado como padrão para o seu usuário. Por exemplo, se você estiver usando o Bash, o retorno será algo como /bin/bash.

⚡ Nota: Neste comando temos uma variável, iremos no aprofundar em variáveis mais adiante.

Como verificar a versão do Shell

Muitas vezes, é útil saber a versão exata do Shell, especialmente se você estiver usando recursos específicos de uma versão mais recente. Para verificar a versão, utilize os comandos abaixo, dependendo do seu Shell:

Para o Bash: `bash -version`
Para o Zsh: `zsh --version`

Por que consultar o Shell é importante?

Diferentes Shells podem interpretar comandos e sintaxes de maneira distinta. Saber qual Shell está ativo ajuda a evitar erros de compatibilidade ao executar scripts.

Alguns arquivos de configuração, como .bashrc ou .zshrc, são específicos para cada tipo de Shell.

🌟 **Dicas e Curiosidades**

Sempre use o comando man (manual) para aprender mais sobre os comandos disponíveis no seu Shell. Por exemplo, man bash exibe o manual completo do Bash.

O Zsh é o Shell padrão no Kali Linux devido aos seus recursos avançados, como autocompletar e plugins, mas o Bash ainda é amplamente utilizado por sua simplicidade e compatibilidade.

Consultando os Shells válidos no sistemas

O arquivo /etc/shells é um arquivo de configuração no Linux e outros sistemas baseados em Unix que contém uma lista de todos os Shells disponíveis no sistema. Ele é utilizado para determinar quais Shells são válidos para serem usados por um usuário.

```
cat /etc/shells
```

Alterando o Shell

Você pode alterar o Shell simplesmente digitando o nome dele no terminal por exemplo se você estiver usando o kali Linux

muito provavelmente o Shell é zsh, e para você alterar para o bash basta digitar bash e enter, exemplo:

```
bash
```

Você pode usar o comando chsh para alterar o Shell padrão de um usuário. Apenas os Shells listados em /etc/shells podem ser definidos.

```
chsh -s /bin/bash
```

SCRIPTS

Exemplo Básico: Script de Boas-Vindas

Como primeiro passo, vamos criar um script simples que exibe uma mensagem de boas-vindas. Este exemplo ajudará você a entender como salvar e executar um Shell Script.

```
#!/bin/bash
# Este é um script simples de boas-vindas
echo "Bem-vindo ao curso Cybersecurity Bash Essentials!"
```

Para executar o script, siga estas etapas:

Salve o conteúdo acima em um arquivo chamado 'bem_vindo.sh'.
Torne o arquivo executável com o comando: **chmod +x**

`bem_vindo.sh`
Execute o script com: **`./bem_vindo.sh`**

Dicas e Curiosidades

- Adicione comentários usando o caractere '#'. Isso ajuda a documentar seu código e torná-lo mais legível.
- Em cibersegurança, scripts podem ser usados para tarefas como detecção de vulnerabilidades ou monitoramento de atividades suspeitas.

🌟 Curiosidade: A palavra 'Shell' refere-se ao ambiente de linha de comando que atua como uma interface entre o usuário e o sistema operacional. O Bash é um dos Shells mais populares, amplamente usado em sistemas Linux.

shebang (#!)

O **shebang** (#!) é uma sequência de caracteres usada no início de scripts em sistemas Unix e Linux para definir qual interpretador deve ser utilizado ao executar o script. Essa linha inicial, chamada de "shebang" (ou "hashbang"), ajuda o sistema operacional a identificar o interpretador correto, como Bash, Zsh, Python, entre outros, que devem processar o conteúdo do script.

Estrutura do Shebang

O shebang é seguido pelo caminho do interpretador. Por exemplo:

`#!/bin/bash` - Indica que o script será executado com o Bash.
`#!/bin/zsh` - Indica que o script será executado com o Zsh.

Como Funciona o Shebang

Quando você executa um script, o sistema lê a primeira linha:

- Se a linha começar com #!, ele usa o caminho especificado para iniciar o interpretador.
- Se não houver shebang, o sistema usará o shell padrão (normalmente o Bash) para executar o script.

Lembre-se que no Kali o padrão é o zsh e no Parrot o padrão é o Bash

Vantagens do Uso do Shebang

- **Portabilidade** - Você pode criar scripts que funcionam em diferentes ambientes, definindo claramente o interpretador.
- **Evitar problemas de compatibilidade** - O shebang garante que o script seja executado pelo shell ou linguagem esperada, evitando erros de sintaxe ou comportamento inesperado.

🌟 Curiosidade: O Bash foi criado em 1989 por Brian Fox e é um software livre amplamente utilizado em servidores, sistemas embarcados e até em algumas distribuições Linux populares.

Gerenciamento de Usuários e Grupo

Eu já disse bem no início do deste material que por mais que o curso tenha o foco em scripts para cibersegurança, se faz necessário o conhecimento mesmo que básico de operações do sistema como por exemplo o gerenciamento de usuário e de grupos. O gerenciamento de usuários e grupos é essencial para controlar o acesso aos recursos do sistema em ambientes Linux. Esta prática é especialmente importante em cibersegurança, pois ajuda a garantir que apenas usuários autorizados tenham acesso a informações sensíveis. Neste capítulo, você aprenderá a criar, gerenciar e excluir usuários e grupos, bem como os comandos essenciais para essa tarefa.

O Usuário root

O usuário `root` é o superusuário do sistema, possuindo privilégios completos e sem restrições. Ele desempenha um papel crítico na administração do sistema, mas deve ser usado com cautela devido ao risco de causar mudanças irreversíveis.
Características do usuário `root`:
- UID (User ID): 0.
- Acesso total para executar comandos, alterar configurações críticas e acessar todos os arquivos do sistema.
- É associado ao grupo `root`.

O Grupo root

O grupo `root` é um grupo especial com GID (Group ID) 0. Embora seja associado principalmente ao usuário `root`, outros usuários podem ser adicionados ao grupo para herdar privilégios específicos dependendo da configuração do sistema.

Comandos para Privilégios Elevados

Existem diferentes formas de acessar privilégios elevados em sistemas Linux. Aqui estão os comandos mais comuns:

Comando	Descrição	Exemplo
su	Muda para outro usuário, geralmente root.	su root
sudo	Executa um comando com privilégios elevados.	sudo apt update
sudo su	Usa sudo para executar o su.	sudo su

Criando e Gerenciando Usuários

Você pode criar usuários no sistema usando o comando **useradd** com privilégios elevados:

```
sudo useradd usuario1
sudo useradd -m -s /bin/bash usuario3
```

Após criar o usuário, defina uma senha inicial com o comando:
`sudo passwd Senha123`

Para listar todos os usuários no sistema, você pode consultar o arquivo “/etc/passwd”
cat /etc/passwd

Para excluir um usuário e seus arquivos associados, use:

sudo userdel -r usuario1

Criando e Gerenciando Grupos

Você pode criar novos grupos com o comando **groupadd** usando privilégios elevados:
sudo groupadd pentest

Para adicionar um usuário a um grupo, use o comando:
sudo usermod -aG pentest usuario1

Para remover um usuário de um grupo, use:
sudo gpasswd -d usuario1 pentest

Configurações de Senhas

O comando chage é usado para gerenciar e consultar informações sobre senhas de usuários. Por exemplo:
chage -l usuario1

Para forçar a alteração da senha no próximo logon:
sudo chage -d 0 usuario1

Para apagar a senha de um usuário:
sudo passwd -d usuario1

stdin, stdout e stderr

No Linux, quase todas as interações entre o sistema operacional e os programas envolvem três fluxos padrão: **stdin (entrada padrão)**, **stdout (saída padrão)** e **stderr (saída de erro padrão)**. Esses fluxos são os pilares para a comunicação entre comandos, arquivos e dispositivos no sistema. Compreender como esses fluxos funcionam é essencial para manipular dados de maneira eficiente e construir scripts robustos. No contexto de cibersegurança, saber redirecionar e gerenciar esses fluxos permite que você processe logs, capture erros e automatize tarefas críticas.

stdin (Standard Input): É o fluxo de entrada padrão de dados, geralmente vindo do teclado ou de um arquivo.
Exemplo:

```
echo "Digite seu nome:“
read nome
```

stdout (Standard Output): É o fluxo de saída padrão, que exibe dados no terminal.
Exemplo:

```
echo "Este texto será exibido no terminal“
echo "Salvando no arquivo" > arquivo.txt
```

stderr (Standard Error): É o fluxo de saída de erros, usado para exibir mensagens de erro separadamente do stdout.

Exemplo:

```
ls /etc > sucesso.txt 2> erro.txt
```

Códigos de erro

Código de Erro	Descrição
0	Execução bem-sucedida, sem erros
1	Erro geral ou falha genérica
2	Arquivo ou diretório não encontrado
126	Comando não executável
127	Comando não encontrado
128	Argumento inválido para o comando exit
130	Terminado com Ctrl+C (interrupção do teclado)
137	Terminado com kill (por exemplo, kill -9)

Com esses conceitos, você poderá controlar e monitorar as operações no Linux de maneira precisa, aprimorando suas habilidades em automação e análise de sistemas. Vamos usá-los mais adiante.

Trabalhando com Arquivos

Trabalhar com arquivos no Linux é uma habilidade essencial tanto para administração de sistemas quanto para cibersegurança. Arquivos podem conter logs críticos, dados

sensíveis e configurações fundamentais para a segurança do ambiente. Neste capítulo, exploraremos os principais comandos para criar, visualizar, editar, mover, copiar e excluir arquivos, com foco em práticas que ajudam a proteger informações e evitar ações acidentais que possam comprometer o sistema. Com esses conhecimentos, você estará preparado para gerenciar arquivos de maneira eficiente, segura e alinhada às melhores práticas de cibersegurança."

Listando arquivos

O comando ls (list) é um dos mais básicos e utilizados no Linux. Ele é usado para listar os arquivos e diretórios de um diretório específico. No contexto de cibersegurança, o comando ls é útil para verificar rapidamente os conteúdos de pastas e entender a estrutura de arquivos, permissões e propriedades de um sistema.

Exemplo:
`ls [opções] [diretório]`

Exibir Arquivos **`ls`**
Exibir Arquivos e Diretórios Ocultos (-a ou --all): **`ls -a`**
Listagem Detalhada (-l): **`ls -l`**
Ordenar por Tamanho (-S): **`ls -lS`**
Exibir Tamanhos Humanamente Legíveis (-h): **`ls -lh`**
Classificar por Data de Modificação (-t): **`ls -lt`**

Coringas

Os **coringas** (ou **wildcards**) são caracteres especiais que ajudam a combinar padrões em comandos no terminal. Eles são frequentemente usados para trabalhar com vários arquivos ou diretórios de uma só vez, simplificando operações como listagem, cópia, remoção, renomeação e muito mais. Em cibersegurança, os coringas são úteis para manipular grandes volumes de arquivos de forma eficiente e rápida.

Coringa	Descrição	Exemplo
*	Substitui zero ou mais caracteres	ls arquivo*.txt
?	Substitui exatamente um caractere	ls arquivo?.txt
[ ]	Representa um conjunto de caracteres	ls arquivo[1-3].txt
[!]	Representa qualquer caractere **exceto** os especificados	ls arquivo[!1-3].txt
{ }	Expande uma lista de itens separados por vírgulas	echo {arq1,arq2,arq3.txt}
~	Representa o diretório inicial do usuário (home)	cd ~/Documentos

Criando Arquivos

Existem várias formas de criar arquivos no Linux, dependendo da sua necessidade. Aqui estão alguns comandos básicos:

Criar um arquivo vazio com o comando touch:
`touch arquivo1.txt`
Criar um arquivo e adicionar conteúdo imediatamente com o comando echo:
`Echo “#!/bin/bash” > scritpt.txt`

Redirecionamento de Saída (> e >>) e Entrada (<)

Os operadores de redirecionamento no Linux são ferramentas poderosas que permitem controlar como os dados fluem entre comandos, arquivos e dispositivos. Eles desempenham um papel central na automação de tarefas e manipulação de dados em scripts e terminais.

Redirecionamento de Saída (> e >>)

O operador > redireciona a saída padrão (**stdout**) de um comando para um arquivo, substituindo o conteúdo existente no arquivo. Já o operador >> faz o mesmo, mas **acrescenta** a saída ao arquivo, sem apagar os dados já existentes.

Uso básico, sobrescrever Arquivo com >

```
echo "Hello, World!" > arquivo.txt
```

Escreve "Hello, World!" no arquivo arquivo.txt. Se o arquivo já existia, seu conteúdo é substituído.

Acrescentar ao Arquivo com >>

```
echo "Outra linha" >> arquivo.txt
```

Adiciona "Outra linha" ao final do arquivo arquivo.txt, sem apagar o conteúdo anterior.

Outros exemplos:

```
ls > lista_diretorios.txt
date >> log.txt
```

Redirecionando para o Vazio (/dev/null)

No sistema Linux, /dev/null é conhecido como o "buraco negro" da entrada e saída. Ele é um arquivo especial que descarta tudo o que é redirecionado para ele, sem gerar nenhum tipo de saída ou armazenamento. É amplamente utilizado para descartar dados desnecessários ou para suprimir mensagens que não são relevantes durante a execução de scripts.

Como Funciona /dev/null?

Ao redirecionar a saída de um comando para /dev/null, você instrui o sistema a ignorar essa saída. Tudo o que for enviado para /dev/null é simplesmente descartado

Assim, se você não quiser que a saída padrão de um comando apareça no terminal, pode redirecioná-la para /dev/null

```
comando > /dev/null
```

Exemplo, para ignorar mensagens de erro, redirecione o fluxo de erro (stderr) para /dev/null

```
ping -c 1 192.168.1.1 2> /dev/null
```

ping -c 1 192.168.1.1 Envia 1 pacote ICMP para o endereço IP
2> /dev/null redireciona mensagens de erro para /dev/null,

Se o endereço IP estiver inacessível, nenhuma mensagem de erro será exibida.

Esse recurso será muito utilizado nos scripts.

Redirecionamento de Entrada (<)

O operador < redireciona a entrada padrão (stdin) de um comando para vir de um arquivo. Isso substitui a necessidade de digitar dados manualmente no terminal.

Uso básico usar um arquivo como entrada de um comando:

```
wc < arquivo.txt
```

Exemplo ordenar um arquivo e salvar a saída:

```
sort < arquivo.txt > arquivo_ordenado.txt
```

Visualizando Arquivos

Para visualizar o conteúdo de arquivos, o Linux oferece uma série de comandos úteis. Aqui estão os principais:

Usar cat para exibir o conteúdo de um arquivo:

```
cat arquivo1.txt
```

Exibir o conteúdo de um arquivo página por página com less:

```
less arquivo1.txt
```

ou

```
cat arquivo | less
```

Exibir as primeiras ou últimas linhas de um arquivo com head ou tail:

```
head arquivo1.txt
tail  arquivo1.txt
```

Neste exemplo vamos veremos as 5 primeiras ou últimas linhas

```
head -n 5 arquivo1.txt
tail -n 5 arquivo1.txt
```

Por que o tail -f é importante em cibersegurança?

Monitoramento de Logs:

Em cibersegurança, logs são uma das principais fontes de informação para identificar atividades suspeitas, ataques em andamento e falhas no sistema.

O tail -f permite que você veja imediatamente novas entradas nos logs, o que é essencial para respostas rápidas.

Exemplo:

```
tail -f /var/log/auth.log
```

Editando Arquivos

Editar arquivos diretamente no terminal é possível usando editores de texto como nano, vi ou vim.

O Nano é um editor de texto fácil de usar, ideal para iniciantes ou usuários que precisam de um editor leve e direto ao ponto. Ele é instalado por padrão em muitas distribuições Linux e oferece uma interface amigável com atalhos exibidos na parte inferior da tela.

O Nano apresenta comandos básicos na parte inferior, como salvar (^O) e sair (^X), onde ^ indica a tecla Ctrl.

Atalhos Úteis:

- Salvar o arquivo: Ctrl+O
- Sair do editor: Ctrl+X
- Cortar texto: Ctrl+K
- Colar texto: Ctrl+U

Basta abrir um arquivo ou criar um novo digitando nano nome_do_arquivo.

```
nano meu_script.sh
```

Abre ou cria o arquivo meu_script.sh para edição. Você pode escrever seu script, salvar com Ctrl+O e sair com Ctrl+X.

O Vi é um editor avançado, projetado para usuários que precisam de funcionalidades robustas e personalização. Ele é uma escolha popular entre administradores de sistemas e profissionais experientes em Linux.

- Modo Comando é usado para navegação, pesquisa e execução de comandos.
- Modo Inserção é usado para editar o texto.
- Para alternar para o Modo Inserção, pressione i. Para voltar ao Modo Comando, pressione Esc.

Atalhos Poderosos:

- Salvar o arquivo: :w
- Sair do editor: :q
- Salvar e sair: :wq
- Sair sem salvar: :q!

Navegação Avançada:

- Mover para o início da linha: 0
- Mover para o final da linha: $
- Pesquisar uma palavra: /palavra

```
vi meu_script.sh
```

O Vim (abreviação de Vi Improved) é uma versão aprimorada do editor Vi, amplamente utilizado em sistemas baseados em Unix/Linux. Ele mantém a filosofia do Vi, com múltiplos modos de operação, mas adiciona recursos modernos.

Diferente do Vi, o Vim suporta realce de sintaxe, funcionalidade de desfazer múltiplos níveis e plugins, o que o torna uma ferramenta poderosa.

```
vim meu_script.sh
```

Se o realce não estiver ativado, você pode ativá-lo com:

```
:syntax on
```

Curisidade: VIM ou VI se tornaram memes pela dificuldade de uso para usuários comuns. Um exemplo é essa postagem de 2014.

"*Estou usando o Vim há cerca de 2 anos agora, principalmente porque não consigo descobrir como sair dele*."

Movendo e Copiando Arquivos

Para mover ou copiar arquivos, usamos os comandos mv e cp, respectivamente.

Mover um arquivo para outro diretório:
`mv arquivo1.txt /caminho/para/destino/`

Copiar um arquivo para outro diretório:
`cp arquivo1.txt /caminho/para/destino/`

Excluindo Arquivos

Para excluir arquivos no Linux, usamos o comando rm:
Excluir um único arquivo:
`rm arquivo1.txt`

Excluir vários arquivos ao mesmo tempo:
`rm arquivo1.txt arquivo2.txt`

Excluir um diretório e seu conteúdo:
`rm -r /caminho/do/diretorio`

Dicas e Boas Práticas
- Sempre tenha cuidado ao usar o comando rm, especialmente com a opção -r, pois ele pode excluir grandes quantidades de dados rapidamente.
- Use head ou tail para visualizar o conteúdo de arquivos grandes sem precisar abrir o arquivo inteiro.
- Mantenha seus arquivos organizados em diretórios para facilitar o gerenciamento.

🌟 **Curiosidade:** O comando rm também virou "meme" como essa camiseta que diz:

"*Dica do Linux para remover o pacote de idioma francês: sudo rm -fr /**"

Essa é uma piada comum em comunidades Linux, já que o comando sudo rm -fr /* apaga tudo no sistema sem pedir confirmação . Não deve ser executado, pois causará danos irreversíveis ao sistema operacional!
Entenda o comando:

Sudo dá privilégios administrativos ao comando. Sem isso, muitos arquivos protegidos pelo sistema não poderiam ser apagados.
rm é o comando para remover arquivos ou diretórios.

-f (force) força a exclusão de arquivos sem solicitar confirmação, mesmo se forem protegidos ou somente leitura.
-r (recursive) remove diretórios e todos os seus conteúdos, incluindo subdiretórios e arquivos.
/* especifica que a ação deve ser aplicada a todos os arquivos e diretórios na raiz do sistema (/). Isso inclui arquivos de configuração do sistema, diretórios de usuários e até o próprio kernel.

Apagou vai pra lixeira?

Não, **os arquivos apagados com o comando rm no Linux não vão para a lixeira.** Diferente do que acontece em sistemas como Windows ou macOS, o comando rm remove os arquivos de forma permanente e direta do sistema, sem colocá-los em um local de recuperação (como uma lixeira).

Permissões de Arquivos

No Linux, as permissões de arquivos desempenham um papel crucial na proteção de dados e no controle de acesso. Compreender e gerenciar essas permissões é essencial para garantir que apenas usuários autorizados possam acessar ou modificar arquivos sensíveis. Este capítulo explora como visualizar, interpretar e modificar permissões de arquivos e diretórios, com exemplos práticos para o contexto de cibersegurança.

Como as Permissões Funcionam no Linux

Cada arquivo ou diretório no Linux possui permissões que determinam quem pode ler, escrever ou executar aquele recurso. Essas permissões são atribuídas a três categorias de usuários:

Usuário (Owner) - O proprietário do arquivo.
Grupo - Um grupo de usuários que compartilham permissões específicas.
Outros (Others) - Todos os outros usuários do sistema.

Visualizando Permissões
Você pode visualizar as permissões de arquivos e diretórios usando o comando **ls** com a opção **-l**:

```
ls -l arquivo.txt
```

Exemplo de saída:
```
-rw-r--r-- 1 usuario grupo 1234 out  2 10:00
```

`arquivo.txt`
Interpretação:
--rw-r--r--: São as permissões.

- **r (read):** Permissão de leitura.
- **w (write):** Permissão de escrita (gravação).
- **x (execute):** Permissão de execução.

- O primeiro caractere (-) indica o tipo de arquivo
(-para arquivo normal, d para diretório).
- O restante indica as permissões para o **usuário**, **grupo** e **outros**, respectivamente.

Neste exemplo o usuário tem permissão de leitura e escrita. O grupo de leitura, e os outros de leitura também.

Modificando Permissões com chmod

O comando **chmod** é usado para alterar permissões de arquivos e diretórios. Ele aceita dois formatos principais: simbólico e numérico.

Formato Simbólico
Use símbolos para adicionar, remover ou definir permissões específicas:

+ Adiciona uma permissão.
- Remove uma permissão.
= Define permissões específicas.

Exemplo:

```
chmod u+rwx arquivo.txt
chmod g-w arquivo.txt
chmod o=r arquivo.txt
```

u+rwx: Adiciona leitura, escrita e execução para o usuário.
g-w: Remove a permissão de escrita para o grupo.
o=r: Define apenas permissão de leitura para outros.

⚡ Nota: Quando você **define permissões**, as permissões existentes são sobrescritas pelas permissões que você especificar. Tudo o que não for explicitamente mencionado será removido e
quando você **adiciona permissões**, as permissões especificadas são somadas às permissões já existentes do arquivo ou diretório. As permissões que já estavam definidas permanecem inalteradas.

Formato Numérico
As permissões podem ser representadas numericamente. Cada permissão tem um valor:
- 4: Leitura (r).
- 2: Escrita (w).
- 1: Execução (x).

Para definir permissões, some os valores e atribua a cada categoria (usuário, grupo, outros):

```
chmod 754 arquivo.txt
```

- 7 (usuário): Leitura, escrita e execução (4+2+1).
- 5 (grupo): Leitura e execução (4+1).
- 4 (outros): Apenas leitura.

Alterando Proprietário e Grupo

Use o comando chown para alterar o proprietário ou grupo de um arquivo:
`sudo chown usuario arquivo.txt`

- Alterar o grupo:
`sudo chown :grupo arquivo.txt`

- Alterar ambos:
`sudo chown usuario:grupo arquivo.txt`

Permissões Especiais

Além das permissões padrão, o Linux suporta permissões especiais para casos específicos:
- **SUID** (Set User ID): Permite que um arquivo seja executado com os privilégios de seu proprietário.
- **SGID** (Set Group ID): Permite que um arquivo ou diretório seja executado com os privilégios de seu grupo.
- **Sticky Bit:** Restringe a exclusão de arquivos em diretórios compartilhados apenas ao proprietário.

Exemplo de ativação:
- Configurar SUID:
`chmod u+s arquivo`

- Configurar Sticky Bit:
`chmod +t diretorio`

Dicas e Boas Práticas
- Sempre use permissões mínimas necessárias para reduzir a exposição a riscos.
- Evite usar permissões 777 (total acesso) a menos que absolutamente necessário.
- Use o comando ls -ld para verificar permissões de diretórios.
- Revise frequentemente as permissões de arquivos críticos para evitar configurações inseguras.

Variáveis

As variáveis no Bash são fundamentais para armazenar e manipular dados durante a execução de scripts. Elas permitem personalizar comandos, simplificar operações e criar scripts mais dinâmicos e reutilizáveis. Neste capítulo, exploraremos como declarar, modificar e usar variáveis no Bash, com exemplos práticos voltados para cibersegurança.

O que são Variáveis no Bash?

Variáveis no Bash são usadas para armazenar valores, como strings, números ou comandos. Elas são identificadas por um nome e podem ser utilizadas em scripts para tornar os comandos mais dinâmicos e adaptáveis.

Como Declarar Variáveis

Para declarar uma variável no Bash, basta atribuir um valor a um nome de variável. Não é necessário declarar o tipo de dado, pois o Bash trata todas as variáveis como strings por padrão.

Sintaxe Básica:

```
NOME_VARIAVEL=valor
```

- Não use espaços ao redor do sinal de igual (“=“).
- Use nomes descritivos para facilitar a leitura do código.

Exemplo:

```
MEU_NOME='Daniel'
MINHA_IDADE=30
echo "Meu nome é $MEU_NOME e minha idade é $MINHA_IDADE."
```

- A saída será: "**`Meu nome é Daniel e minha idade é 30`**."

Tipos de Variáveis

Embora o Bash trate todas as variáveis como strings por padrão, você pode manipular diferentes tipos de dados dependendo do contexto.

- **Letras**: Somente alfabeto **(A-Z, a-z)** pode iniciar o nome de uma variável.
- **Números**: Podem ser usados, mas **não como o primeiro caractere**.
- **Underscore (_)**: Aceito em qualquer parte do nome
- **Case Sensitive**: Variáveis diferenciam maiúsculas de minúsculas

Não aceita ser variável no Linux

- **Caracteres especiais**: Símbolos como @, $, &, %, -, etc., não podem ser usados
- **Espaços**: Não são permitidos no nome
- **Iniciar com números**: O nome da variável não pode começar com números.
- **Palavras reservadas**: Certas palavras têm significados específicos no shell e não podem ser usadas.

  Exemplos inválidos: if, then, else, while, do

Variáveis de Ambiente

Variáveis de ambiente são variáveis especiais usadas pelo sistema e seus programas. Elas podem ser acessadas por qualquer processo e são úteis para configurar o comportamento do sistema.

Exemplo de Variáveis de Ambiente Comuns:

- “PATH”: Contém os diretórios onde o sistema busca executáveis.
- “HOME”: Diretório inicial do usuário.
- “USER”: Nome do usuário logado.

Exibindo Variáveis de Ambiente:
Use o comando “printenv” ou “env” para listar todas as variáveis de ambiente:

```
printenv
```

Variáveis pré-definidas

$SHELL: Indica o shell padrão que está sendo usado, por exemplo, /bin/bash.

$PWD: Exibe o diretório de trabalho atual.

$OLDPWD: Mostra o diretório de trabalho anterior, útil para navegar entre diretórios.

$LANG: Define a configuração de linguagem e localidade do sistema, como pt_BR.UTF-8.

$UID: Exibe o ID do usuário que está executando o shell.

$USER: Exibe o nome do usuário atual

$HOSTNAME: Mostra o nome do host da máquina.

$RANDOM: Retorna um número aleatório cada vez que é referenciado.

$SECONDS: Conta o número de segundos desde que o shell foi iniciado.

$PS1: Define o prompt padrão do shell, que você pode personalizar.

$?: Contém o status de saída do último comando executado. 0 significa sucesso, e qualquer outro valor indica falha.

$#: Mostra o número de argumentos passados para o script.

$@: Lista todos os argumentos fornecidos ao script como strings separadas.

$0: Exibe o nome do script que está sendo executado.

$$: Retorna o ID do processo (PID) do shell atual.

Argumentos nos Scripts Bash

Os argumentos são informações passadas para um script no momento de sua execução, permitindo que ele seja mais dinâmico e adaptável a diferentes situações. No Bash, os argumentos são acessados usando variáveis posicionais como

$1, $2, $3, etc., representando os valores fornecidos na linha de comando.

Quando você executa um script Bash e fornece valores após o nome do script, esses valores se tornam argumentos que podem ser usados no código.

Exemplo Básico de Argumentos
Crie um script chamado exemplo.sh:

```
#!/bin/bash
echo "Olá, $1! Você passou o argumento: $2"
```

Ao executar esse arquivo você pode adicionar as variáveis como se fosse um parâmetro do comando
Exemplo

```
./exemplo.sh Daniel "Bash é incrível"
```

Onde:
$1: Representa o primeiro argumento (Daniel).
$2: Representa o segundo argumento (Bash é incrível).

O resultado será:
Olá, Daniel! Você passou o argumento: Bash é incrível

Variáveis Especiais de Argumentos

Variável	Descrição
$0	Nome do script em execução.
$1, $2, ...	Representam os argumentos passados para o script.

$#	Número total de argumentos fornecidos ao script.
$*	Todos os argumentos como uma única string.
$@	Todos os argumentos, mas tratados como strings separadas.
"$@"	Preserva a separação entre os argumentos, mesmo quando contêm espaços.

SPOILER ALERT

No exemplo a seguir vamos encontrar as declarações if, then que será apresentada mais adiante, mas é importante usá-las agora pois demonstra a importância dessas variáveis especiais e o uso de argumentos.

```
!/bin/bash

# Verifica se o número de argumentos é igual a zero
if [ $# -eq 0 ]; then
    echo "Erro: Nenhum argumento foi fornecido."
    echo "Uso: $0 <arg1> [arg2] [arg3] ..."
    exit 1
fi
```

Este script valida o número de argumentos fornecidos ($#) na sua execução e se for igual a zero (-eq 0) exibe uma mensagem sobre o erro e como usar o script exibindo o seu nome ($0).

Alias

Um **alias** é uma forma de criar um comando personalizado no terminal. Ele substitui um comando ou uma sequência de comandos por um nome mais curto e fácil de lembrar.

Para ver todos os alias atualmente configurados no terminal basta digitar **`alias`**

Sintaxe básica:

```
alias nome_do_alias='comando ou sequência de comandos'
```

Para remover

```
unalias nome_do_alias
```

Criar um Alias Básico

```
alias ll='ls -la'
```

Agora, ao digitar ll, o comando ls -la será executado.

Alias com Comandos Longos

```
alias atualizar='sudo apt update && sudo apt upgrade -y'
```

Você também pode utilizar alguns comandos de aplicativos junto com o alias, como por exemplo 9º netstat para verificar conexões ativas:

```
alias conexoes='netstat -ant'
```

🌟 **Curiosidade**: O conceito de alias foi introduzido no Shell C (csh) como uma maneira de reduzir o número de comandos que precisavam ser digitados repetidamente. Ele se tornou tão popular que foi adotado em outros Shells como o Bash.

Alias Temporários vs Permanentes

Alias criados diretamente no terminal são válidos apenas durante a sessão atual.

Ao fechar o terminal, eles são perdidos.

Para criar alias que persistem entre sessões, adicione-os ao arquivo de configuração do Bash, como ~/.bashrc ou ~/.bash_aliases.

Abra o arquivo ~/.bashrc

```
nano ~/.bashrc
```

Adicione o alias

```
alias atualizar='sudo apt update && sudo apt upgrade -y'
```

Os alias são uma poderosa ferramenta para tornar o ambiente Bash mais eficiente e personalizado.

Parsing no Shell

Parsing é o processo de analisar e extrair informações específicas de um texto ou arquivo de dados. No Shell, parsing é uma prática essencial para trabalhar com a saída de comandos, arquivos de texto, logs e outros dados. Ferramentas como cut, awk, sed, e redirecionadores são amplamente utilizadas para esse propósito.

O Que é Parsing?

Parsing no Shell consiste em:

- Dividir o texto em partes menores com base em delimitadores (como vírgulas, espaços ou tabulações).
- Filtrar ou extrair partes específicas do conteúdo.
- Transformar os dados para análise ou uso posterior.

Por que Parsing é Importante?

- Automatizar tarefas repetitivas que envolvem manipulação de dados.
- Processar logs de sistema e gerar relatórios.
- Preparar dados para outros scripts ou sistemas.

🌼 **Curiosidade**: Parsing no Shell tem raízes na necessidade de manipular dados textuais em sistemas UNIX nos anos 70. Ferramentas como awk e sed foram criadas para processar grandes volumes de texto de maneira eficiente, uma necessidade na época dos mainframes.

Comandos especiais

Neste capítulo iremos explorar alguns comandos que merecem um espaço especial pois serão de extrema utilidade para operações diária e principalmente em cibersegurança.

SORT

O comando sort organiza linhas de texto em ordem alfabética, numérica ou personalizada. Ele é extremamente útil para estruturar dados antes de analisá-los, permitindo identificar padrões, duplicatas e outros insights.

Como funciona?

- Ordena linhas de um arquivo ou entrada padrão.
- Pode trabalhar com critérios como ordem alfabética, numérica ou por colunas específicas.

Sintaxe:

```
sort [opções] [arquivo]
```

Principais Opções

-n: Ordena numericamente.
-r: Inverte a ordem, criando uma ordem decrescente.
-k <n>: Ordena com base na coluna especificada.
-u: Remove duplicatas automaticamente (não tão eficiente quanto uniq que veremos a seguir).

Um exemplo com resultado

```
cat logs.txt
```

192.168.1.1 - Successful Login
192.168.1.2 - Failed Login
192.168.1.1 - Successful Login
192.168.1.3 - Failed Login

sort logs.txt
192.168.1.1 - Successful Login
192.168.1.1 - Successful Login
192.168.1.2 - Failed Login
192.168.1.3 - Failed Login

UNIQ

O comando uniq filtra e exibe apenas as linhas únicas em uma entrada, eliminando duplicatas consecutivas. Ele funciona melhor quando os dados estão ordenados, tornando sua combinação com sort muito comum.

Como funciona?

- Remove duplicatas consecutivas.
- Pode contar ocorrências de duplicatas.

Sintaxe:
uniq [opções] [arquivo]

Principais Opções

-c: Exibe a contagem de cada linha única.
-d: Exibe apenas as linhas duplicadas.
-u: Exibe apenas as linhas não duplicadas.

Um exemplo com resultado

```
sort logs.txt | uniq -c
   2 192.168.1.1 - Successful Login
   1 192.168.1.2 - Failed Login
   1 192.168.1.3 - Failed Login
```

Como UNIQ e SORT funcionam juntos?

Para que uniq funcione corretamente, os dados precisam estar ordenados. Portanto, o uso combinado de sort e uniq é uma prática comum. O sort organiza os dados, e o uniq remove ou analisa as duplicatas.

Como no exemplo, o sort organiza os IPs, e o uniq -c mostra a frequência de cada entrada.

Na cibersegurança, a análise de logs, listas de endereços IP e outros dados estruturados é uma tarefa crítica. Usar sort e uniq facilita:

- **Detecção de IPs suspeitos**: Identificar os IPs que aparecem mais frequentemente em logs.
- **Filtragem de dados**: Eliminar duplicatas em listas de hashes, usuários ou arquivos.
- **Análise de padrões**: Descobrir quais eventos ocorrem mais frequentemente e priorizar ações de mitigação.

Os comandos sort e uniq são ferramentas simples, mas poderosas, especialmente em cibersegurança. Com eles, é possível processar grandes volumes de dados de forma eficiente, extrair insights importantes e identificar padrões ou anomalias que podem indicar uma ameaça.

Grep

O comando grep (Global Regular Expression Print) é uma das ferramentas mais poderosas e amplamente utilizadas no Bash para buscar e filtrar padrões de texto em arquivos ou na entrada padrão. Sua capacidade de usar expressões regulares torna-o essencial para análises detalhadas e para encontrar rapidamente informações relevantes em grandes volumes de dados.

Principais Características

- Busca eficiente de strings ou padrões em arquivos.
- Suporte a expressões regulares básicas e avançadas.
- Pode filtrar dados diretamente da entrada padrão ou arquivos.

Sintaxe:
grep [opções] "padrão" [arquivo]

Opções Comuns

-i: Ignora diferenças entre maiúsculas e minúsculas.
-v: Inverte a correspondência, exibindo linhas que **não** correspondem ao padrão.
-n: Mostra o número da linha junto com o resultado.
-c: Conta o número de ocorrências do padrão.
-r ou -R: Busca recursivamente em diretórios.
-l: Lista apenas os nomes dos arquivos que contêm o padrão.
--color: Realça o texto encontrado no terminal.

Exemplos Práticos

Busca Simples
`grep "error" logs.txt`
Exibe todas as linhas do arquivo logs.txt que contêm a palavra error.

Busca Recursiva em Diretórios
`grep -r "unauthorized access" /var/logs/`
Busca o padrão unauthorized access em todos os arquivos dentro do diretório /var/logs.

Ignorar Maiúsculas e Minúsculas
`grep -i "login" logs.txt`
Encontra login, LOGIN ou qualquer variação de maiúsculas/minúsculas.

Contar Ocorrências
`grep -c "failed" logs.txt`
Conta quantas vezes o padrão failed aparece no arquivo.

Exibir o Número das Linhas
`grep -n "attack" logs.txt`
Mostra as linhas contendo attack, juntamente com seus números.

🌟 **Curiosidade**: O comando grep, criado por **Ken Thompson** em 1973, foi inicialmente parte do editor de texto ed. Ele era tão útil para busca e filtragem que se tornou um comando independente no UNIX.

Expressões Regulares no grep

As expressões regulares tornam o grep uma ferramenta ainda mais poderosa.

Começando com uma Palavra
`grep "^user" logs.txt`
Encontra linhas que começam com user

Terminando com uma Palavra
`grep "error$" logs.txt`
Encontra linhas que terminam com error.

Padrões com Caracteres Especiais
`grep "[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}" data.txt`
Busca por números no formato 123-45-6789 (comum para números de série ou CPF).

Ou (Alternativas)
`grep -E "error|failed" logs.txt`
Encontra linhas que contenham error **ou** failed.

Casos de Uso em Cibersegurança

Identificar IPs Suspeitos em Logs, o comando a seguir é capaz de extrair endereços IP, contar as ocorrências e classificar por frequência.

```
grep -oE "([0-9]{1,3}\.){3}[0-9]{1,3}" logs.txt |
sort | uniq -c | sort -nr
```

O comando grep é uma ferramenta essencial para quem trabalha com dados em texto, especialmente em cibersegurança. Sua versatilidade e capacidade de usar

expressões regulares tornam-no ideal para análise de logs, extração de informações e detecção de padrões maliciosos.

AWK

O awk é uma linguagem de programação leve e poderosa para manipulação de texto e extração de dados em arquivos ou streams de entrada. Ele é amplamente usado em ambientes Unix/Linux para tarefas que envolvem análise de grandes volumes de texto, especialmente em cibersegurança, onde a eficiência na manipulação de logs e dados estruturados é essencial.

O awk processa texto linha por linha, dividindo as linhas em campos (por padrão, separados por espaços ou tabulação) e permitindo realizar ações baseadas nesses dados. É ideal para filtrar, formatar e transformar texto, sendo uma ferramenta indispensável para administradores de sistemas e profissionais de cibersegurança.

Em resumo ele permite:

- Filtrar linhas com base em padrões.
- Selecionar colunas específicas de texto.
- Realizar cálculos.
- Reformatar e manipular dados.

Sintaxe:

```
awk 'padrão {ação}' arquivo
```

- padrão: Define quais linhas serão processadas.
- ação: Especifica o que fazer com as linhas correspondentes ao padrão.

- arquivo: Arquivo ou entrada padrão (stdin) a ser processado

Exemplo:
awk '{print $1}' logs.txt
Exibe apenas o primeiro campo de cada linha do arquivo logs.txt.

Conceitos Importantes

Estrutura de Campos

Por padrão, o awk divide cada linha em campos utilizando o espaço como delimitador. Os campos podem ser referenciados por $n, onde n é o número do campo:

- $1: Primeiro campo.
- $2: Segundo campo.
- $0: A linha inteira.

Operadores e Condições

- /padrão/: Aplica ações somente às linhas que correspondem ao padrão.
- &&: E lógico.
- ||: Ou lógico.
- !: Negação.

Funções do AWK

Filtrar Dados
`awk '/error/ {print $0}' logs.txt`
Exibe apenas as linhas que contêm a palavra error.

Selecionar Campos Específicos
`awk '{print $1, $3}' logs.txt`
Exibe o primeiro e o terceiro campos de cada linha.

Realizar Cálculos
`awk '{total += $3} END {print total}' dados.txt`

Soma o terceiro campo de todas as linhas e exibe o total.

O awk é uma ferramenta extremamente poderosa para manipulação de texto, especialmente em cibersegurança, onde a análise eficiente de logs e dados é crucial. Sua capacidade de filtrar, transformar e organizar informações torna-o indispensável para analistas e administradores que precisam lidar com grandes volumes de dados.

CUT

O comando cut no Shell é usado para extrair partes específicas de um texto, como colunas, caracteres ou campos delimitados. É uma ferramenta simples e eficiente para manipular linhas de texto em arquivos ou saídas de comandos.

Como Funciona o cut?

O cut divide uma linha de texto com base em delimitadores (como vírgulas ou tabulações) ou em posições específicas, extraindo apenas os segmentos desejados. Ele trabalha linha por linha, tornando-o ideal para processar arquivos tabulares ou relatórios gerados por outros comandos.

Principais Opções do cut

-d (Delimitador) Define o delimitador que separa os campos na linha. O padrão é tabulação (\t), mas você pode especificar outro delimitador, como vírgula ou espaço.

Exemplo:

```
echo "nome,email,idade" | cut -d ',' -f 2
```

Saida:

```
email
```

-f (Campos)
Especifica quais campos (colunas) devem ser extraídos com base no delimitador.
Exemplo:
```
echo "nome,email,idade" | cut -d ',' -f 1,3
```
Saída:
nome,idade

-c (Caracteres)
Extrai caracteres específicos ou intervalos de caracteres de cada linha.
Exemplo:
```
echo "123456789" | cut -c 1-4
```
Saída:
1234
--complement
Extrai tudo exceto os campos ou caracteres especificados.
Exemplo:
```
echo "nome,email,idade" | cut -d ',' --complement -f 2
```
Saída:
nome,idade

O cut é uma ferramenta poderosa para tarefas rápidas de parsing e manipulação de texto. Apesar de suas limitações, é ideal para situações em que você precisa de simplicidade e eficiência ao extrair dados de arquivos ou saídas de comandos. Ao combiná-lo com outras ferramentas, como grep ou awk, ele pode ser parte de soluções mais robustas para processamento de dados no Shell.

SED

O comando sed é uma poderosa ferramenta de manipulação de texto usada para processar e transformar arquivos ou fluxos de entrada em um ambiente Unix/Linux. Ele permite buscar, substituir, adicionar ou excluir linhas de texto, sendo amplamente utilizado em automação de tarefas e scripts, especialmente na área de cibersegurança, para manipular rapidamente grandes volumes de dados.

O sed processa texto linha por linha e aplica comandos ou expressões para realizar transformações. Ele funciona em fluxo, permitindo modificar o texto diretamente sem alterar os arquivos originais (a menos que seja explicitamente solicitado).

Sintaxe
sed [opções] 'comando' [arquivo]

Principais Comandos do SED

s/padrão/substituição/: Substitui o padrão pelo texto especificado.
d: Exclui linhas.
p: Imprime linhas (usado com a opção **-n** para controlar a saída).
a texto: Adiciona texto após uma linha.
i texto: Insere texto antes de uma linha.
c texto: Substitui uma linha inteira por outro texto.

Exemplos

Substituir Texto

```
sed 's/erro/sucesso/' arquivo.txt
```

Substitui a primeira ocorrência de erro por sucesso em cada linha.
Para substituir todas as ocorrências em uma linha, use a flag g:

```
sed 's/erro/sucesso/g' arquivo.txt
```

Substituir em um Arquivo e Salvar

```
sed -i 's/falha/sucesso/g' arquivo.txt
```

Substitui todas as ocorrências de falha por sucesso e salva as alterações diretamente no arquivo (-i modifica o arquivo in-place).

Excluir Linhas Específicas

```
sed '2d' arquivo.txt
```

Exclui a segunda linha.

```
sed '/temp/d' arquivo.txt
```

Exclui todas as linhas que contêm a palavra temp.

Imprimir Apenas Linhas Correspondentes

```
sed -n '/alert/p' arquivo.txt
```

Imprime apenas as linhas que contêm *alert*

Inserir ou Adicionar Linhas

```
sed '3i\Nova linha antes da terceira linha'
arquivo.txt
```

Insere uma nova linha antes da terceira linha.

```
sed '3a\Nova linha após a terceira linha'
arquivo.txt
```

Adiciona uma nova linha após a terceira linha.

Alterar Linhas Inteiras
`sed '3c\Linha completamente alterada' arquivo.txt`
Substitui a terceira linha por Linha completamente alterada.

Usos Avançados

Trabalhar com Intervalos
`sed '2,4d' arquivo.txt`
Exclui linhas do intervalo 2 a 4.

`sed '1,5s/erro/correção/g' arquivo.txt`
Substitui erro por correção apenas nas linhas de 1 a 5.

Substituir com Expressões Regulares
`sed 's/[0-9]\{4\}-[0-9]\{2\}-[0-9]\{2\}/DATA_ENCONTRADA/g' arquivo.txt`
Substitui datas no formato YYYY-MM-DD por DATA_ENCONTRADA.

Adicionar Prefixo ou Sufixo
`sed 's/^/Prefixo: /' arquivo.txt`
Adiciona Prefixo: no início de cada linha.

`sed 's/$/ :Sufixo/' arquivo.txt`
Adiciona :Sufixo ao final de cada linha

Na área de cibersegurança, o sed é uma ferramenta fundamental para:

- **Sanitização de Dados**: Limpar logs ou arquivos antes de compartilhá-los, removendo informações sensíveis.
- **Filtragem de Dados**: Excluir ou modificar informações irrelevantes em logs ou resultados de varreduras.
- **Automatização**: Substituir padrões em vários arquivos ou gerar relatórios rapidamente.
- **Criação de Scripts**: Manipular arquivos de configuração ou logs em pipelines automatizados.

Casos de Uso em Cibersegurança

```
sed 's/192\.168\.0\.[0-9]\{1,3\}/[REMOVIDO]/g' logs.txt
```

Substitui todos os IPs da rede 192.168.0.x por [REMOVIDO].

Automatizar Atualização de Configurações

```
sed -i 's/port=8080/port=9090/' config.cfg
```

Atualiza o valor da porta de 8080 para 9090 em um arquivo de configuração.

IF, THEN, ELSE

O uso de condicionais no Shell Script é fundamental para criar scripts dinâmicos e interativos que tomam decisões com base em determinadas condições. O if, then, else, e elif permitem executar diferentes blocos de código dependendo do resultado de testes lógicos ou comparações.

estrutura básica de um bloco condicional no Shell Script é a seguinte:

```
if [ condição ]
then
    # Código a ser executado se a condição for
verdadeira
else
    # Código a ser executado se a condição for
falsa
fi
```

Quando há várias condições possíveis, o elif (else if) pode ser usado:

```
if [ condição1 ]
then
    # Código para condição1
elif [ condição2 ]
then
    # Código para condição2
else
    # Código caso nenhuma condição seja
verdadeira
Fi
```

No Shell Script, a estrutura condicional IF, THEN, e ELSE é essencial para criar lógica e tomar decisões com base em

condições dinâmicas. Os operadores desempenham um papel fundamental nesse contexto, pois são eles que definem as condições que serão avaliadas dentro dessa estrutura.
Eles ajudam a comparar valores, verificar propriedades de arquivos, validar entradas de texto ou combinar múltiplas condições.

Em cibersegurança, a combinação de operadores e condicionais é usada para validar permissões de arquivos, identificar vulnerabilidades ou realizar ações somente em condições seguras

Exemplo:

```
if [ -f "/etc/passwd" ] && [ ! -w "/etc/passwd" ]
then
    echo "O arquivo é seguro."
else
    echo "Alerta: Verifique as permissões do
arquivo!"
fi
```

Vamos analisar esse código:

Este script verifica se o arquivo /etc/passwd é seguro com base em duas condições:

A primeira condição [-f "/etc/passwd"]:

- O operador -f verifica se /etc/passwd é um arquivo regular (não um diretório ou outro tipo de arquivo especial).
- Se for verdadeiro, significa que o arquivo existe e é regular.

A segunda condição [! -w "/etc/passwd"]:

- O operador -w verifica se o arquivo /etc/passwd tem permissões de escrita (ou seja, se pode ser modificado).
- O operador lógico ! inverte o resultado, indicando que o teste será verdadeiro se o arquivo não puder ser escrito.

O operador lógico &&:

- Combina as duas condições. Ambas precisam ser verdadeiras para que o bloco then seja executado.

Bloco then:

- Se ambas as condições forem verdadeiras, o script exibe a mensagem: "O arquivo é seguro."

Bloco else:

- Se alguma das condições falhar, o script exibe a mensagem: "Alerta: Verifique as permissões do arquivo!"

Estamos vendo que é importante iniciarmos um capitulo sobre operadores.

Operadores

No contexto do Shell Script, **operadores** são símbolos ou palavras-chave usados para realizar operações em dados ou expressões. Eles são fundamentais para implementar a lógica de programação e permitem realizar tarefas como comparações, cálculos, validações e manipulações de texto ou arquivos.

Os operadores são a base para tomar decisões em scripts, avaliar condições e automatizar processos de forma eficiente. Eles podem ser usados para:

- Comparar números ou strings.
- Verificar características de arquivos ou diretórios.
- Combinar múltiplas condições lógicas.
- Realizar ações condicionais e iterativas em automações.

Categorias de Operadores

Os operadores no Shell Script são classificados em diferentes categorias, cada uma com funções específicas:

- **Numéricos**: Usados para comparar números inteiros.
- **Strings**: Utilizados para avaliar e comparar variáveis de texto.
- **Lógicos**: Permitem combinar ou inverter condições.
- **Arquivos**: Avaliam propriedades de arquivos e diretórios, como existência, permissões ou tamanho.

Operadores Numéricos

Os operadores numéricos são usados para realizar comparações entre valores inteiros. Eles são fundamentais para verificar condições baseadas em números, como contadores ou resultados de cálculos. Em scripts, é comum utilizá-los para determinar se um valor é maior, menor ou igual a outro. Por exemplo, podem ser usados para verificar se um limite foi atingido ou se um contador atingiu o valor esperado.

Numéricos

Operador	Descrição
-eq	Igual a
-ne	Diferente de

-lt	Menor que
-le	Menor ou igual a
-gt	Maior que
-ge	Maior ou igual a

Strings

Operador	Descrição
=	Igual a
!=	Diferente de
-z	Verdadeiro se a string estiver vazia
-n	Verdadeiro se a string não estiver vazia

Operadores Lógicos

Os operadores lógicos são usados para combinar ou inverter condições em um script. Eles permitem executar ações com base em múltiplas condições simultâneas (AND, OR) ou negar uma condição específica (NOT). Esses operadores são essenciais para criar lógica condicional mais avançada, como validar múltiplas verificações em uma única expressão.

Lógicos

Operador	Descrição
&&	E lógico (AND)
||	Ou lógico (OR)
!	Não lógico (negação)

Operadores de Arquivos

Os operadores de arquivos permitem verificar propriedades específicas de arquivos e diretórios, como existência, permissões ou tamanho. Eles são amplamente utilizados em scripts de automação e cibersegurança para garantir que um arquivo está disponível, possui permissões corretas ou não está vazio. Esses operadores são indispensáveis para validar recursos antes de realizar operações que possam impactar o sistema ou o ambiente de trabalho.

Arquivos

Operador	Descrição
-e	Verdadeiro se o arquivo existir
-f	Verdadeiro se for um arquivo regular
-d	Verdadeiro se for um diretório
-s	Verdadeiro se o arquivo não estiver vazio
-r	Verdadeiro se o arquivo for legível
-w	Verdadeiro se o arquivo for gravável
-x	Verdadeiro se o arquivo for executável

Arrays no Shell Script

No Shell Script, arrays são estruturas que permitem armazenar múltiplos valores em uma única variável. Cada valor é acessado por um índice numérico, começando do índice 0. Eles são amplamente usados para organizar dados, iterar sobre listas de valores e simplificar scripts que precisam lidar com grandes quantidades de informações.

Existem duas formas principais de declarar arrays no Shell Script

Declaração Simples
```
array=("valor1" "valor2" "valor3")
```

- Cada elemento é separado por um espaço e colocado entre parênteses.
- Os valores podem ser strings, números ou uma combinação de ambos.

Ou adicionando Elementos Individualmente
```
array[0]="valor1"
array[1]="valor2"
array[2]="valor3"
```

Exibir elementos de um Array

```
echo ${array[0]}  # Exibe o primeiro elemento
echo ${array[2]}  # Exibe o terceiro elemento
```

Para exibir todos os elementos de um array, use a notação @ ou *
```
echo ${array[@]}  # Exibe todos os elementos do array
echo ${array[*]}  # Também exibe todos os elementos
```

Para obter o número de elementos, use a notação ${#array[@]}:
```
echo ${#array[@]}  # Exibe o número de elementos
```

Modificar Arrays

Substituir um Valor
```
array[1]="novo_valor"
```

Remover Elementos
```
unset array[1]  # Remove o segundo elemento
```

Exemplo de script com array

```
#!/bin/bash
# Script para verificar a conectividade com uma
lista de IPs usando ping.

# Declaração do array com os IPs a serem
verificados
ips=("192.168.0.1" "192.168.0.2" "8.8.8.8")

# Iteração sobre os IPs do array
for ip in "${ips[@]}"
do
    echo "Verificando conexão com $ip..."
    # Executa o comando ping uma vez para
verificar a conectividade
    if ping -c 1 $ip &> /dev/null
    then
        echo "Conexão com $ip bem-sucedida."
    else
        echo "Falha ao conectar com $ip."
```

```
    fi
done

# Fim do script
```

Os arrays no Shell Script são ferramentas poderosas para manipular e organizar dados, aumentando a eficiência de scripts e permitindo criar soluções dinâmicas. Com o uso de arrays e loops, é possível realizar operações complexas de forma clara e organizada, tornando-os indispensáveis para automação e cibersegurança.

Looping (FOR, WHILE, UNTIL)

Em Shell Script, **loops** são usados para executar repetidamente um bloco de código enquanto uma condição for verdadeira (ou até que ela se torne verdadeira). Eles são fundamentais para automatizar tarefas repetitivas, tornando scripts mais eficientes e flexíveis.
Existem três tipos principais de loops no Shell Script:

- **FOR**: Itera sobre uma lista de valores.
- **WHILE**: Continua a execução enquanto uma condição for verdadeira.
- **UNTIL**: Continua a execução até que uma condição se torne verdadeira.

FOR

O loop FOR é usado para iterar sobre uma lista de itens ou um intervalo numérico. Ele é ideal quando você conhece previamente os elementos ou o número de iterações.

Sintaxe:

```
for var in lista
do
    # Bloco de código
done
```

Exemplo Prático
Iterando sobre uma lista de nomes:

```
#!/bin/bash
nomes=("Ana" "Lucas" "Rafael")

for nome in "${nomes[@]}"
do
    echo "Olá, $nome!"
done
```

- Útil para trabalhar com arrays ou intervalos numéricos.
- Simples de implementar para tarefas com iterações conhecidas.

WHILE

O loop WHILE executa o bloco de código enquanto a condição especificada for verdadeira. É mais adequado para situações em que a quantidade de iterações não é conhecida previamente.

Sintaxe:

```
while [ condição ]
do
    # Bloco de código
done
```

Exemplo Prático
Contando de 1 a 5:

```
#!/bin/bash
contador=1

while [ $contador -le 5 ]
do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

- Adequado para cenários em que a condição depende de variáveis dinâmicas.
- Uso em loops infinitos combinado com condições, pode ser usado para serviços contínuos

UNTIL

O loop UNTIL é semelhante ao WHILE, mas executa o bloco de código **até que** a condição especificada se torne verdadeira. Ele é útil para situações onde o estado inicial já é falso.

Sintaxe

```
until [ condição ]
do
    # Bloco de código
done
```

Exemplo Prático
Esperando até que um valor atinja 5:

```
#!/bin/bash
contador=1

until [ $contador -gt 5 ]
do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

Comparação entre FOR, WHILE e UNTIL

Tipo de Loop	Quando Usar?
FOR	Quando você sabe exatamente quantas iterações deseja realizar.
WHILE	Quando a condição de execução é dinâmica e pode mudar durante a execução.
UNTIL	Quando a lógica do problema se encaixa melhor no conceito de "até que algo".

Casos de Uso em Cibersegurança

FOR: Verificar uma lista de IPs ou arquivos.

```
#!/bin/bash
# Script para verificar conectividade com uma
lista de IPs

ips=("192.168.1.1" "10.0.0.1" "8.8.8.8")

for ip in "${ips[@]}"
do
    echo "Verificando conectividade com $ip..."
    if ping -c 1 $ip &> /dev/null
    then
        echo "Conexão com $ip bem-sucedida."
    else
        echo "Falha ao conectar com $ip."
    fi
done
```

WHILE: Monitorar Logs Continuamente

```
#!/bin/bash
# Script para monitorar logs em tempo real

echo "Monitorando o arquivo /var/log/auth.log.
Pressione Ctrl+C para sair."
while true
do
```

```
    tail -f /var/log/auth.log
done
```

UNTIL: Esperar um Serviço Estar Ativo

```
#!/bin/bash
# Script para aguardar até que o serviço SSH
esteja ativo

echo "Verificando se o serviço SSH está ativo..."
until systemctl is-active --quiet sshd
do
    echo "O serviço SSH ainda não está ativo.
Tentando novamente em 5 segundos..."
    sleep 5
done

echo "O serviço SSH está ativo! Continuando a
execução."
```

Os loops FOR, WHILE e UNTIL são ferramentas essenciais em Shell Script para criar scripts dinâmicos e flexíveis. Cada tipo de loop atende a necessidades específicas, desde a manipulação de listas até a execução baseada em condições.

Funções

As funções no Bash permitem encapsular um bloco de código que pode ser reutilizado em diferentes partes do script.

Essa prática melhora a organização, reduz a redundância e facilita a manutenção do código. Usar funções é especialmente útil em scripts longos ou que realizam tarefas repetitivas.

O que é uma Função no Bash?

Uma função é um bloco de código identificado por um nome que pode ser chamado em qualquer parte do script. Ao invés de duplicar o código, você define a lógica uma vez dentro da função e a chama sempre que necessário.

Como Declarar uma Função?

Existem duas formas principais de declarar uma função no Bash, com parênteses:

```
nome_da_funcao() {
    # Bloco de código da função
}
```

Sintaxe com function (Opcional):

```
function nome_da_funcao {
    # Bloco de código da função
}
```

Para executar uma função, basta chamá-la pelo nome

```
nome_da_funcao
```

Funções são um dos pilares para criar scripts organizados, reutilizáveis e fáceis de manter. Em Bash, uma

função encapsula um bloco de código que pode ser chamado várias vezes ao longo do script, reduzindo redundâncias e facilitando a leitura. Vamos explorar as vantagens das funções e entender como elas podem transformar um simples script em uma solução mais eficiente e profissional.

Imagine que você precise verificar a conectividade com diversos IPs dentro de um script. Sem o uso de funções, seria necessário duplicar o código de verificação para cada IP. Isso tornaria o script maior, mais complexo e suscetível a erros. Com uma função, a lógica de verificação é definida uma vez e pode ser chamada quantas vezes for necessário:
Vamos ver um exemplo prático de um script com funções:

```
#!/bin/bash
# Função para verificar conectividade com um IP

verificar_conexao() {
    local ip=$1
    if ping -c 1 $ip &> /dev/null
    then
        echo "Conexão com $ip bem-sucedida."
    else
        echo "Falha ao conectar com $ip."
    fi
}

# Lista de IPs
ips=("192.168.1.1" "8.8.8.8" "10.0.0.1")

# Loop para chamar a função
for ip in "${ips[@]}"
do
    verificar_conexao $ip
```

```
done
```

Variável Local vs Global

Quando usamos “local”, a variável criada é restrita ao escopo da função em que foi declarada.

Sem o uso de local, as variáveis declaradas dentro de uma função no Bash podem se tornar globais (ou seja, podem ser acessadas fora da função), o que pode causar problemas em scripts maiores.

```
#!/bin/bash

# Função que usa 'local' para restringir o escopo
minha_funcao() {
    local var_local="Eu sou local"
    var_global="Eu sou global"
    echo "$var_local"
    echo "$var_global"
}

minha_funcao
```

Vantagens de usar funções

Além da reutilização do código as funções ajudam a dividir o script em **blocos lógicos**, tornando-o mais legível e fácil de entender. Um script que utiliza funções corretamente

permite que até mesmo alguém que não o escreveu consiga compreender suas partes principais rapidamente.

As funções são ferramentas essenciais para criar scripts Bash eficientes, organizados e profissionais. Elas promovem **reutilização**, **modularidade**, **facilidade de manutenção** e **clareza**, além de simplificar a escalabilidade e a adaptação dos scripts.

Executar Scripts em Background com &

No Bash, executar scripts ou comandos em background significa iniciar uma tarefa sem bloquear o terminal, permitindo que você continue trabalhando enquanto o script é executado. Isso é especialmente útil para tarefas que consomem tempo, como backups, análises de logs ou varreduras de rede.

Para executar um script ou comando em segundo plano, basta adicionar o caractere & ao final da linha de execução.

Como Funciona o &

Quando você adiciona & no final de um comando ou script, o Bash inicia a tarefa em **background**. Isso significa que o controle do terminal retorna imediatamente ao usuário, enquanto a tarefa continua sendo executada em segundo plano.

Exemplo:

```
./backup.sh &
```

Quando você executa o script com &, o terminal exibirá algo como:

```
[1] 12345
```

Onde:

- **[1]**: É o número do job em segundo plano.
- **12345**: É o número do processo (PID) associado à tarefa em execução

Enquanto o script estiver "rodando" você pode consultá-lo com o comando Jobs

```
jobs
```

Saída típica:

```
[1]+  Running                 ./backup.sh &
```

Se você quiser trazer o processo de volta para o primeiro plano, use o comando:

```
fg %1
```

Onde **%1** refere-se ao número do job mostrado pelo comando Jobs

Parando um Processo em Background

Se necessário, você pode interromper um processo em background usando o comando kill seguido pelo número do PID

Encontre o PID do processo:

```
jobs -l
```

Interrompa o processo com o Kill
`kill PID`

Melhores Práticas

Quando um script é executado em background, sua saída padrão pode aparecer no terminal, misturando-se com outras atividades. Para evitar isso, redirecione a saída para um arquivo:

./backup.sh > output.log 2>&1 &

> output.log: Redireciona a saída padrão (stdout) para o arquivo output.log.
2>&1: Redireciona a saída de erro (stderr) para o mesmo arquivo.

Executar scripts em background usando & é uma técnica poderosa para gerenciar tarefas de longa duração no Bash. Ela melhora a eficiência e permite multitarefa, especialmente em cenários onde o tempo é crítico.

Agendando Tarefas com cron

O cron é uma ferramenta poderosa e amplamente utilizada no Linux para agendar e automatizar a execução de tarefas. Ele permite que você execute comandos ou scripts em intervalos regulares, como minutos, horas, dias ou meses, sem

intervenção manual. Isso é especialmente útil para backups, limpezas, monitoramento de sistemas e outras tarefas recorrentes.

O Que é o cron?

Cron - Um daemon que roda em segundo plano e executa tarefas programadas.
Ele verifica os arquivos de configuração chamados **crontabs** para determinar quais tarefas executar e quando.
Crontab é o arquivo ou comando utilizado para gerenciar as tarefas agendadas pelo cron.

Comandos Básicos do cron

Abrir o Editor de Agendamento:
`crontab -e`
Permite adicionar, editar ou excluir tarefas agendadas.

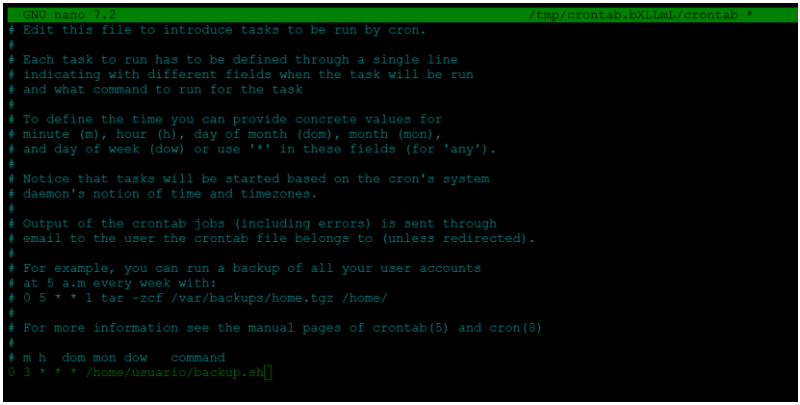
Listar Tarefas Agendadas:
`crontab -l`
Exibe todas as tarefas do usuário atual.

Remover Todas as Tarefas:
`crontab -r`
Apaga todas as tarefas do usuário atual.

Agendar uma tarefa com o crontab

Abra o editor de agendamento com o comando

```
crontab -e
```



Cada linha no arquivo crontab representa uma tarefa e segue esta estrutura:

```
* * * * * comando_a_ser_executado
```

Campos:

1º: Minuto (0-59)
2º: Hora (0-23)
3º: Dia do mês (1-31)
4º: Mês (1-12)
5º: Dia da semana (0-7, onde 0 e 7 são domingo)

```
*  *  *  *  * comando_a_ser_executado
-  -  -  -  -
|  |  |  |  |
|  |  |  |  +---- Dia da semana (0 - 7) (Domingo pode ser 0 ou 7)
|  |  |  +------- Mês (1 - 12)
|  |  +---------- Dia do mês (1 - 31)
|  +------------- Hora (0 - 23)
+---------------- Minuto (0 - 59)
```

`0 3 * * * /home/usuario/backup.sh`
Neste exemplo, será executado o script de backup todos os dias às 3:00 da manhã.

Limpar Arquivos Temporários Semanalmente:
`0 0 * * 0 rm -rf /tmp/*`
Remove todos os arquivos no diretório /tmp todos os domingos à meia-noite.

Operadores Especiais no crontab

*** (Qualquer Valor):**
O comando será executado em todos os valores possíveis para aquele campo.
Exemplo: * * * * * executa o comando a cada minuto.

***/n (Intervalo):**
Define uma repetição regular dentro de um intervalo.
Exemplo: */10 * * * * executa a tarefa a cada 10 minutos.

- (Intervalo de Valores):
Especifica um intervalo contínuo.

Exemplo: 0-5 * * * * executa a tarefa nos primeiros 5 minutos de cada hora.

, (Valores Específicos):
Define valores específicos separados por vírgulas.
Exemplo: 0,15,30,45 * * * * executa a tarefa a cada 15 minutos.

Cada usuário tem seu próprio arquivo crontab, geralmente localizado em /var/spool/cron/crontabs/

Exemplos Comuns de Tarefas Agendadas

Executar Scripts de Qualquer Lugar

Uma das grandes vantagens de trabalhar com scripts no Linux é a possibilidade de executá-los de qualquer lugar no sistema, sem precisar navegar até o diretório onde estão armazenados. Para isso, você precisa entender e configurar corretamente a variável de ambiente PATH.

O que é o PATH?

A variável PATH é uma lista de diretórios onde o sistema operacional procura por programas ou scripts executáveis.

Quando você digita um comando no terminal, o sistema verifica os diretórios especificados no PATH para encontrar o executável correspondente.

Exemplo: Se você digita ls, o sistema verifica os diretórios no PATH e encontra o programa ls, normalmente localizado em /bin/.

Por que Configurar o PATH para Scripts?

Ao configurar o PATH para incluir o diretório onde seus scripts estão localizados, você pode executar seus scripts de qualquer lugar, sem precisar fornecer o caminho completo ou estar no mesmo diretório.

Adicionar Scripts ao PATH?

Existem duas abordagens principais para configurar o PATH

Colocar seus Scripts em um Diretório que Já Está no PATH

Os diretórios padrão incluídos no PATH geralmente incluem:

- /usr/local/bin
- /usr/bin
- /bin
- /home/usuario/bin (em alguns sistemas)

Ao mover qualquer script seu para um desses diretório você pode executar o script de qualquer lugardigitando o nome dele somente

```
meu_script.sh
```

Adicionar o Diretório dos Seus Scripts ao PATH

Adicionar o seu **diretório personalizado** ao PATH é vantajoso porque centraliza os executáveis e scripts que você usa regularmente em um único local, facilitando a organização e o acesso. Isso permite que qualquer script ou programa dentro desse diretório seja executado diretamente no terminal, sem a necessidade de caminhos completos. Além disso, manter um diretório específico no PATH evita a modificação de pastas críticas do sistema, reduzindo o risco de conflitos ou problemas de segurança ao gerenciar seus próprios arquivos executáveis.

Para isso, edite o arquivo ~/.bashrc (ou ~/.bash_profile, dependendo da distribuição)

```
nano ~/.bashrc
```

Adicione o Diretório ao PATH

```
export PATH=$PATH:/home/usuario/meus_scripts
```

Para verificar quais diretórios estão atualmente no PATH, use o comando:

```
echo $PATH
```

Boas Práticas ao Configurar o PATH

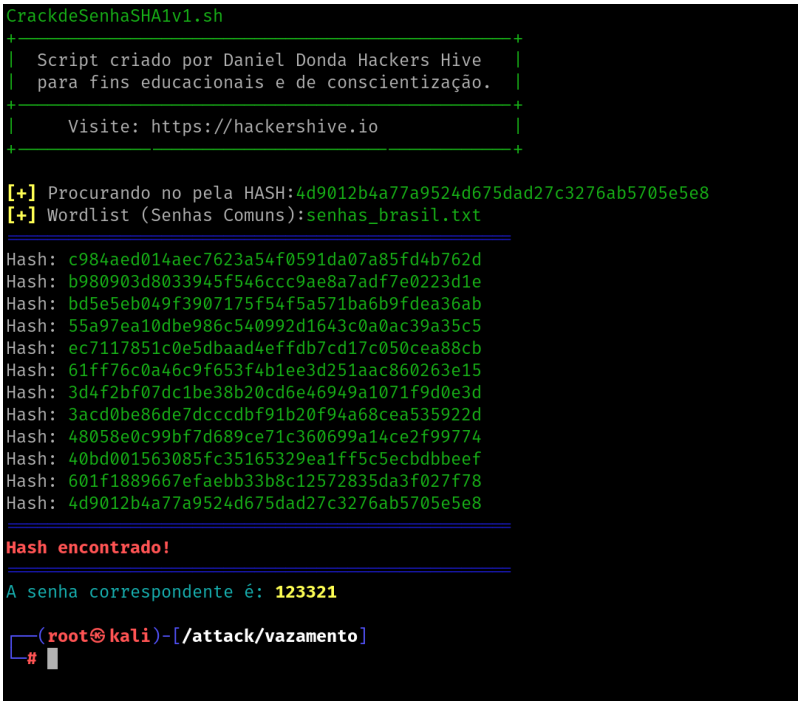
- Mantenha Diretórios Organizados
- Evite Adicionar Diretórios Sensíveis
- Certifique-se de que os Scripts são Executáveis (chmod +x)

Configurar o PATH corretamente é um passo essencial para otimizar seu ambiente de trabalho e aumentar a eficiência no uso de scripts. Seja colocando seus scripts em um diretório já incluído no PATH, seja adicionando diretórios personalizados, essa prática torna seus scripts mais acessíveis e simplifica seu fluxo de trabalho. Lembre-se de seguir as boas práticas para manter seu sistema seguro e organizado.

Personalize Seus Scripts com Cores no Bash

Adicionar cores aos seus scripts Bash é uma maneira eficiente de destacar informações importantes, tornando-os mais intuitivos e agradáveis de usar. Cores ajudam a diferenciar mensagens de erro, sucessos ou alertas, facilitando a identificação rápida do status de uma operação.

No Bash, as cores são controladas usando **sequências de escape ANSI**, que configuram as propriedades de cor e estilo do texto exibido no terminal.



Introdução às Sequências de Escape ANSI

As sequências de escape ANSI são códigos usados para personalizar cores, estilos e posicionamento do texto no terminal.

Exemplo:

```
echo -e "\033[1;31mTexto em vermelho e negrito\033[0m"
```

\033: Indica o início da sequência de escape.

[... m: Configura a formatação do texto (cor, estilo, fundo).
0, 1, 4: Representam os estilos (normal, negrito, sublinhado).
30-37: Representam as cores do texto.
40-47: Representam as cores do fundo.

No Bash, você pode usar tanto \033 quanto \e para iniciar uma sequência de escape ANSI. Ambos funcionam de forma idêntica, mas \033 é uma representação mais explícita do caractere de escape (o código ASCII 27), enquanto \e é uma forma abreviada suportada pelo Bash. Para evitar confusões, é recomendado adotar um padrão consistente no seu script.

```
\e[<código de estilo>;<código de cor>m
```

Exemplo:

```
echo -e "\e[31mTexto Vermelho\e[0m"
```

Neste livro, usamos \033 para manter a clareza e a compatibilidade universal.

Abaixo estão tabelas com os códigos mais utilizados para estilos, cores de texto e cores de fundo

Cores Regulares

Código	Descrição
\033[0;30m	Preto
\033[0;31m	Vermelho
\033[0;32m	Verde
\033[0;33m	Amarelo
\033[0;34m	Azul
\033[0;35m	Roxo
\033[0;36m	Ciano
\033[0;37m	Branco

Negrito

Código	Descrição
\033[1;30m	Preto em negrito
\033[1;31m	Vermelho em negrito
\033[1;32m	Verde em negrito
\033[1;33m	Amarelo em negrito
\033[1;34m	Azul em negrito
\033[1;35m	Roxo em negrito
\033[1;36m	Ciano em negrito
\033[1;37m	Branco em negrito

Sublinhado

Código	Descrição
\033[4;30m	Preto sublinhado
\033[4;31m	Vermelho sublinhado
\033[4;32m	Verde sublinhado
\033[4;33m	Amarelo sublinhado
\033[4;34m	Azul sublinhado
\033[4;35m	Roxo sublinhado
\033[4;36m	Ciano sublinhado
\033[4;37m	Branco sublinhado

Alta Intensidade

Código	Descrição
\033[0;90m	Preto (alta intensidade)
\033[0;91m	Vermelho (alta intensidade)
\033[0;92m	Verde (alta intensidade)
\033[0;93m	Amarelo (alta intensidade)
\033[0;94m	Azul (alta intensidade)
\033[0;95m	Roxo (alta intensidade)
\033[0;96m	Ciano (alta intensidade)

\033[0;97m	Branco (alta intensidade)

https://github.com/HackersHiveClub/Cybersecurity-Bash-Essential

Usar variáveis para definir as cores no início do script é uma prática recomendada, pois melhora a organização e facilita a manutenção do código. Ao invés de repetir os códigos ANSI diretamente em diferentes partes do script, você pode atribuí-los a variáveis descritivas, como Red ou Green, tornando o código mais legível e intuitivo. Além disso, se for necessário alterar uma cor ou estilo, basta ajustar a variável correspondente uma única vez, propagando a mudança automaticamente para todo o script. Essa abordagem não só economiza tempo, como também reduz erros, especialmente em scripts longos ou complexos.

Veja no exemplo a seguir:

Aplicação Prática em Scripts

```
#!/bin/bash

# Definindo cores
Red='\033[0;31m'
Green='\033[0;32m'
Yellow='\033[0;33m'
Reset='\033[0m'

# Lógica de exemplo
if [ -d "/home" ]; then
    echo -e "${Green}[SUCESSO] O diretório /home
existe.${Reset}"
```

```
else
    echo -e "${Red}[ERRO] O diretório /home não
foi encontrado.${Reset}"
fi
echo -e "${Yellow}[AVISO] Certifique-se de
verificar permissões.${Reset}"
```

Tabela de Comandos

Comando	Descrição
AWK	Manipula e processa texto baseado em padrões e colunas.
Alias	Cria atalhos para comandos ou sequências de comandos.
CUT	Extrai partes específicas de texto com base em delimitadores ou posições.
ELSE	Parte de uma estrutura condicional, executada se a condição for falsa.
FOR	Executa um conjunto de comandos em loop para cada item de uma lista.
Grep	Procura por padrões em arquivos ou entradas de texto.
IF	Avalia uma condição e executa comandos se ela for verdadeira.
SED	Edita texto de forma não interativa com padrões definidos.
SORT	Ordena linhas de texto em arquivos ou entradas.
Shell	Interface para interação com o sistema operacional, geralmente via terminal.
THEN	Parte de uma estrutura condicional, executada se a condição for verdadeira.
UNIQ	Remove linhas duplicadas consecutivas em um arquivo ou entrada.
UNTIL	Executa um loop até que uma condição seja verdadeira.
WHILE	Executa um loop enquanto uma condição for verdadeira.

apt	Gerencia pacotes de software em distribuições baseadas no Debian.
array	Armazena múltiplos valores em uma única variável no Shell.
bash	Um shell popular usado em sistemas baseados em UNIX.
cat	Exibe o conteúdo de arquivos no terminal.
chage	Gerencia a validade e a expiração de senhas de usuários.
chmod	Altera permissões de arquivos e diretórios.
cp	Copia arquivos ou diretórios.
crontab	Agende tarefas para execução automática.
date	Exibe ou define a data e hora do sistema.
echo	Imprime mensagens ou variáveis no terminal.
export	Define ou exporta variáveis de ambiente.
fg	Restaura um processo em segundo plano para o primeiro plano.
gpasswd	Gerencia senhas de grupos no sistema.
head	Exibe as primeiras linhas de um arquivo.
jobs	Exibe uma lista de processos em execução ou suspensos.
kill	Envia sinais para encerrar processos.
less	Exibe arquivos página por página.
ls	Lista arquivos e diretórios.
mv	Move ou renomeia arquivos e diretórios.
nano	Editor de texto simples para o terminal.
passwd	Altera senhas de usuários.
ping	Testa conectividade de rede com um host.
printenv	Exibe variáveis de ambiente.
rm	Remove arquivos ou diretórios.
sleep	Pausa a execução por um período de tempo definido.

stderr	Fluxo de saída para mensagens de erro.
stdin	Fluxo de entrada padrão para comandos ou programas.
stdout	Fluxo de saída padrão para comandos ou programas.
su	Permite trocar para outro usuário.
sudo	Executa comandos com privilégios elevados.
tail	Exibe as últimas linhas de um arquivo.
touch	Cria arquivos vazios ou altera a data de modificação de arquivos existentes.
useradd	Adiciona novos usuários ao sistema.
userdel	Remove usuários do sistema.
usermod	Modifica contas de usuário.
vi	Editor de texto avançado no terminal.
vim	Versão aprimorada do editor de texto vi.

Indice dos comandos

FIM