# POWŁOKA I GUI – JAK PRACOWAĆ BEZPIECZNIE

## MIT BEZPIECZEŃSTWA

Na Linuksie nic mi nie grozi, to moja maszyna i wiem, co robię. Narażeni na atak są użytkownicy [wstaw nazwę dowolnego OS, o którym lub którego użytkownikach masz niskie mniemanie], nie ja.

Pierwsze doświadczenia z Linuksem zdobywałem jakieś ćwierć wieku temu na domowym pececie, korzystając z dystrybucji na płytach CD dołączonych do gazet. O samym systemie nauczyłem się wtedy niewiele, za to straciłem część danych przez nieumiejętne partycjonowanie dysku przy próbach instalacji*. W kolejnych latach (i firmach) zebrałem sporą listę „sukcesów": a to wpisałem `reboot` w niewłaściwym oknie, a to zamazałem sektor startowy dysku z produkcyjnymi danymi**, a to nie przeczytałem dokładnie listy katalogów przy wywoływaniu `rm -rf`. Rachunek prawdopodobieństwa zazwyczaj jednak był dla mnie łaskawy. Na przykład w tym ostatnim przypadku system kopii zapasowych wdrożyłem dwa dni wcześniej, nie zdążyłem go jedynie przetestować… Zadziałał.

Kto po kilkuletniej karierze w IT nie ma podobnie wstydliwych wspomnień, niech pierwszy rzuci klawiaturą. Większość linuksowych narzędzi napisano z założeniem, że osoba przy klawiaturze rozumie, co robi, czytaj: często nie wybaczają błędów (zwłaszcza gdy popełniasz je z konta `root`). Wprawdzie nic nie zastąpi uważności i skupienia przy pracy oraz znajomości narzędzi, ale eliminacja takich błędów nie sprowadza się wyłącznie do „ogarnij się, patrz, co wpisujesz, przeczytaj manual". Odpowiednia konfiguracja środowiska i stosowanie dobrych praktyk nie tylko zwiększają komfort pracy, ale i utrudniają popełnianie pomyłek. Także tych katastrofalnych w skutkach.

W tym rozdziale skupię się znacznie bardziej na codziennej pracy w środowisku użytkownika oraz na tworzeniu skryptów niż na czynnościach administracyjnych i konfiguracyjnych obejmujących cały system. Będzie on miał w dużej mierze formę poradnikową, dlatego nie umieściłem na jego końcu checklisty – cały jest bowiem zbiorem rad i wskazówek.

Zakładam, że znasz podstawy składni skryptów powłoki, a co najmniej potrafisz napisać i uruchomić `helloworld.sh`. Jeśli potrzebujesz pomocy w tym zakresie, możesz zajrzeć do *Advanced Bash-Scripting Guide*[1] (ma on już wprawdzie swoje lata, ale podstawy nieszczególnie zmieniły się od czasu jego stworzenia). Zachęcam też do intensywnego korzystania ze wspomnianych w rozdziale stron podręcznika systemowego.

---

* Partycja i system plików na partycji to dwie osobne struktury, a ich rozmiary zmienia się innymi narzędziami…

** Na szczęście nośnik był spartycjonowany przy użyciu GPT, więc pierwszy sektor (512 bajtów) nie zawierał ani tablicy partycji, ani kodu bootloadera (nie był to zresztą dysk startowy tego systemu).

# BHPS: BEZPIECZEŃSTWO I HIGIENA POWŁOKI ORAZ SKRYPTÓW

> *90% of the code I write is only to catch the 5% of things that go wrong.*
> (znalezione gdzieś na Stack Overflow)

Powłoka to nie tylko środowisko pracy służące do uruchamiania programów. To także specyficzny, zorientowany tekstowo język programowania, który ma swoją gramatykę oraz zbiór reguł i założeń (zwykle nie całkiem jasnych dla większości ludzi, którzy z niej korzystają). Nie mam zamiaru zmieniać tego rozdziału w kurs dla deweloperów skryptów: zakładam, że znasz absolutne podstawy ich tworzenia. Zamiast tego **przyjrzę się obszarom, których niezrozumienie może prowadzić do błędów w działaniu skryptów** (a każdy taki błąd, w zależności od kontekstu, może być błędem bezpieczeństwa). Pokażę też na przykładach, jak dostosowanie konfiguracji środowiska powłoki ułatwia życie i zmniejsza szansę popełnienia kosztownych pomyłek.

> Jeśli nie zaznaczono inaczej, polecenia i skrypty w tej części rozdziału odnoszą się do powłoki bash. Jeśli korzystasz z innej, nie wszystkie podane sposoby i rozwiązania zadziałają. Wyjątkowo też w przykładach wiersz polecenia (*prompt*) pokazuję w skrótowej formie (`$`) zamiast w typowej, rozwiniętej (`uzytkownik@host:katalog$`).

## W jakim języku rozmawiamy?

Spieszę z wyjaśnieniem, zanim w związku z uwagą powyżej uniksowi puryści zażądają mojej głowy*: **Bash nie jest formalnym ani nieformalnym standardem** tego, jak ma wyglądać i działać powłoka. Bywa często tak postrzegany, bo jest domyślnie obecny w wielu popularnych dystrybucjach serwerowych i desktopowych. Niemniej jednak przy pisaniu skryptu, który ma działać gdziekolwiek indziej niż na maszynie, na której go stworzono, warto pamiętać, że `/bin/bash` i `/bin/sh` to niekoniecznie to samo.

Pod tą drugą nazwą np. w Debianie kryje się link do powłoki `dash` – bardzo niewygodnej w pracy interaktywnej, ale za to szybszej od `basha` (co ważne przy wykonywaniu skryptów) i zorientowanej na zgodność z normą POSIX (o której za moment). Oprócz modnych i pełnych wizualnych „wodotrysków” `fish` i `zsh` w wielu systemach nadal są dostępne `ksh` czy `tcsh`, sięgające korzeniami jeszcze klasycznych Uniksów z lat osiemdziesiątych XX wieku. Cokolwiek jest dostarczane jako „standardowe” `/bin/sh` w danej dystrybucji – nie zmieniaj tego bez głębszej analizy. Nie ma zresztą takiej potrzeby: **w każdym skrypcie powłoki może (i stanowczo powinna!) znajdować się preambuła** określająca, za pomocą czego ma on być uruchamiany (tzw. *shebang*).

* Albo, co gorsza, wzorem anglojęzycznych kolegów każą mi oddać legitymację (*hand in your geek card!*)...

Piszesz i testujesz w Bashu, wykorzystując specyficzne dla niego funkcje? Dodaj na początku:

```
#!/bin/bash
```

zamiast typowego:

```
#!/bin/sh
```

Warto jednak pamiętać, że `bash` **może wcale nie być zainstalowany** na docelowym systemie, podobnie jak narzędzia GNU coreutils (`ls`, `cat`, `rm` itp.). Raczej nie spotkamy go w systemach wbudowanych (*embedded*) czy specjalizowanych, lekkich dystrybucjach. Takie środowiska potrzebują powłok stworzonych z myślą o minimalnym rozmiarze i użyciu zasobów, np. ash czy BusyBox. Ta druga jest ciekawa: zawiera bardzo wiele wbudowanych poleceń analogicznych do tych z pakietu coreutils. Są one wprawdzie bardzo uproszczone i pozbawione wielu funkcji, ale dzięki temu w jednym pliku wykonywalnym może się mieścić całe środowisko pracy: powłoka, programy do wyświetlania plików, kopiowania, usuwania itd.*. Charakterystyczne w takich systemach jest to, że np. `/bin/ls` czy `/bin/cat` to linki symboliczne do binarki `busybox`.

**Istnieje formalny, komercyjny standard zwany POSIX**[2], który powstał jako próba wypracowania jednolitego API programistycznego i środowiska użytkownika (powłoki) dla różnych systemów z rodziny Unix. Jeśli dany skrypt jest napisany z założeniem zgodności z POSIX (nie „pod Basha" czy dowolną inną powłokę), powinien bez problemu zadziałać na każdej z nich. Na Linuksie można w wielu powłokach i narzędziach włączyć za pomocą odpowiednich przełączników bądź zmiennych środowiskowych tryb zwiększonej kompatybilności z POSIX (choć nie znaczy to, że funkcje wykraczające poza standard przestaną działać całkowicie).

Aby zapewnić, że skrypt napisany na jednej maszynie będzie zgodny z powłoką (i narzędziami) na innej, można:

- trzymać się zasady „uruchamiaj na tym, na czym było pisane" (np. Bash na Ubuntu);
- jeśli powyższe jest niemożliwe, jasno udokumentować wymagania środowiska („Linux, Bash w wersji X, coreutils w wersji Y"), w razie potrzeby włącznie z weryfikowaniem ich w samym skrypcie;
- trzymać się standardu POSIX – to najlepsza opcja, jeśli potrzebna jest przenośność między różnymi systemami operacyjnymi (Linux, macOS, AIX, Solaris…).

Dostęp do dokumentacji standardu nie jest na szczęście płatny (jak to czasami bywa z dokumentami normatywnymi takich organizacji jak ISO czy IEEE), ale wymaga rejestracji na opengroup.org[3].

* Taka konstrukcja powoduje również, że BusyBox jest mniej wrażliwy na podmianę systemowych binariów lub bibliotek, przez co może być bardzo przydatny w analizie zainfekowanego czy przejętego systemu.

## Co ja właściwie uruchamiam?

Po podaniu polecenia powłoka musi ustalić, co tak naprawdę ma do uruchomienia. Komenda może być np. aliasem, funkcją bądź poleceniem, które jest wbudowane w nią samą (*builtin*). Jeśli tak nie jest, a polecenie nie rozpoczyna się od ścieżki do pliku wykonywalnego (bezwzględnej, czyli zaczynającej się od katalogu głównego, bądź względnej, choćby składającej się z kropki i ukośnika), wówczas następuje poszukiwanie pliku o wskazanej nazwie w katalogach, których lista zawarta jest w zmiennej o nazwie PATH. Jej zawartość może być nieco zaskakująca – oto przykłady z dwóch popularnych dystrybucji:

```
# Debian 12 – root
root@debian12:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

# Debian 12 – zwykły użytkownik
stefan@debian12:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games

# RHEL 9 (i odpowiedniki) – root
[root@alma9 ~]# echo $PATH
/root/.local/bin:/root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin

# RHEL 9 (i odpowiedniki) – zwykły użytkownik
[stefan@alma9 ~]$ echo $PATH
/home/stefan/.local/bin:/home/stefan/bin:/usr/local/bin:/usr/bin:/usr/
local/sbin:/usr/sbin
```

Innymi słowy, jeśli chcesz przejąć konto użytkownika, ale możesz jedynie skopiować niepostrzeżenie jeden plik do jego katalogu domowego, to np. na Almie/RHEL wgranie złośliwej wersji `ls` do `.local/bin` daje sporą szansę powodzenia. Oczywiście wszędzie można też próbować podłożyć spreparowane `.profile` czy `.bashrc`…

Działanie logiki ustalania, „co uruchomić” można obejrzeć w `bashu` za pomocą polecenia `type -a`:

```
$ type -a ls
ls is aliased to `ls --color=auto'
ls is /usr/bin/ls
ls is /bin/ls

$ type -a echo
echo is a shell builtin
echo is /usr/bin/echo
echo is /bin/echo
```

*Shell builtin*, polecenie wbudowane, ma wyższy priorytet. Może działać nieco inaczej niż jego wersja z pliku binarnego (np. obsługiwać inne przełączniki). Chcąc więc dowiedzieć się, co dokładnie potrafi `echo` uruchamiane poleceniem `echo` (nie `/usr/bin/echo`!), muszę zajrzeć do `man bash-builtins`, nie do `man echo`.

W skryptach, których docelowe środowisko działania nie jest znane i przewidywalne (np. zmienna PATH może być zmodyfikowana), można stosować ustawione „na sztywno" ścieżki do poleceń:

```
#!/bin/sh
GREP="/usr/bin/grep"
$GREP Awaria /opt/aplikacja/logs/*.log > /home/stefan/awarie.log
```

Nigdy nie dodawaj katalogu bieżącego (`.`) do zmiennej PATH.

Jeśli to zrobisz, do zaatakowania Cię wystarczy podłożenie złośliwej binarki o nazwie np. `ls` w ogólnodostępnym katalogu (np. `/tmp`) i fałszywa wiadomość w rodzaju: „pomocy, ja nie rozumiem, co ten serwer wyrabia, nic nie działa, wejdź szybko do tempa i zobacz, co tam jest".

## Używaj cudzysłowów, waliduj wejście

Spacja, tabulator, dolar ($), nawias okrągły (()), kwadratowy ([]) i klamrowy ({}), tylda (~), backtick (`), backslash (\), znak nowej linii... To wszystko może być częścią nazwy pliku. Praca z takimi „potworkami" nastręcza sporo problemów i lepiej takich nazw unikać. Dla pojedynczych plików szybkim obejściem problemu: „chcę to podejrzeć/skasować/zmienić nazwę, ale nie wiem, jak wpisać oryginalną", jest użycie pojedynczego cudzysłowu lub dopełnianie tabulatorem (jeśli pierwszy znak nazwy jest nietypowy, spróbuj poprzedzić go backslashem) lub... Midnight Commander. To ostatnie jest może „mało eleganckie" i „lamerskie", ale jeśli jest pod ręką, a Ty masz do naprawienia awarię, zrób to, co rozumiesz i co zadziała. Rozwiązań „eleganckich" i „pro" warto się oczywiście uczyć, ale zrób to jednak już po opanowaniu sytuacji.

Pisząc skrypty, trzymaj się zasady: **zmienna musi być ujęta w cudzysłowy, gdy się do niej odwołujesz**:

```
$ NAZWA='ala ma kota'  # nie odwołuję się do niczego, więc mogę użyć
                       # pojedynczych cudzysłowów

$ mkdir $NAZWA         # powstały TRZY katalogi: 'ala', 'ma' oraz 'kota'
$ ls                   # ls sortuje wyniki alfabetycznie
ala   kota   ma

$ mkdir "$NAZWA"       # to zadziała zgodnie z oczekiwaniami
$ ls -1                # wyświetl jeden plik per linia, nie w kolumnach
ala
ala ma kota
kota
ma
```

Jest to szczególnie ważne, jeśli zmienna ma stanowić nazwę pliku bądź zawiera dane z zewnątrz (podane w argumencie, pobrane przez `read`, odczytane z systemu plików itp.). Pomoże to np. na pliki ze spacjami w nazwie. Nie chroni jednak przed ewidentnie złośliwym wejściem, np. przed podłożeniem parametru w rodzaju:

```
alamakota`echo AAAAkluczSSHwlamywacza >> /root/.ssh/authorized_keys`.txt
```

Wobec takich prób ataku skrypt może bronić się przez bardzo dokładne sprawdzanie, czy wejście nie zawiera „podejrzanych" znaków, jak: dolar (`$`), backtick (`` ` ``) lub dowolne nawiasy `()[]{}<>`, i przerwanie działania w razie ich wykrycia. Można też zastosować radykalne (ale najbezpieczniejsze) podejście: przyjąć określony zestaw znaków jako dozwolony w danej sytuacji i pozbyć się wszystkich innych. Poniżej przykład funkcji, którą można wykorzystać np. do oczyszczania niezaufanych ciągów wejściowych:

```
#!/bin/bash

# funkcja zamieni na znak podkreślenia każdy znak prócz liter, cyfr, kropek
# i ukośników. Dodatkowo tr zamieni znaki końca wiersza (sed też mógłby to
# robić, ale wyrażenie byłoby jeszcze mniej przejrzyste)

sanitize()
{
    sed 's/[^[:alnum:]/.]/_/g'  | tr '\n' '_'
}

# przykładowa zmienna zawierająca wiele niebezpiecznych znaków, m.in.
# tabulator, spacje, koniec wiersza, nawiasy, dolar, cudzysłów

ZLE='/home/stefan/uwaga:tabu      →      lator nowy
dokument Q!W@E#R$T%Y^U&I*O(P)-[+]ala{ma}kota|psa\rybki<>żół"wia?.txt'

CZYSTE=$(echo -n "$ZLE" | sanitize)

echo "$CZYSTE"
```

Wynik działania powyższego skryptu jest następujący:

```
/home/stefan/uwaga_tabu_lator_nowy_dokument_Q_W_E_R_T_Y_U_I_O_P_____
ala_ma_kota_psa_rybki__żół_wia_.txt
```

Rzeczywisty przykład ataku wykorzystującego złośliwy ciąg umieszczony w nazwie pliku opisała w sierpniu 2025 firma Trellix[4].

Przy okazji warto wspomnieć o **zmiennej** `TMOUT`. Można przed wykonaniem polecenia `read` przypisać jej wartość liczbową – w ten sposób ustawia się **czas** (w sekundach), **po którym Bash przerwie oczekiwanie na wprowadzenie danych**. Uniemożliwi to np. zablokowanie skryptu w trakcie działania przez niepodanie oczekiwanego wejścia.

## O filozofii porównań i naturze nawiasu

Czasem trzeba zadawać sobie ważne pytania. Ponieważ ostateczna odpowiedź na wielkie pytanie o życie, wszechświat i całą resztę jest już znana*, proponuję następujące trzy kolejne:
czym jest [, czym jest [[ oraz czy 41 > 42?

Obsadzony w roli wyroczni bash odpowiada na dwa pierwsze pytania dość pewnie:

```
$ type -a [
[ is a shell builtin
[ is /usr/bin/[
[ is /bin/[

$ type -a [[
[[ is a shell keyword
```

Czy jednak zagadnienie wielkości liczby 41 wobec 42 jest rozstrzygalne?

Spróbujmy zadać takie pytanie, pamiętając, że zmienna $? zawiera kod wyjścia poprzedniego polecenia, czyli pożądaną odpowiedź. Będzie równa zero dla odpowiedzi twierdzącej (test logiczny zwrócił prawdę) i jeden dla przeczącej (test logiczny zwrócił fałsz)**.

Jak to zrobić?

```
$ [ 41 > 42 ]    # składnia formalnie poprawna, ale zaraz okaże się,
$ echo $?        # że jest to ŹLE zadane pytanie, a bash odpowie
0                # błędnie, tzn. że 41 JEST większe od 42
$ ls
42               # na dodatek powstał tu plik, o który nie prosiłem
```

To nie było poprawne sprawdzenie.

Spróbujmy zrobić to inaczej:

```
$ [[ 41 > 42 ]]  # zadajmy to samo (z pozoru!) pytanie, ale inaczej
$ echo $?
1                # tym razem poprawnie: 41 NIE jest większe od 42
```

**Dlaczego pierwszy z podanych przykładów nie działa poprawnie?**
Ponieważ nawias kwadratowy otwierający [ jest poleceniem powłoki (a dokładnie: odpowiednikiem komendy test). Całość jest traktowana przez powłokę jak typowe polecenie, nie jak operacja porównania. Standardowo komenda test (oraz jej odpowiednik, czyli polecenie [ ) oczekuje napisów jako argumentów; jeśli otrzyma pojedynczy argument (bez operatorów), sprawdza, czy nie jest on pustym ciągiem bądź nieistniejącą zmienną. Znak większości (>) jest zaś traktowany w typowym poleceniu powłoki jako przekierowanie wyjścia.

* I brzmi oczywiście: 42.

** Każdy program zwraca po zakończeniu wartość liczbową (kod wyjścia). W testach logicznych w powłoce **zero** to **sukces**/logiczna **prawda**, zaś dowolna wartość **niezerowa** (w tym **jeden**) to **porażka**/logiczny **fałsz**. Kończący się program sygnalizuje kodem wyjścia 0: „udało mi się zrobić wszystko poprawnie". Dowolna inna wartość świadczy o problemie, więc może być traktowana jako kod błędu o określonym znaczeniu opisanym w dokumentacji (zajrzyj np. do man tar i man zip, pod koniec obu stron podręcznika znajdziesz listę zwracanych wartości/kodów błędów).

Innymi słowy: **w pierwszym przykładzie nie mamy do czynienia ze sprawdzeniem, „czy 41 jest większe od 42”**, ale z dialogiem z bashem o treści:

– Czy 41 istnieje? [ 41 ..... ]
– Ciekawe pytanie. Sprawdzam... zrobione. Już wiem. .....
– Świetnie. Zapisz mi to do pliku o nazwie „42”! ..... > 42
– Nie wiem, co mam tam zapisać. Jak chcesz się dowiedzieć, czy 41 istnieje, to trzeba było sobie sprawdzić kod wyjścia. Ale skoro nalegasz, to zrobię Ci pusty plik o nazwie „42”. Tylko nie zdziw się, że kod wyjścia z takiej operacji będzie zależeć jedynie od tego, czy uda mi się ten plik stworzyć (albo nadpisać).

**Dlaczego drugi przykład** (z podwójnymi nawiasami) **zadziałał prawidłowo?** Podwójny nawias kwadratowy to słowo kluczowe powłoki (*shell keyword*), czyli element jej gramatyki. Jego użycie otwiera kontekst przeznaczony specjalnie do wykonywania porównań i testów logicznych. Dlatego można w nim m.in. odwoływać się do liczb i operatorów typu >, <, =>, =<.

Uwaga: konstrukcja `[[ ]]` nie jest częścią standardu POSIX. Jest specyficzna dla `basha` (oraz niektórych nowszych powłok) i nie można zakładać, że będzie dostępna w każdym innym shellu na dowolnym systemie operacyjnym. Takie konstrukcje określane są czasem jako „bashyzmy” (*bashisms*)[5]. Aby sprawdzić, czy występują w danym skrypcie, można użyć polecenia `checkbashisms` (dostępnego, w zależności od dystrybucji, w pakiecie o takiej samej nazwie albo w `devscripts`).

Co zrobić, jeśli używana powłoka nie wspiera konstrukcji z podwójnymi nawiasami kwadratowymi? Albo inaczej: **czy polecenie `test` lub `[` może służyć do porównywania liczb?** Oczywiście, należy tylko użyć **właściwego operatora**:

```
$ [ 41 -gt 42 ]    # -gt = greater than        -lt = less than
$ echo $?          # -ge = greater or equal    -le = less or equal
1                  # -eq = equal               -ne = not equal
```

**Taki sposób porównywania jest najbardziej przenośny**, czyli ma szansę zadziałać w dowolnym środowisku. Wszystkie dostępne operatory opisuje `man [` oraz `man test` (dla polecenia zewnętrznego z `/usr/bin`) lub `man bash-builtins` (dla wersji wbudowanej w powłokę).

Częstym **błędem** jest użycie operatora = do **porównania liczb**. Służy on do **porównywania napisów**:

```
$ a="7 "; b="007"
$ [ "$a" = "$b" ]   # Bashu, czy wartości tych zmiennych są takie same?
$ echo $?
1                   # Nie są. To dwa różne napisy.
                    #  ... no co? Jestem shellem, myślę napisami!

$ [ "$a" -eq "$b" ] # Bashu, czy to są równe liczby całkowite?
$ echo $?
0                   # A, to co innego! Tak, są równe.
```

## Zmienne – bezpieczniej

Czy wielka firma o uznanej pozycji może wypuścić na rynek software z błędem powodującym… usunięcie wszystkich danych użytkownika? Oczywiście. Właśnie taką „niedoróbkę" zawierał linuksowy klient platformy dystrybucji gier Steam autorstwa Valve[6]. W jednym ze skryptów pomocniczych tego pakietu znajdowała się następująca linijka:

```
rm -rf "$STEAMROOT/"*
```

Logika ustawiania zmiennej `$STEAMROOT` czasem zawodziła, czego skutkiem u niektórych użytkowników było podstawienie przez skrypt pustej wartości, czyli wykonanie polecenia:

```
rm -rf /*
```

Klient Steam jest uruchamiany z konta zwykłego użytkownika, więc rezultatem powyższego było „jedynie" usunięcie wszystkich plików osobistych i zawartości podpiętych dysków zewnętrznych (mówimy o używanych do gier komputerach typu desktop).

**Skrypt powinien sprawdzać poprawność danych wejściowych**, np. pochodzących z argumentu. Jeżeli np. oczekiwana jest ścieżka do katalogu, można zastosować konstrukcję w rodzaju:

```
test -d "$SCIEZKA" || echo "$SCIEZKA nie jest poprawną nazwą katalogu" && exit
```

Warto stosować dodatkowe sprawdzenie także przed potencjalnie destrukcyjnymi operacjami, jak skasowanie katalogu czy nadpisanie dużej liczby plików.

Bash, w przeciwieństwie do niektórych starszych powłok, zawiera obsługę tablic (indeksowanych oraz asocjacyjnych), a także kilka ciekawych, mniej znanych możliwości związanych z obsługą zmiennych pozwalających tworzyć skrypty bardziej odporne na błędy (a co za tym idzie: bezpieczniejsze). Oto one.

**Zmienne *readonly* (czyli stałe)** – po ustawieniu nie można zmienić ich wartości ani usunąć ich poleceniem `unset`. Można je zadeklarować na trzy sposoby:

```
$ declare -r zmienna="wartość"
$ typeset -r zmienna="wartość"
$ readonly zmienna="wartość"
```

**Zmienne typu liczba całkowita (integer)** – standardowo zmienne powłoki są napisami (niektórzy przewrotnie mówią, że jest to język *string-typed*). Polecenie `declare -i ZMIENNA` tworzy zmienną, która zawsze będzie przechowywana w pamięci jako liczba całkowita, przy przypisaniu będzie zaś m.in. akceptować składnię z operatorami matematycznymi, zapis ósemkowy (z zerem na początku) czy szesnastkowy (zaczynający się od `0x`). Napisy przy przypisaniu traktowane są jako nazwy zmiennych (bez potrzeby poprzedzania znakiem dolara).

```
$ declare -i MEGABAJT="1024*1024"
$ echo $MEGABAJT
```

```
1048576
$ a=4;b=3
$ W1="a+b"              # W1 to zwykły string
$ declare -i W2="a+b"   # W2 to integer wyliczany w chwili przypisania
$ echo $W1
a+b
$ echo $W2
7
$ W2=c             # zmienna c nie istnieje, więc liczbowo ma wartość 0
$ echo $W2
0
```

**Wartość domyślna zmiennej** – jedna z konstrukcji związanych z substytucją zmiennych pozwala odwołać się do zmiennej z zastrzeżeniem, że jeśli nie będzie ona istnieć lub będzie pusta, należy podstawić zadeklarowany z góry ciąg. Ma ona postać `${zmienna:-wartośćDomyślna}`. Przykład:

```
#!/bin/bash
echo "Podaj ścieżkę do pliku konfiguracyjnego lub wciśnij Enter: "

read SCIEZKA

PLIK_KONF="${SCIEZKA:-/etc/aplikacja/settings.conf}"
```

Zwróć uwagę na brak spacji między dwukropkiem a minusem. Gdyby się tam pojawiła, `bash` odczytałby ją jako zupełnie inną konstrukcję: rozwinięcie podciągu (*substring expansion*). Podobnych składniowo konstrukcji jest zresztą więcej, ale nie chcę zbaczać z tematu, czyli zagadnień związanych z bezpieczeństwem. Jeśli temat Cię zaciekawił, odsyłam do podręcznika (`man bash`). Zwracam jednocześnie uwagę, że wspomniane konstrukcje są bashyzmami.

## Włączanie zawartości z zewnętrznych plików

Szczególną ostrożność trzeba zachować, jeśli skrypt uruchamia inne skrypty lub włącza je za pomocą `source nazwa` lub `. nazwa`*. Powinny być to pliki, których nie może zmodyfikować niepowołany użytkownik (np. w celu podłożenia poleceń do wykonania).

Przy okazji powtórzę to, o czym wspominałem już w rozdziale 2: konstrukcje typu: `curl http://domena.com/skrypt.sh | sh -c` to **bardzo zła praktyka** (użycie `sudo sh` podnosi taki przykład do kategorii „koszmarnie złe”). Trudno mi wyobrazić sobie gorszy przypadek całkowitego braku walidacji wejścia niż pobieranie z zewnątrz już nie argumentów, ale całego skryptu (w sposób zautomatyzowany i bez analizy zawartości)**.

* Znak kropki na początku linii to polecenie (tak, polecenie!) równoważne poleceniu `source`.

** Warto też wiedzieć, że taka technika jest lubiana przez atakujących, pozwala bowiem pobrać i uruchomić złośliwe oprogramowanie bez pozostawiania śladu na dysku.

## Sekrety na widoku

Częstą sytuacją w skryptach jest uruchamianie programów klienckich, które nawiązują połączenia z serwerami wymagającymi uwierzytelnienia, np. `mysql` logujący się do bazy danych (nazwą użytkownika i hasłem) czy `curl` używany do komunikacji z jakimś API i wysyłający w nagłówku żądania token autoryzacyjny.

Kwestia bezpiecznego obchodzenia się z danymi uwierzytelniającymi składa się tak naprawdę z dwóch problemów.

**Po pierwsze: sekrety trzeba przechowywać bezpiecznie**. Jeśli są umieszczone otwartym tekstem w skrypcie, to musi on mieć uprawnienia ustawione w taki sposób, aby niepowołani użytkownicy nie mogli go odczytać. Jeśli ze skryptu korzysta wiele osób (np. leży w `/usr/local/bin`, a każdy użytkownik uruchamia go ze swoimi danymi uwierzytelniającymi), to dane takie powinny być trzymane w osobnych plikach (np. pełniących funkcję konfiguracji, którą użytkownicy trzymają we własnych katalogach domowych).

**Po drugie: lista procesów wraz z ich argumentami zwykle jest widoczna dla wszystkich użytkowników w systemie**. Istnieją nawet narzędzia do jej ciągłego śledzenia i rejestrowania, np. pspy[7]. Podawanie danych uwierzytelniających jako argumentów polecenia (np. `mysql --user=konto --password=haslo -h serwer baza`) stwarza ryzyko ich ujawnienia i warto rozważyć, czy przechowywanie ich w pliku konfiguracyjnym specyficznym dla danej aplikacji nie jest lepszym pomysłem. W rozdziale 7 opisuję też sposoby na ukrycie listy procesów przed użytkownikami (zob. → *Ukrywanie procesów i ochrona systemu plików /proc* R07s295).

## Nie przeszkadzać sąsiadom, nie zostawiać po sobie śmieci

Jeśli skrypt potrzebuje plików wyjściowych (wynikowych, logów, tymczasowych itp.), warto zaprojektować go tak, aby każda uruchamiana instancja tworzyła swoje własne, z unikalnymi nazwami. W razie uruchomienia takiego skryptu dwukrotnie nie pojawią się konflikty między nimi, a Ty oszczędzisz sobie nerwów przy analizowaniu problemów. Do generowania unikalnych nazw plików można wykorzystać np. PID procesu powłoki, w której wykonywany jest skrypt (zmienna `$$`), polecenie `date "+%s.%N"`, czy zmienną `$SRANDOM`*. Tworzenie plików tymczasowych o losowych nazwach można też powierzyć komendzie `mktemp`.

Dobrze wychowany skrypt nie pozostawia po sobie plików roboczych, szczególnie jeśli zawierają one informacje poufne. Najlepszymi manierami wykazują się takie, które potrafią posprzątać po sobie także w razie przerwania pracy.

Polecenie `trap` pozwala zdefiniować zachowanie skryptu w razie otrzymania sygnału przerywającego jego działanie (np. z powodu naciśnięcia Ctrl+C, zerwania sesji SSH czy zamknięcia okna terminala). W momencie otrzymania przez skrypt wskazanego sygnału wykonywane jest ustalone polecenie bądź funkcja. Przykład:

* Uwaga: `SRANDOM`, a nie `RANDOM` – ta druga ma zakres jedynie do 32767. Patrz: `man bash`.

```
#!/bin/bash
PLIK_TYMCZASOWY=$(mktemp)

# definicje reakcji na sygnały
trap "echo 'Ten skrypt ignoruje wciśnięcia Ctrl+C'" SIGINT
trap posprzataj SIGHUP SIGQUIT SIGABRT

function posprzataj()
{
    echo "Przerywam pracę. Trwa sprzątanie..."
    rm "$PLIK_TYMCZASOWY"
    echo "Koniec."
    exit
}

# właściwa część skryptu wykonująca pracę...
```

Zamiast nazw sygnałów można używać ich numerów. Spis sygnałów (nazwy, numery i znaczenie) można znaleźć np. w podręczniku: `man 7 signal`. Oczywiście nie da się zdefiniować przechwycenia sygnału numer 9 (`SIGKILL`).

## Znaleźć i nie zgubić: find, -exec oraz xargs

Opis poniżej dotyczy GNU `find`, czyli odmiany dostępnej w większości dystrybucji Linuksa i zawierającej nieco więcej opcji, niż definiuje to standard POSIX.

Program `find` to potężne narzędzie, zwłaszcza gdy weźmiesz pod uwagę fakt, że można połączyć wyszukiwanie plików według zadanych kryteriów z wykonaniem na nich dowolnej akcji. Listę znalezisk można bowiem przekazać do wskazanego polecenia na trzy sposoby (a w zasadzie pięć):

1. przełącznik `-exec` (w dwóch odmianach: ze średnikiem i z plusem),
2. przełącznik `-execdir` (również w dwóch odmianach);
3. przekazanie całej listy przez potok do `xargs`.

Jak to z potężnymi narzędziami bywa, takie połączenia mogą spowodować poważne zamieszanie, jeśli coś pójdzie nie tak. Nie mam tu na myśli jedynie pomyłki operatora. Musisz zdawać sobie sprawę, że **ścieżki i nazwy plików są niezaufanymi danymi wejściowymi** (tzn. potencjalny atakujący może podłożyć plik o złośliwej nazwie).

Klasyczne wywołanie `find` z `-exec` może mieć dwie formy*:

```
# załóżmy, że wyszukiwanie zwraca 50 plików:

$ find /ścieżka -jakieś -kryteria -exec przetwarzarka {} \;
# program 'przetwarzarka' zostanie wywołany 50 razy (jeden plik naraz)

$ find /ścieżka -jakieś -kryteria -exec przetwarzarka {} \+
# program 'przetwarzarka' zostanie wywołany jednokrotnie z 50 argumentami
```

Używając drugiej konstrukcji, trzeba pamiętać, że nie każde polecenie jest w stanie poprawnie i wydajnie obsłużyć listę o długości setek czy tysięcy argumentów.

* Dla przypomnienia: w zależności od powłoki konieczne może być wyescape'owanie (poprzedzenie znakiem ucieczki, czyli backslashem) takich znaków, jak: końcowy średnik, końcowy plus, a nawet nawiasy klamrowe wskazujące miejsce podstawienia znalezionej ścieżki.

Obie konstrukcje są dotknięte problemem, który w bezpieczeństwie znany jest jako *race condition* lub TOCTTOU (*time-to-check to time-to-use*). Między stworzeniem przez `find` listy a faktycznym wykonaniem polecenia na – np. – tysiąc pięćsetnym argumencie może minąć sporo czasu. W tym momencie atakujący może podmienić fragment istniejącej (i już znalezionej) ścieżki na link symboliczny do innego miejsca, powodując, że akcja zostanie wykonana na zupełnie innym pliku.

Zamiast `-exec` (oraz dotkniętej tym samym problemem konstrukcji `-print0 | xargs -0`) należy używać `-execdir`, które ma wbudowane zabezpieczenia przed „wrogimi" działaniami środowiska uruchomieniowego, nie tylko przed *race condition*. Przykładowo plik o nazwie `'-rf $HOME'` (uwaga: nazwa rozpoczyna się od spacji) podstawiony do `-execdir rm {}`) nie spowoduje katastrofy. Zainteresowanym polecam dokumentację `find` szczegółowo opisującą konstrukcję tych zabezpieczeń[8].

Wspomnę przy okazji, że jeśli trzeba tylko usunąć znalezione pliki, w GNU `find` nie ma w ogóle potrzeby wywoływania `rm` jako zewnętrznego polecenia przez `-execdir` czy `xargs`, ponieważ program ma zaimplementowaną akcję `-delete`:

```
# usuń z archiwum pliki starsze niż 90 dni (według daty modyfikacji)
$ find /opt/archiwum -type f -mtime +90 -delete
```

Częstym błędem w wywołaniu `find` z kryteriami typu `-name '*.abc'` jest pomijanie cudzysłowów, co może spowodować rozwinięcie `*.abc` przez powłokę jako wzorca nazwy (*globbing pattern*):

```
# prawidłowo:
$ find /opt/archiwum -type f -name '*.jpg'
# find, sporządzając listę plików sam, sprawdzi, czy pasują do wzorca

# błędnie:
$ find /opt/archiwum -type f -name *.jpg
# jeśli w bieżącym katalogu są pliki pasujące do maski *.jpg, to powłoka
# podstawi ich nazwy przed uruchomieniem find i w rzeczywistości wykona np.:
$ find /opt/archiwum -type f -name IMG001.jpg IMG002.jpg IMG003.jpg
# jeśli takich plików akurat nie ma, polecenie zadziała w oczekiwany
# sposób, ale jest to zdawanie się na przypadek – nie rób tak
```

Jeżeli korzystasz z `xargs` (np. w połączeniu z `find`), pamiętaj, że białe znaki w nazwach plików (spacja, tabulator, znak końca wiersza) mogą spowodować nieprawidłowe przekazanie nazwy. Jedynym jednoznacznie bezpiecznym separatorem jest znak zerowy (kod ASCII 0, *NULL*, `'\0'`), ponieważ nie może być on częścią nazwy pliku w żadnym systemie operacyjnym. Zarówno `find` w wydaniu GNU, jak i `xargs` w wersji dostępnej na większości Linuksów obsługują ten sposób oddzielania elementów listy po podaniu odpowiedniej opcji:

```
$ find /ścieżka -kryteria -print0 | xargs -0 jakieśpolecenie
```

W niektórych systemach konieczne może być użycie `xargs -r0`.

## Jeszcze raz: nie zostawiać po sobie śmieci

Nie trzeba chyba nikogo przekonywać, że funkcja autozapisu dostępna w edytorach tekstowych to wybawienie, które ratuje przed utratą wyników pracy. Niestety może ona czasem obrócić się przeciw Tobie. Na przykład skanery szukające podatności w aplikacjach webowych lubią sprawdzać, czy w serwowanych katalogach nie zapodziały się pliki autozapisu edytora Vim o nazwach `config.php.swp` i podobnych. Kto wie, może robocza kopia edytowanego niegdyś konfigu zawiera loginy, hasła lub klucze API…

Upewnij się, że na serwerze, który wystawia pliki do Internetu, serwowane katalogi nie zawierają żadnych zapomnianych *dotfiles* (ukrytych plików z nazwą zaczynającą się od kropki). Wyjątkiem mogą być `.htaccess` używane przez serwer Apache HTTP, ale w → *Serwery WWW* R06s241 tłumaczę, dlaczego lepiej z nich nie korzystać.

Dla porządku wspomnę jeszcze o katalogu `.git`, którego obecność w miejscu dostępnym dla osób niepowołanych niesie ze sobą co najmniej dwa zagrożenia. Po pierwsze, może on przechowywać poświadczenia umożliwiające dostęp do repozytorium. Po drugie zaś, pozwala odtworzyć zawartość repozytorium (włącznie z poprzednimi wersjami plików!), co również może skutkować ujawnieniem poufnych informacji[9].

## Od A do… czego?

Polecenia takie jak `sort` czy `ls` (które w standardowym trybie wyświetla pliki w kolejności posortowanej według nazw) muszą oczywiście znać kolejność alfabetyczną, by wykonać sortowanie. Jak powszechnie wiadomo, wygląda ona tak:

```
A B C D E F G H I J K L M N O P Q R S Š Z Ž T U V W Õ Ä Ö Ü X Y
```

Coś się nie zgadza? Wszystko jest w porządku, „Z” jest mniej więcej w dwóch trzecich, a „Y” to ostatnia litera. Tak właśnie wygląda alfabet… estoński. Czy to wydumany przykład? Pewnie sam byłbym gotów kłócić się, że tak, gdyby nie to, że spotkałem się z tak skonfigurowanym systemem osobiście i dobre kilkadziesiąt minut zajęło mi zrozumienie, czemu „sortowanie zgłupiało”.

Od tamtego czasu stosuję się do dwóch prostych reguł:

1. **Jeśli nie masz pewności co do ustawień regionalnych** (alfabet i jego kolejność, separatory tysięcy w liczbach, format zapisu daty i godziny), to nie zakładaj niczego na ich temat. **Narzuć je**, ustawiając odpowiednią zmienną środowiskową (przed poleceniem lub na początku skryptu) albo odpowiednią opcję polecenia. Przykładowo `date` czy `ls` pozwalają sterować formatem, w jakim wypisana będzie data i godzina.
2. W wyrażeniach regularnych (np. w `grep`) **nie używaj `[a-z]` lub `[A-Z]`** w znaczeniu „dowolna mała/wielka litera” (taki zapis nie jest poprawny nawet dla języka polskiego: pomija Ź oraz Ż). Zamiast tego użyj odpowiednio `[:lower:]`, `[:upper:]` bądź `[:alpha:]` (małe/wielkie/dowolne litery).

**Ustawieniami regionalnymi** sterują zmienne `LANG` oraz `LC_*` . Każda z nich może być nieustawiona, mieć przypisaną wartość `"C"` (wyłączenie mechanizmów inter-

nacjonalizacji, język angielski, US-ASCII bez unikodu) lub właściwą dla danego języka i kraju, np. `en_GB.UTF-8` (oczywiście dany język musi być zainstalowany w systemie). Wiele programów dopasowuje swoje wyjście i interfejs (a czasem nawet język komunikatów zapisywanych w logach) do ustawionego języka*.

Zmienne `LC_*` mają priorytet nad ogólną `LANG`: pozwalają niezależnie od języka zmienić format zapisu liczb, czasu czy walut; ich dokładną listę zawiera `man 7 locale` (warto też zajrzeć do `man utf-8`). Globalne ustawienie systemu w większości dystrybucji zmienia się za pomocą `localectl` (ale ta używana przez Ciebie może działać inaczej – zajrzyj do dokumentacji).

Najprostszym sposobem na to, by środowisko zachowało się przewidywalnie pod względem internacjonalizacji, jest jej wyłączenie przez `LANG=C` lub `LANG=C.UTF-8` (to drugie nie każdy system wspiera) albo ustawienie wybranego języka (np. `LANG=en_US.UTF-8`). W tym pierwszym wypadku warto sprawdzić, czy nie powoduje on przejścia używanych w skrypcie programów w tryb nieobsługujący UTF-8 (i problemów z przetwarzaniem plików zawierających znaki spoza standardowego zestawu ASCII). Dla całkowitej pewności można jeszcze usunąć przez `unset` zmienne `LC_*`.

## Debugging skryptów

Shellcheck to narzędzie, które powinna znać każda osoba pracująca ze skryptami powłoki. Jest dostępne w postaci polecenia (w repozytoriach większości dystrybucji) oraz strony internetowej[10], na której można wkleić skrypt do sprawdzenia (trzeba oczywiście pamiętać, aby na tej ostatniej nie wklejać wrażliwych danych takich, jak: zmienne zawierające nazwy plików czy hostów, adresy IP albo sekrety, np. hasła**). Narzędzie nie tylko wychwytuje błędy składniowe, ale przede wszystkim zwraca uwagę na stosowanie niebezpiecznych konstrukcji i praktyk, np. pomijanie cudzysłowu tam, gdzie należałoby go umieścić. Strona projektu na GitHubie zawiera także wiele przykładów źle napisanego kodu[11].

Od osoby zajmującej się testowaniem oprogramowania usłyszałem kiedyś, że „analizowanie skryptów Basha to koszmar, tam się nie da nic debugować". Nie do końca zgadzam się z tym stwierdzeniem, istnieje bowiem funkcja, która znacząco pomaga w analizie sposobu działania skryptów i szukaniu przyczyn błędów. Chodzi o `set -x`, które powoduje, że przed wykonaniem każdej komendy `bash` wypisze na wyjście to, co dokładnie będzie wykonane – już po rozwinięciu aliasów i tyldy, podstawieniu wartości zmiennych itd. Można dodać ją na początku skryptu lub tuż przed miejscem, w którym pojawia się problem. Oto przykład takiego skryptu:

```
#!/bin/bash
set -x

for x in {01..12};    # Uwaga, takie generowanie ciągu to bashyzm.
```

* Sam staram się trzymać zasady, by globalnym językiem systemu (zwłaszcza na serwerze) był jednak angielski `en_US.UTF-8`, zaś języki narodowe były ustawieniem poszczególnych użytkowników na ich kontach.

** Nawet jeśli aplikacja webowa obiecuje przetwarzanie danych tylko po stronie klienta, to z czasem może się to zmienić niepostrzeżenie dla użytkownika.

```
do
    NEWDIR="dane_2025_$x"
    mkdir "$NEWDIR"
    mv "2025-$x.dat" "$NEWDIR"
done
```

Bez `set -x` skrypt nie wypisałby na wyjściu nic poza ewentualnymi błędami operacji na plikach. Tymczasem po wstawieniu tego polecenia na początku i uruchomieniu całości otrzymujemy:

```
stefan@serwer:/srv/dane$ ./set.sh
+ for x in {01..12}
+ NEWDIR=dane_2025_01
+ mkdir dane_2025_01
+ mv 2025-01.dat dane_2025_01
+ for x in {01..12}
+ NEWDIR=dane_2025_02
+ mkdir dane_2025_02
+ mv 2025-02.dat dane_2025_02
+ for x in {01..12}
......
```

Dzięki ustawieniu `set -x` nie ma potrzeby dodawania do skryptu niezliczonych poleceń `echo` w celu sprawdzenia, jakie są wartości zmiennych w danym momencie. Można użyć go też w powłoce interaktywnej (a gdy nie jest już potrzebne, wyłączyć przez `set +x`):

```
stefan@serwer:/srv/dane$ ls
dokument-1    skrypt-1
dokument-2    skrypt-2
dokument-3    skrypt-3

stefan@serwer:/srv/dane$ set -x

stefan@serwer:/srv/dane$ mkdir $HOME/arch{1,2,3}
+ mkdir /home/stefan/arch1 /home/stefan/arch2 /home/stefan/arch3

# Uwaga: generowanie listy przez arch{1,2,3} to też bashyzm. Ale jaki wygodny!

stefan@serwer:/srv/dane$ for i in {1..3}; do cp *-$i "$HOME/arch$i" ; done
+ for i in {1..3}
+ cp dokument-1 skrypt-1 /home/stefan/arch1
+ for i in {1..3}
+ cp dokument-2 skrypt-2 /home/stefan/arch2
+ for i in {1..3}
+ cp dokument-3 skrypt-3 /home/stefan/arch3
```

Nie edytuj (nie zapisuj zmian) skryptów, które są w trakcie wykonywania.

Grozi to wykonaniem częściowo starej i częściowo nowej wersji: rezultat będzie trudny do przewidzenia, a w skrajnym przypadku może być katastrofalny[12].

## Hardening skryptów – przydatne opcje

Utwardzanie skryptów shellowych polega przede wszystkim na przygotowaniu ich na nietypowe bądź celowo złośliwe wejście (np. nazwy plików z „dziwnymi" znakami) i przestrzeganiu dobrych praktyk omówionych wcześniej. Warto jednak pokazać jeszcze kilka przydatnych opcji powłoki.

Domyślnie `bash` kontynuuje działanie, jeśli któraś komenda w skrypcie nie wykona się poprawnie (zwróci niezerowy kod wyjścia). Polecenie `set -e` włącza tryb, w którym błąd pojedynczego polecenia przerywa wykonanie całego skryptu. Niezerowy kod wyjścia (porażka) polecenia jest dopuszczalny w miejscach, gdzie sprawdzane są warunki logiczne (`if`, `elif`, `test`, pętle, ciągi poleceń połączone `||` i `&&` itp. – szczegółowo omawia to `man bash`).

Kolejne przydatne ustawienie to `set -u`. Powoduje ono, że odwołanie się do nieustawionych wcześniej zmiennych jest traktowane jako błąd i przerywa wykonanie skryptu (domyślnie `bash` zwraca po prostu pustą wartość). Taki tryb pomaga wychwycić zarówno błędy logiczne, wynikające z błędnej konstrukcji skryptu, jak i zwyczajne literówki w nazwach zmiennych.

Przydatne jest także `set -o pipefail`. Jego włączenie zmienia sposób ustawiania kodów wyjścia poleceń połączonych potokiem. Dla ciągu `komenda1 | komenda2 | komenda3` standardowo kod wyjścia jest pobierany z ostatniej komendy. Może więc wskazywać zero (sukces), nawet jeśli wcześniejsze nie wykonały się poprawnie i zgłosiły porażkę (kod niebędący zerem). Po włączeniu `pipefail` zmienna `$?` dla takich złożonych poleceń będzie miała wartość niezerową, jeśli którekolwiek z poleceń w łańcuchu się nie uda. Można pomyśleć, że działa to bardziej intuicyjnie („czy w poprzedniej linii coś poszło nie tak?").

Jeśli Twój skrypt pobiera dane z sieci za pomocą `curl` czy `wget`, używaj HTTPS i nie wyłączaj walidacji certyfikatów. Jeśli jakiś wewnętrzny system korzysta z certyfikatu samopodpisanego albo podpisanego lokalnym CA organizacji, pobierz certyfikat CA do pliku i za pomocą odpowiedniego przełącznika wskaż go tym programom. Jeżeli ufasz takiemu CA całkowicie, możesz dodać jego certyfikat do systemowego zbioru (w zależności od dystrybucji zwykle jest to `/etc/ssl/` lub `/etc/pki/`).

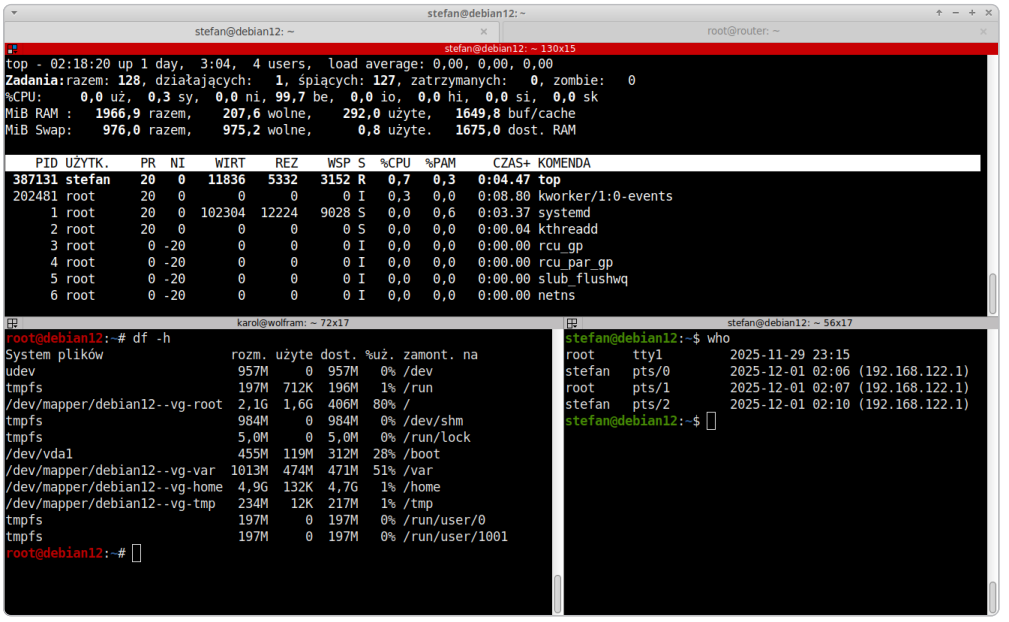
Warte przeczytania zbiory dobrych praktyk dotyczących tworzenia i bezpieczeństwa skryptów powłoki udostępniają Google[13] oraz Apple[14].

## Praca interaktywna w powłoce: porady

Powłoka to nie tylko interpreter języka skryptowego. To przede wszystkim środowisko pracy. Zależy mi na tym, aby dawała mi właściwe wskazówki wizualne i nie ułatwiała popełniania błędów. W tej części rozdziału opowiem o tym, z jakich mechanizmów można w tym celu skorzystać.

## Wygoda i ergonomia środowiska

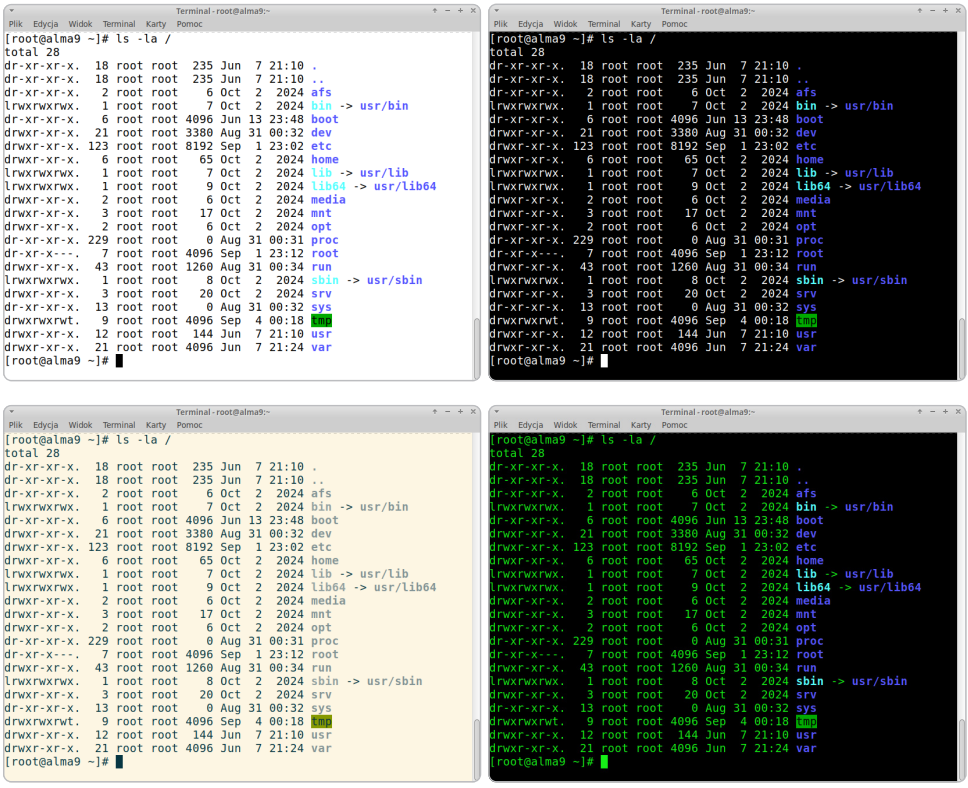
Spędzasz w shellu sporo czasu? **Dostosuj go tak, aby pomagał i nie drażnił.** Zacznij od emulatora terminala (rysunek 18*): ustaw czcionkę o odpowiedniej wielkości i wygodnym do czytania kroju. Sprawdź też, czy schemat kolorów Ci odpowiada (rysunek 19). Nawet jeśli wydaje Ci się to nieważne, pomyśl o tym jak o odpowiedniku właściwego ustawienia fotela i lusterek, gdy siada się za kierownicą. W aucie znaczenie tej czynności dla bezpieczeństwa jest poniekąd intuicyjne, zapewniam jednak, że w pracy z powłoką ma podobne znaczenie. Niewygoda i „zła widoczność" na dłuższą metę rozpraszają i skutkują irytacją, co sprzyja popełnianiu błędów.




Rysunek 18. Terminator – bardzo wygodny emulator terminala dla środowiska graficznego

**Warto znać skróty klawiszowe** pozwalające szybko zmienić rozmiar czcionki w danym oknie. Jeśli korzystasz z linuksowego środowiska graficznego, zwykle jest to Ctrl+plus/minus (w niektórych terminalach trzeba użyć Ctrl+Shift). Wypróbuj też kombinację Ctrl+kółko myszy (lub jego odpowiednik na laptopowym touchpadzie czy trackpoincie); osobiście to jej używam najczęściej.

* Emulatorów także jest wiele do wyboru. Moim absolutnym faworytem w linuksowym środowisku graficznym jest `terminator` ze względu na dostępną w menu podręcznym funkcję dzielenia okna na części, które można potem dowolnie organizować, przemieszczać, zmieniać ich rozmiar itp.




Rysunek 19. Schemat kolorów terminala ma znaczenie – dopasuj go do siebie

Wprowadź kolorowanie promptów w powłoce adekwatne do potrzeb i własnego środowiska. Służy do tego zmienna `PS1` ustawiana przez pliki konfiguracyjne powłoki, np. `~/.bashrc`. W wielu systemach korzystam z konfiguracji podobnej do tego, co domyślnie w powłoce użytkownika ustawia Debian:

```
stefan@debian12:/usr/bin$
```

```
PS1='\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ '
#    w wolnym tłumaczeniu:
#    [green]username@host[neutral]:[blue]currentdir[neutral]$

# \[\033     - kod sterujący do podania przed kodem koloru
# [01;32m    - kod koloru (tu: zielony)
# [00m       - kod wyłączający kolor (powrót do koloru standardowego
#                                     dla danego terminala)
# \]         - kod sterujący do podania po kodzie koloru/wyłączenia
# \u \h \w   - kody podstawiające username, hostname, katalog bieżący itp.
```

Więcej o kodach sterujących terminali (*ANSI escape codes*) możesz dowiedzieć się np. z Wikipedii[15] oraz prezentacji Gynvaela Coldwinda[16].

Oto przykładowy schemat ustawienia kolorów dla kilku systemów, jednoznacznie odróżniający konto zwykłego użytkownika od `roota`, własną stację roboczą od serwerów oraz maszyny deweloperskie od produkcyjnych:

```
karol@moj-laptop:~$
root@moj-laptop:~#

zwykly-user@serwer-devel:~$
root@serwer-devel:~#

zwykly-user@serwer-prod:~$
root@serwer-prod:~#

root@router:~#
```

Takie ustawienie szybko staje się interpretowaną przez mózg wizualną wskazówką i redukuje szansę wpisania `reboot` „na prodzie" zamiast „na teście". Uważaj tylko na prawidłowe escape'owanie kodów kolorów w zmiennej `PS1`.

Alternatywnie (np. jeśli masz problemy z rozróżnianiem kolorów) możesz poeksperymentować z jaskrawością, pogrubieniem bądź podświetleniem:

```
karol@moj-laptop:~$
root@serwer-devel:~#
karol@serwer-prod:~$
```

Wyniki poleceń `ls` czy `grep` bardzo zyskują na czytelności, gdy są pokolorowane. Najprościej wymusić to aliasami ustawionymi w konfiguracji powłoki:

```
alias ls='ls --color=auto'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
```

Warto definiować aliasy dla często używanych komend dla przyspieszenia pracy, np.:

```
alias ll='ls -l'
```

Włącz też podświetlanie składni w używanym edytorze: ułatwi to edycję konfiguracji czy pisanie skryptów. Jeśli trzeba, doinstaluj i włącz odpowiednie wtyczki (np. w Debianie czy Ubuntu zamiast domyślnego `vim-tiny` zainstaluj `vim-full` i wpisz `syntax on` do pliku `~/.vimrc`).

Ogromnym przyspieszaczem pracy w `bashu` są dostępne w repozytoriach dodatki `bash-completion` oraz `command-not-found`. Po zainstalowaniu pierwszego mechanizm dopełniania klawiszem Tab zyskuje znajomość składni wielu komend i potrafi adekwatnie do wpisywanego polecenia dopełniać charakterystyczne parametry typu `--przełącznik-o-długiej-nazwie=/ścieżka`. Drugi podpowiada, co doinstalować, jeśli wprowadzone polecenie nie jest dostępne w systemie. Pomoże też, jeśli zdarzy Ci się literówka w nazwie programu. Nie podchodź jednak do jego sugestii bezkrytycznie – mogą czasem zwieść na manowce. Wspominałem już o atakach typu *typosquatting* na ten mechanizm (zob. → RO2s087).

## Wstydliwa historia

Jest wpół do trzeciej w nocy. To druga awaria w tym tygodniu, na szczęście już prawie opanowana. Płacą za dyżury jak należy, razem z premią zbierze się na nadpłatę hipoteki, ale i tak masz dość. Oczy pieką, głowa ćmi. Jeszcze tylko szybki rzut oka na trzeci serwer i można wracać do łóżka…

```
root@server02:~# ssh stefan@seerver03
ssh: Could not resolve hostname seerver03: Name or service not known
root@server02:~# myS3cretP@ss
myS3cretP@ss: command not found
root@server02:~#
```

A niech to szlag. Palce okazały się szybsze od oczu i głowy: nim dotarło do Ciebie, że w nazwie hosta jest literówka, a na ekranie komunikat błędu, wstukały Twoje hasło, spodziewając się, że zapyta o nie klient SSH. Właśnie zapisało się w historii powłoki – i to w miejscu, gdzie może je zobaczyć cały zespół. Trzeba natychmiast je zmienić…

Czy na pewno?

Na razie jest tylko w pamięci procesu powłoki, nie zostało jeszcze zapisane do pliku historii. Co zrobić?

1. Wpisz `HISTSIZE=0` – ta zmienna decyduje o tym, ile elementów zapisać w historii. Ustawienie zera całkowicie wyłącza historię dla danej instancji powłoki.
2. Zamknij Basha poleceniem `exit` bądź `logout` (jeśli o wpół do trzeciej w nocy pamiętasz, czym jest `$$`, to możesz użyć `kill -9 $$`, też zadziała*).
3. Idź spać. Zostaw już ten trzeci serwer, do diabła z nim.

Opisaną metodę (bez punktu 3) można zastosować także przed wykonaniem komendy, której zapisanie w historii byłoby niepożądane (a czasem nie ma innego wyjścia, bo np. trzeba podać dane uwierzytelniające jako argument polecenia).

Istnieje jeszcze inny sposób, aby wykonane polecenie nie znalazło się w historii, jednak nie zawsze działa – jest nim wpisanie spacji przed poleceniem. Ustawienie zmiennej `HISTCONTROL` w Bashu decyduje, czy przy zapisie do historii pomijać polecenia, przed którymi znajduje się spacja (`ignorespace`), duplikaty poleceń (`ignoredups`) czy jedno i drugie (`ignoreboth`). To ostatnie ustawienie jest standardem w Debianie i Ubuntu (ale już np. nie w Fedorze/RHEL i pokrewnych).

Jeśli wszystko inne zawiedzie, usuń lub nadpisz plik `~/.bash_history`. Oczywiście jeśli w systemie skonfigurowany jest dodatkowy mechanizm rejestracji sesji (np. `tlog`) albo szczegółowa polityka mechanizmu `auditd`, usunięcie historii nie na wiele się zda. W takiej sytuacji musisz zmienić hasło.

* Jeśli nie pamiętasz, nie przejmuj się. Cofnij się o kilka stron albo zajrzyj do `man bash`.

## Jak szybko i prosto uruchomić malware... przez przypadek

Jeśli masz do czynienia z **potencjalnie złośliwym plikiem** binarnym lub skryptem (np. skopiowanym z zainfekowanego systemu), jak najszybciej **zabierz mu prawo wykonania** (`chmod ugo-x`). O ile trudno przypadkiem wpisać `./podejrzany` w wierszu poleceń, o tyle np. w Midnight Commanderze omyłkowe uruchomienie złośliwej binarki jest niestety bardzo łatwe: wystarczy ją podświetlić i nacisnąć klawisz Enter. Nie pytajcie, skąd wiem.

## Procesy i sesje w tle a blokowanie sesji terminalowej

Blokowanie ekranu może kojarzyć się ze środowiskiem graficznym i wygaszaczem, ale w konsoli wirtualnej (VT, *virtual terminal*), czyli całkowicie tekstowym środowisku*, także można to zrobić. Służą do tego narzędzia `vlock` (dostępne w zależności od dystrybucji: w pakiecie `vlock` lub `kbd`) bądź `physlock`. Uruchamia się je jak każde inne polecenie i służą do zablokowania konsoli bez wylogowywania się. Jest to bardzo przydatne, jeśli np. w tle działa jakieś zadanie (uruchomione poleceniem zakończonym znakiem `&`). Zabezpieczona konsola będzie wymagać podania hasła:

```
This TTY is now locked.

Please press [ENTER] to unlock.
```

Narzędzie `vlock` potrafi nawet pokazać „wygaszacz ekranu" w postaci ASCII-art, jeśli ma zainstalowaną odpowiednią wtyczkę.

Oczywiście zadania w tle można uruchamiać także za pomocą menedżerów sesji (np. `screen` czy `tmux`) oraz `systemd-run`. W takiej sytuacji można się wylogować, a procesy uruchomione w sesji `screen` czy `tmux` będą działać dalej**. Ten ostatni menedżer pozwala w konfiguracji zdefiniować automatyczne blokowanie konsoli przy braku aktywności: można ustawić służące do tego polecenie (np. `vlock`) i czas, po jakim blokada ma zostać aktywowana. Opisane narzędzia będą też działać w sesjach SSH.

Jeśli korzystasz z konsol wirtualnych, zwróć uwagę, że w razie przejścia komputera w stan uśpienia lub hibernacji zalogowana sesja będzie dostępna po wybudzeniu.

## Konsola współdzielona przez Internet

Istnieją usługi służące do współdzielenia terminala przez sieć, np. `tmate`, `tty-share` czy `teleconsole`. Zasada działania każdej z nich polega na instalacji programu przekazującego kontrolę nad terminalem publicznej usłudze działającej w Internecie. Połączenie z tak udostępnionym terminalem wymaga jedynie przeglądarki internetowej i unikalnego linku.

---

* Do przełączania między wirtualnymi terminalami (`tty1`, `tty2` itd.) służą skróty klawiszowe Ctrl+Alt+F1, Ctrl+Alt+F2 itd. Najczęściej pierwsza lub siódma wirtualna konsola przeznaczona jest dla środowiska graficznego, zaś na pozostałych uruchamiany jest program pozwalający na zalogowanie w trybie tekstowym.

** Jeśli w Twoim systemie się tak nie dzieje i wylogowanie „ubija" sesję `tmux`, konieczne może być włączenie lingeringu sesji w `systemd` dla danego konta (`loginctl enable-linger`).

Oczywiście jest to szybkie i wygodne, ale czy oddałbym kontrolę nad terminalem serwerowi w Internecie, utrzymywanemu przez obcą osobę? Przy moim dość restrykcyjnym podejściu do bezpieczeństwa zdecydowanie nie. Gdyby takie rozwiązanie było potrzebne mnie albo organizacji, w której bym pracował, wolałbym sam uruchomić lokalnie usługę udostępniającą terminal za pomocą WWW (używając programu `ttyd`). Zadbałbym też, aby była dostępna tylko dla tych, którzy faktycznie by jej potrzebowali (np. schowana wewnątrz firmowego VPN).

Najlepiej, moim zdaniem, zrobić coś takiego „tradycyjnie": upewnić się, że zainteresowane osoby mają na poziomie sieci dostęp do danego serwera przez SSH, i użyć współdzielonych sesji w programie `tmux`.

## CO MOŻE PÓJŚĆ NIE TAK PRZY PRACY W ŚRODOWISKU GRAFICZNYM

Z linuksowych środowisk graficznych korzystam stale od kilkunastu lat. To one są dla mnie głównym i naturalnym sposobem pracy z komputerem, w innych systemach bywam, gdy wymaga tego praca lub konieczność użycia niedostępnego pod Linuksem oprogramowania.

Czy GUI działające na systemie spod znaku pingwina jest idealnie bezpiecznym środowiskiem pracy? Nie. **Nie ma takiej kombinacji systemu i oprogramowania, która byłaby odporna na wszystkie ataki i całkowicie zwalniała z... myślenia i elementarnej ostrożności.**

**Lista zagrożeń, o których warto pamiętać**, pracując z linuksowym pulpitem:

- Pod żadnym pozorem nie loguj się do GUI jako `root`. Obecne dystrybucje i środowiska graficzne umożliwiają nieuprzywilejowanym użytkownikom bezpieczne „wyklikanie" operacji, które tradycyjnie wymagają podwyższonych uprawnień, np. montowania nośników przenośnych czy konfiguracji sieci bezprzewodowej lub VPN. Zwykle wystarczy być członkiem odpowiedniej grupy (zajrzyj do dokumentacji dystrybucji, zob. też → tabela 18 R03s123). Za mapowanie nazw grup do zezwoleń na określone operacje odpowiada mechanizm *polkit* (dawniej *PolicyKit*).
- Ilość malware'u przeznaczonego dla linuksowego desktopu jest niewielka. Nie znaczy to jednak, że nie ma go w ogóle. O zainfekowanych pakietach w sklepach z aplikacjami oraz złośliwych wtyczkach do przeglądarek internetowych i narzędzi deweloperskich pisałem już w rozdziale 2. O zaawansowanych szkodnikach opowiem więcej w rozdziale 12.
- Instaluj aktualizacje bezpieczeństwa. W Firefoksie, Chromium, pakiecie LibreOffice czy czytnikach PDF regularnie pojawiają się podatności, do których wykorzystania wystarczy otwarcie spreparowanego dokumentu. Szansa napotkania takiego pliku „w przyrodzie" jest wprawdzie niewielka, ale nie zerowa.
- Linki phishingowe w wiadomościach i fałszywe strony sklepów internetowych, banków czy operatorów płatności mogą wyglądać tak samo wiarygodnie na każdym systemie operacyjnym. Używaj dwuskładnikowego uwierzytelniania oraz sprawdzonych menedżerów haseł (np. KeepassXC), rozważ integrację

menedżera haseł z przeglądarką – taka wtyczka dodatkowo pomoże odróżnić fałszywe witryny od prawdziwych*.

- Żaden OS nie ochroni przed skutkami kradzieży danych z serwisów, w których masz konto. Nie korzystaj z tego samego hasła w różnych miejscach, raz na jakiś czas sprawdź, np. w serwisie HIBP[17], czy Twój adres mailowy nie pojawił się w jakimś wycieku.
- Wobec powszechności fałszywych reklam prowadzących do złośliwych stron absolutnie kluczowym elementem bezpieczeństwa w Internecie jest, moim zdaniem, bloker reklam (np. wtyczka uBlock Origin w Firefoksie lub uBlock Origin Lite w Chromium/Chrome/Microsoft Edge).
- Zwróć uwagę na okres wsparcia przeglądarek w używanej przez Ciebie dystrybucji, zwłaszcza w okresach przejściowych (np. Debian *oldstable*[18]) lub jeśli pakiet pochodzi z repozytoriów dodatkowych.
- Pliki z nieznanego bądź podejrzanego źródła warto sprawdzać w VirusTotal (uwaga na kwestię poufności wysyłanych tam plików – wspominałem już o tym w rozdziale 2) lub przynajmniej przeskanować lokalnie za pomocą ClamAV**. Do analizy niebezpiecznych plików można też wykorzystać maszynę wirtualną (np. z dystrybucją Remnux[19]), ale wymaga to wiedzy znacznie szerszej niż zawarta w niniejszej książce.
- Jeśli masz zainstalowany pakiet Wine służący do emulacji Windows i uruchamiania na Linuksie oprogramowania dla tego systemu, windowsowy malware także ma szansę zadziałać. Zachowaj szczególną ostrożność.
- Jeśli używany przez Ciebie menedżer plików ma opcję automatycznego montowania podłączonych nośników danych, wyłącz ją.
- W komputerach przenośnych duże znaczenie ma kwestia blokowania ekranu. Sprawdź, czy działa ono zgodnie z oczekiwaniami, także po zamknięciu klapy laptopa, wybudzeniu z uśpienia czy hibernacji. Zwróć przy tym uwagę, że w niektórych środowiskach graficznych aplikacja wygaszacza ekranu wcale nie jest do tego potrzebna. Za blokadę ekranu z jego wyłączeniem (przejściem monitora w tryb oszczędzania energii) może odpowiadać menedżer okien/sesji we współpracy z menedżerem logowania (stosowne ustawienia będą np. w panelu opcji zasilania).
- Jeśli uruchamiasz na swoim komputerze serwery usług na własne potrzeby, złośliwa strona otwarta w przeglądarce może próbować wysyłać do nich żądania[20] i w ten sposób wejść w interakcję np. z działającymi lokalnie modelami LLM. Zabezpiecz takie usługi, choćby prostym uwierzytelnieniem z użyciem innego niż domyślne hasła.

* Jeśli menedżer haseł jest zintegrowany z przeglądarką i w bazie ma prócz loginu i hasła zapisany także URL strony, podstawi automatycznie dane logowania na autentycznej witrynie, ale na sfałszowanej już nie.

** Więcej o tym skanerze antywirusowym w → *Wykrywanie złośliwego oprogramowania* R11s474.

## GUI a dostęp zdalny do systemu

Serwer SSH wraz z zaszytą w nim funkcjonalnością transferu plików przez SCP/SFTP to standardowy sposób na zarządzanie maszyną przez sieć. A co, jeśli nie wszystko, czego potrzebujesz, działa w linii poleceń? W tym miejscu spróbuję podsumować różne podejścia do zagadnienia: „mam komputer z Linuksem, używam środowiska graficznego i potrzebuję dostać się do tego zdalnie".

Uwaga wstępna: mam bardzo mało zaufania do komercyjnych usług i zamkniętych narzędzi „zdalnego pulpitu" oferujących „bezproblemowe omijanie NAT i firewalli"* – nie chcę oddawać w ręce ich dostawców tak wrażliwych informacji, jak dane uwierzytelniające i obraz pulpitu, oraz uruchamiać takiej usługi w systemie jako `root`. Jeśli chodzi o warstwę sieciową dostępu zdalnego, zdecydowanie wolę korzystać z utrzymywanych samodzielnie serwerów VPN (np. bazujących na WireGuard na tanim VPS-ie) czy radzić sobie z problemem zmiennego IP publicznego na łączach konsumenckich za pomocą usług typu *dynamic DNS*. Jeśli chodzi o warstwę aplikacyjną, dobór wymienionych niżej narzędzi także odzwierciedla to podejście.

**Linux z GUI: co skonfigurować jako „serwer zdalnego pulpitu"?**

- Jeśli masz dostęp do komputera docelowego przez SSH i sporadycznie potrzebujesz wyklikać coś w jednej czy dwóch aplikacjach używających GUI, możesz skorzystać z tunelowania X (*X11 forwarding* (rysunek 20), opcja klienta konsolowego: `ssh -X`, szczegółowo opisywałem ją w → *Tunele (forwardowanie połączeń)* R04s165). Oczywiście do wyświetlenia okna konieczny jest serwer X11 po stronie klienta (jeśli klientem jest Linux z GUI, to sprawa załatwiona; w macOS trzeba będzie doinstalować np. Xquartz, a w Windows np. xming**).
- Jeśli potrzebujesz pełnego dostępu do pulpitu, większość dystrybucji oferuje różne implementacje protokołu VNC (np. `tightvnc`, `tigervnc`) oraz XRDP, czyli serwer znanego z Windows protokołu zdalnego pulpitu (RDP). Bez problemu można za ich pomocą zamienić komputer z Linuksem w „serwer terminali graficznych" dla wielu użytkowników jednocześnie.
- Do udostępnienia przez VNC istniejącej sesji z działającego X serwera użyj `x0vncserver` (z pakietu `tigervnc`) lub `x11vnc`. Jeśli zamiast X korzystasz już z nowego serwera wyświetlania Wayland, możesz udostępnić go przez VNC za pomocą pakietu `wayvnc`.
- Pakiet `novnc` zawiera klienta VNC zaimplementowanego w HTML5 z użyciem WebSockets. W połączeniu z serwerem VNC umożliwia on udostępnienie pulpitu klientom używającym jedynie przeglądarki internetowej. Konfiguracja na bazie dokumentacji (np. bazująca na serwerze Apache HTTP) jest dość prosta, natomiast należy koniecznie uwzględnić w niej mechanizmy uwierzytelnienia (nie tylko na warstwie VNC, ale też serwera WWW, choćby z użyciem HTTP Basic Authentication***).

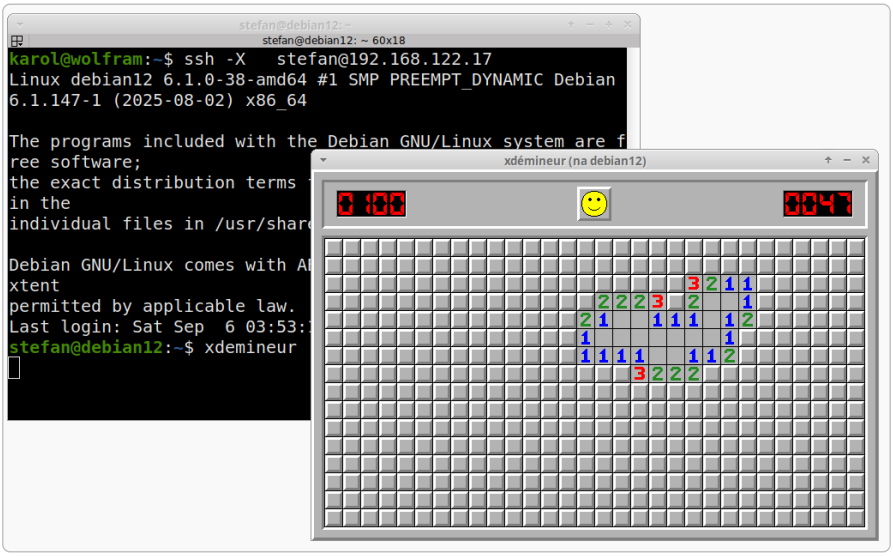
* Przykładową listę takich narzędzi można znaleźć na stronie LOLRMM, https://lolrmm.io/

** Albo użyć WSL2 lub klienta SSH z wbudowaną obsługą X11, jak komercyjny, zamknięty MobaXterm.

*** Taka konfiguracja to jeden z niewielu przypadków, w których rozważałbym użycie modułu serwera Apache umożliwiającego uwierzytelnienie z użyciem PAM, czyli kont systemowych (`mod-authnz-pam`).

W zależności od wersji i implementacji protokół VNC może mieć poważne ograniczenia pod kątem bezpieczeństwa (hasła ograniczone do ośmiu znaków, słabe szyfrowanie). Zalecaną praktyką, zwłaszcza przy połączeniach przez Internet, jest pozostawienie serwera VNC nasłuchującego jedynie na interfejsie pętli lokalnej (127.0.0.1 lub ::1) i łączenie się do niego np. za pośrednictwem tuneli SSH.

Niezależnie od zastosowanej metody komunikacji na komputerze, do którego się łączysz, **nie rekonfiguruj serwera X do przyjmowania zdalnych połączeń** i nie łącz się z nim bezpośrednio za pośrednictwem sieci: nie ma takiej potrzeby (związane z tym niebezpieczeństwa opisałem w → *Środowisko graficzne i polkit (Policykit)* R06s258).

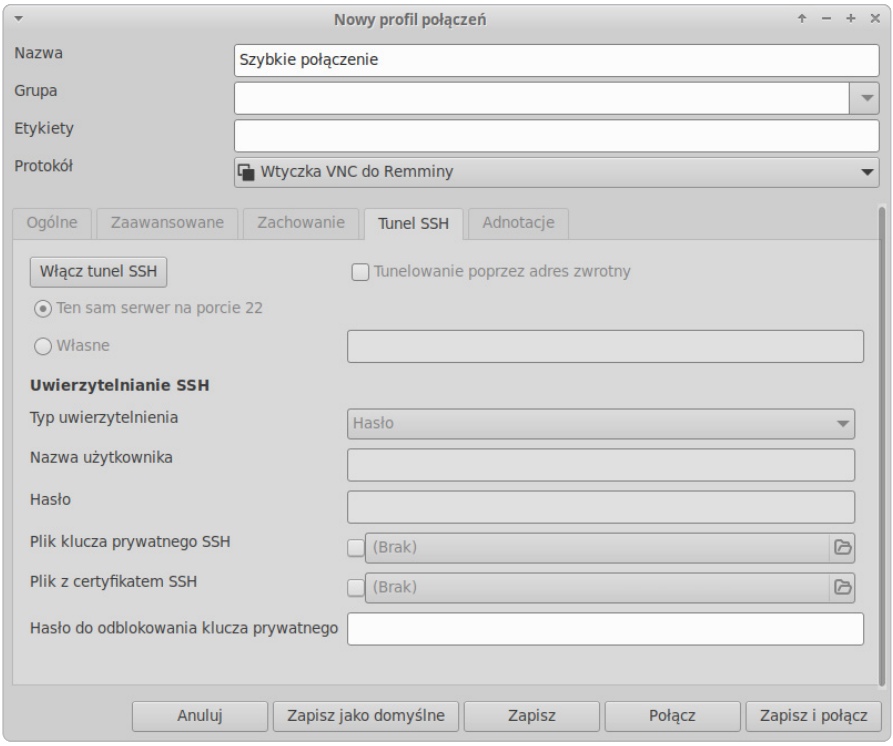



Rysunek 20. Forwardowanie X11 w praktyce: gra działa na zdalnym komputerze debian12 na koncie stefan, ale wyświetla okno na moim pulpicie

**Linux z GUI jako stacja robocza i klient „zdalnego pulpitu”.**

Jeśli regularnie pracujesz ze zdalnym pulpitem (np. administrujesz systemami), polecam program Remmina (rysunek 21). Po zainstalowaniu wtyczek obsługujących żądane protokoły to prawdziwy kombajn. Wygodnie obsłuży zarówno RDP (włącznie ze współdzieleniem folderów), jak i VNC (także tunelowane przez SSH) czy X2Go.

Oczywiście dostępne są także proste narzędzia, np. `freerdp` czy `xtightvncviewer`, wymagające podania wszystkich ustawień jako parametrów wiersza poleceń. Lekkim klientem dla tunelowanych przez SSH połączeń VNC jest `ssvnc`: prostszy od Remminy, ale umożliwiający wygodną, graficzną konfigurację i zapisanie ustawień połączeń dla różnych hostów.




Rysunek 21. Remmina: tworzenie połączenia VNC, opcje tunelowania SSH

## LINUX DLA CAŁEJ RODZINY

Linuksowy desktop ma szansę sprawdzić się całkiem nieźle w roli **komputera dla dziecka**. Mnóstwo popularnych usług dostępnych jest przez przeglądarkę, wsparcie dla gier z każdym rokiem wygląda lepiej, a znaczna część malware'u na takim systemie nie zadziała. Z podobnych powodów można także rozważyć Linuksa jako **desktop dla osoby starszej**, nie do końca samodzielnej i niemającej dużych kompetencji cyfrowych, a mimo to chcącej korzystać z dobrodziejstw sieci, np. do komunikacji z dziećmi i wnukami. Warto skonfigurować sobie w takiej sytuacji jakąś formę zdalnego pulpitu.

Za bezwzględnie konieczny w takich zastosowaniach uważam **bloker reklam** (przede wszystkim ze względu na ochronę przed oszustwami i fałszywymi reklamami na platformach społecznościowych), w tym dodanie do niego listy złośliwych domen CERT Polska[21]. W systemie należy też skonfigurować automatyczną, nienadzorowaną instalację aktualizacji bezpieczeństwa.

Do **monitorowania i ograniczania czasu** spędzanego przez dziecko przed ekranem można zastosować np. narzędzie Timekpr-next[22] (z powodzeniem używałem go przez kilka lat). Jeśli chodzi o ograniczenie tego, co dziecko może z komputerem robić, warto rzucić okiem na pakiet `malcontent` (wraz z `malcontent-gui`) albo –

alternatywnie – przetestować projekt LittleBrother[23]. Jeśli chodzi o **filtrowanie ruchu pod kątem treści** nieodpowiednich dla dzieci, najprostsze jest zastosowanie jednego z otwartych serwerów DNS oferujących taką filtrację. Miej jednak świadomość, że skuteczność takich rozwiązań jest umiarkowana i odwrotnie proporcjonalna do wieku i stopnia zmotywowania dziecka.

Blokady ustawione w telefonie czy komputerze dziecka i tak nie zdadzą się na wiele. Jeśli samo ich nie obejdzie, będzie mieć dostęp do „zakazanych" treści, choćby używając niezabezpieczonego urządzenia kolegi czy koleżanki. Ważne, by zrobiło sobie przy tym jak najmniejszą krzywdę i żebyśmy nie zabrnęli w ślepą uliczkę coraz większych restrykcji dla samych restrykcji.

To jest książka o technicznej stronie bezpieczeństwa IT, nie poradnik wychowawczy, nie będę więc wdawał się w rozważania, od jakiego wieku dzieci powinny mieć dostęp do komputera, telefonu i Internetu oraz do jakiego stopnia należy je przy tym monitorować i ograniczać. Ustalenie z nimi zasad i reguł, **uświadamianie na temat zagrożeń**, wreszcie ciekawe spędzanie czasu bez ekranów jest, moim zdaniem, ważniejsze od wyboru takich czy innych narzędzi – i to temu poświęciłbym najwięcej uwagi.

## CZY MÓJ SYSTEM MNIE ŚLEDZI?

Użytkownicy zamkniętych, komercyjnych systemów operacyjnych mają ograniczony wpływ na to, co ich własny komputer robi za ich plecami. Z roku na rok coraz szerzej otwieram oczy, czytając o mechanizmach reklamowych zaszytych w menu z programami czy instalowanej domyślnie AI mającej bez przerwy śledzić działania użytkownika, rejestrując i analizując działania na pulpicie[24]. Trudno byłoby mi uznać urządzenie, które robi coś takiego, za moją własną maszynę pracującą dla mnie.

Funkcjonalność reklamowa systemu wzbudza u mnie „jedynie" głęboką niechęć wobec bycia profilowanym i sprzedawanym reklamodawcom jako produkt oraz marnotrawienia zasobów komputera na coś, co mnie nie jest potrzebne. W drugim przypadku jednak mowa o narażeniu na niewyobrażalny poziom inwigilacji, gdyby takie mechanizmy zostały wykorzystane niezgodnie z deklarowanymi celami producenta. Wszystko jedno, czy działoby się to przy jego współpracy, np. na rozkaz jakiegoś rządu, czy też – co znacznie bardzie prawdopodobne – bez jego udziału, po obejściu zabezpieczeń przez „zwyczajnych" włamywaczy. Trudno mi w ogóle usprawiedliwiać tworzenie czegoś takiego, zaś pomysł, by włączać takie funkcje domyślnie każdemu użytkownikowi, pochodzi moim zdaniem z pogranicza szaleństwa i czystego zła.

Czy ekosystem linuksowy wygląda pod tym względem lepiej?

Zdecydowanie tak. Praktycznie każda popularna dystrybucja biurkowa składa się niemal w całości z wolnego i otwartego oprogramowania, w którego filozofię wpisane jest poszanowanie wolności użytkownika. Trudno w systemach o całkowicie jawnym kodzie stworzyć mechanizmy rażąco ingerujące w prywatność użytkowników

i pozostać niezauważonym. Nie spodziewam się, by Canonical, SUSE czy społeczności stojące za Fedorą bądź Debianem próbowały wciskać mi na siłę wszystkowidzącego szpiega, dla niepoznaki zwanego inteligentnym asystentem.

**Istnieją w systemach linuksowych mechanizmy i aplikacje, których użycie może mieć wpływ na prywatność użytkownika**, jednak cele i zakres zbierania przez nie danych są całkowicie nieporównywalne z tym, co spotkamy w produktach Microsoftu czy Apple'a.

Oto kilka przykładów:

- W Debianie dostępny jest pakiet `popularity-contest` (`popcon`), którego zadaniem jest zbieranie informacji o tym, które pakiety są zainstalowane i używane*. Pozyskane tą drogą statystyki popularności pakietów są anonimowe, jawne[25] i służą np. do ustalania, które programy umieszczać na podstawowych nośnikach instalacyjnych. Uczestnictwo w zbieraniu statystyk jest opcjonalne: instalator systemu pyta o to i daje możliwość odmowy, zaś pakiet można w każdej chwili odinstalować.
- Analogiczny pakiet można nadal znaleźć w repozytoriach Ubuntu, obecnie jednak nie jest on już instalowany domyślnie. Projekt Ubuntu przestał korzystać z jego statystyk w 2020 roku. Zbiera za to (anonimowe) informacje o konfiguracji systemu i zainstalowanym sprzęcie (za pomocą pakietu `ubuntu-report`). Znacznie więcej danych zbierają niektóre zamknięte, komercyjne programy, np. Steam.
- W chwili pisania książki trwają prace nad dodaniem mechanizmu telemetrii** do Fedory[26] oraz dyskusja nad tym, czy powinien on być domyślnie włączony czy nie[27].
- W 2012 roku autorzy Ubuntu postanowili dodać do wyszukiwarki środowiska graficznego (służącej m.in. do odnajdywania programów w menu, z racji konstrukcji interfejsu używanej często w tym celu) funkcję wysyłającą jednocześnie wpisywane zapytanie do wyszukiwarki Amazon. Wobec ogromnego sprzeciwu użytkowników oraz organizacji chroniących prywatność[28] w kolejnych wydaniach funkcja była domyślnie wyłączona, a następnie została usunięta.
- W wielu dystrybucjach dostępne są narzędzia ułatwiające wysyłanie raportów o błędach (np. `reportbug` w Debianie czy ABRT w systemach z rodziny RHEL/Fedora). Zgłoszenia błędów to podstawowy sposób informowania autorów, że coś w systemie nie działa właściwie i wymaga poprawek. Warto jednak pamiętać, że treść raportu, wraz z zebranymi informacjami o konfiguracji systemu i podanym adresem e-mail, będzie widoczna publicznie w systemie śledzenia błędów dystrybucji. Szczegóły konfiguracji sprzętowej

* To, czy pakiet jest używany, ustalane jest na podstawie daty ostatniego dostępu do jego plików (chyba że `/usr` jest zamontowany z opcją `noatime`). Wysyłanym informacjom towarzyszy generowany losowo identyfikator systemu (służący jedynie ustaleniu, ile realnie systemów wysyła dane), a całość jest szyfrowana przed wysłaniem.

** Czyli informowania autorów o tym, które funkcje są używane, z jaką konfiguracją i opcjami, niekoniecznie z jednoznaczną identyfikacją użytkownika. Anonimowe statystyki ułatwiają im np. decydowanie, które błędy mają wyższy priorytet (bo dotyczą większej liczby użytkowników) lub jakie funkcje można usunąć, bo prawie nikt z nich nie korzysta.

czy fragmenty logów to informacje, do których publikowania lepiej podchodzić ostrożnie. Jeszcze większej uwagi wymagają zrzuty pamięci jądra lub aplikacji (*core dumps*), które mogą zawierać np. treść otwartych plików czy klucze szyfrujące.

- Wiele aplikacji wyposażonych jest we własne, niepowiązane z dystrybucjami funkcje telemetrii oraz raportowania o błędach, np. załamania aplikacji. Chyba najbardziej znanym przykładem jest Mozilla Firefox, od lat wyświetlająca przy pierwszym uruchomieniu pytanie o zgodę na zbieranie takich danych. W 2025 roku pojawiły się jednak duże kontrowersje wobec planowanej zmiany w polityce prywatności Mozilli, z której firma się potem częściowo wycofała[29]. Przeglądarka bywa też krytykowana za to, że jej domyślną wyszukiwarką jest Google (co stanowi jednak istotne źródło przychodów projektu).
- W niektórych systemach za rozwiązywanie nazw DNS odpowiada usługa `systemd-resolved`, która zgodnie z zamysłem autorów w razie nieskonfigurowania w systemie serwerów nazw automatycznie korzysta z usług Cloudflare, Google oraz Quad9. Podejście to jest źródłem kontrowersji i niektóre dystrybucje wyłączają tę funkcję[30].

PRZYPISY

twierdza.sekurak.pl/r5


1 Cooper M., *Advanced Bash-Scripting Guide. An in-depth exploration of the art of shell scripting* [w:] *The Linux Documentation Project (TLDP)*, March 10, 2014, https://tldp.org/LDP/abs/html/

2 *POSIX – Portable Operating System Interface for UNIX, IEEE Std 1003.1-2024*, zob. *POSIX* [w:] *Wikipedia, the Free Encyclopedia*, https://en.wikipedia.org/wiki/POSIX

3 The Open Group, opengroup.org. Opis procedury: Hussain S., *Getting a PDF version of the POSIX standard document*, Unixism, July 29, 2020, https://unixism.net/2020/07/getting-a-pdf-version-of-the-posix-standard-document/

4 Bade S., *The Silent, Fileless Threat of VShell*, Trellix, August 21, 2025, https://www.trellix.com/blogs/research/the-silent-fileless-threat-of-vshell/

5 Bogaty zbiór przykładów bashyzmów zawiera np. tekst *How to make bash scripts work in dash* [w:] *Greg's Wiki*, https://mywiki.wooledge.org/Bashism

6 Valve Software, *Moved ~/.local/share/steam. Ran steam. It deleted everything on system owned by user. #3671*, GitHub, https://github.com/ValveSoftware/steam-for-linux/issues/3671

7 Breuker D., Kanzo K. (kazkansouh), socketz, *pspy – unprivileged Linux process snooping*, GitHub, https://github.com/DominicBreuker/pspy

8 Voelker B., Youngman J. i in., *Security Considerations for find* [w:] *GNU Findutils*, https://www.gnu.org/software/findutils/manual/html_node/find_html/Security-Considerations-for-find.html

9 Brizinov S., *How I made $64k from deleted files – a bug bounty story*, Medium, April 22, 2025, https://medium.com/@sharon.brizinov/how-i-made-64k-from-deleted-files-a-bug-bounty-story-c5bd3a6f5f9b

10 ShellCheck, https://www.shellcheck.net/

11 *Gallery of bad code* [w:] Holen V. (koalaman), *ShellCheck – A shell script static analysis tool*, GitHub, https://github.com/koalaman/shellcheck?tab=readme-ov-file#gallery-of-bad-code

12 https://thomask.sdf.org/blog/2019/11/09/take-care-editing-bash-scripts.html

13 Google, *Shell Style Guide*, https://google.github.io/styleguide/shellguide.html

14 Apple, *Shell Script Security*, https://developer.apple.com/library/archive/documentation/OpenSource/Conceptual/ShellScripting/ShellScriptSecurity/ShellScriptSecurity.html

15 *ANSI escape code* [w:] *Wikipedia, the Free Encyclopedia*, https://en.wikipedia.org/wiki/ANSI_escape_code

16 Coldwind G., *Terminalowe ciekawostki* [w:] *Mega Sekurak Hacking Party INTRO*, YouTube, 9 września 2025, https://www.youtube.com/watch?v=IxeBRT0bG28

17 Have I Been Pwned, HIBP, http://haveibeenpwnd.com/

18 Przykład: https://www.debian.org/releases/bookworm/amd64/release-notes/ch-information.html#browser-security

19 REMnux, *REMnux: A Linux Toolkit for Malware Analysis*, https://remnux.org/

20 Moberly Ch., *Why are developers so vulnerable to drive-by attacks?*, GitLab, September 7, 2021, https://about.gitlab.com/blog/why-are-developers-vulnerable-to-driveby-attacks/

21 CERT Polska, *Lista Ostrzeżeń przed niebezpiecznymi stronami*, https://cert.pl/lista-ostrzezen/

22 Bezverhijs E., *Timekpr-nExT, Keep control of computer usage*, https://mjasnik.gitlab.io/timekpr-next/

23 Rickert M. (marcus67), *Parental Control Application LittleBrother*, GitHub, https://github.com/marcus67/little_brother/

24 *Microsoft Recall* [w:] *Wikipedia, the Free Encyclopedia*, https://en.wikipedia.org/wiki/Microsoft_Recall

25 Allombert B., *Debian Popularity Contest*, Popcon, last generated: September 29, 2025, https://popcon.debian.org/

26 Day A. i in., *Changes/Metrics* [w:] *Fedora Project Wiki*, https://fedoraproject.org/wiki/Changes/Metrics

27 Brockmeier J., *"Opt-in" metrics planned for Fedora Workstation 42*, https://lwn.net/Articles/980598/

28 Lee M., *Privacy in Ubuntu 12.10: Amazon Ads and Data Leaks*, Electronic Frontier Foundation (EFE), October 29, 2012, https://www.eff.org/deeplinks/2012/10/privacy-ubuntu-1210-amazon-ads-and-data-leaks

29 jzb, *Mozilla reverses course on its terms of use*, LWN.net, March 3, 2025, https://lwn.net/Articles/1012788/

30 Corbet J., *Fedora and fallback DNS servers*, LWN.net, February 25, 2021, https://lwn.net/Articles/847257/