Мини гайд

Bash для Security Ops

Практический handbook:

скрипты, пайплайны и CLI-рецепты для defensive security, audit, forensics и monitoring

Ivan Piskunov · Ivan Piskunov © 2026 · релиз 1.1

Практическое приложение: рабочие скрипты, cheat sheets и examples вынесены в ZIP-пакет.

Disclaimer и ограничение ответственности

Материал предоставлен только для обучения, профессионального развития, администрирования собственных систем, легитимного тестирования безопасности и ethical hacking. **Команды и скрипты не предназначены для несанкционированного доступа, взлома, кражи данных, обхода защит, нарушения приватности, скрытого закрепления в системах или работы с чужой инфраструктурой без явного письменного разрешения.**

Автор не несет ответственности за ущерб, блокировки, потерю данных, нарушение закона, договорных обязательств, внутренних политик организаций или правил провайдеров, возникшие из-за неправильного либо неправомерного применения материалов.

Материал собран, переработан и дополнен на основе многочисленных открытых источников: официальной документации, статей, GitHub-репозиториев, блогов, справочников по CLI, энциклопедий по пентесту и defensive security, а также практических подходов из инженерной работы. Текст не является копией одного источника и задуман как самостоятельный прикладной мини handbook.

Оглавление

Предисловие: зачем нужен этот handbook
Правовые и этические рамки
Среда 2026: Linux, Windows 11, WSL и CLI-стек
Базовый шаблон безопасного bash-скрипта
JSON, XML и текст: практический парсинг security data
Linux log triage: auth.log, journalctl и audit-сигналы
Windows Event Logs из bash: WSL, PowerShell, Sysmon, Defender
Сетевой baseline и authorized port scanning
Filesystem baseline для форензики и контроля изменений
Linux security audit: пользователи, права, SSH, SUID и PATH
Password hygiene: проверка утекших паролей без раскрытия секрета
Malware triage: strings, хеши и VirusTotal без лишней утечки данных
Отчетность, автоматизация и эксплуатация
Hardening и QA самих bash-скриптов
Состав ZIP-пакета
Приложение A: ежедневные defensive one-liners
Приложение B: переменные окружения Linux и Windows
Приложение C: эксплуатационные ошибки и диагностика
Источники и ориентиры для дальнейшей работы

Предисловие: зачем нужен этот handbook

Идея простая: bash не должен быть только языком для «склеить пару команд» через «трубу» | . В руках security engineer это быстрый слой автоматизации между логами, файловой системой, API, Windows Event Log, SIEM-export, EDR-output и обычным Linux toolchain. Этот handbook специально убирает длинную вводную теорию и оставляет то, что можно положить в рабочую директорию, адаптировать под свою инфраструктуру и запустить в легальном scope.

Ожидаемый читатель уже понимает программирование, Linux, сети и базовую безопасность. Поэтому синтаксис bash объясняется не с нуля, а через практические решения: как валидировать вход, как не сломать прод, как не утечь секретами, как сделать вывод пригодным для отчета и как не превратить полезный one-liner в неуправляемый «хрупкий артефакт».

> В загруженной книге сильная рамка построена вокруг защитных операций, обработки данных, мониторинга логов, сетевого baseline, контроля файловой системы, анализа вредоносных файлов по хешам и администрирования безопасности. В этом handbook эти темы переработаны в самостоятельный практический набор, актуализированный под 2026 год и дополненный Windows 11 / Windows Server / WSL сценариями.

- Главная единица пользы — воспроизводимый скрипт с понятным входом, выходом и ограничениями.
- По умолчанию все сценарии read-only, кроме явных baseline/report output операций.
- Любое взаимодействие с сетью, API или учетными данными рассматривается через призму authorization, privacy и минимизации данных.

Правовые и этические рамки

Для cybersecurity-практики важен не только код, но и контекст применения. Один и тот же port scan может быть нормальным inventory-контролем внутри своего VLAN и нарушением закона на чужом диапазоне. Один и тот же password audit может быть полезной проверкой hygiene и грубым нарушением privacy, если собрать чужие пароли без согласия.

1. **Фиксируйте scope:** хосты, подсети, аккаунты, журналы, временное окно, владелец системы, разрешение.
2. **Не собирайте лишнее:** если задачу решает hash или event id, не тащите полный payload, секреты и персональные данные.
3. **Не выгружайте чувствительные файлы** в публичные sandbox/репутационные сервисы. Для VirusTotal в handbook используется hash lookup, а не file upload.
4. **Не используйте обфускацию как способ скрыть вредоносную активность.** Для легитимных скриптов лучше подпись, контроль хеша, ACL и code review.

Среда 2026: Linux, Windows 11, WSL и CLI-стек

Практический стек на 2026 год выглядит гибридно. На Linux остаются `bash`, `find`, `awk`, `sed`, `grep`, `ss`, `journalctl`, `auditd`, `sha256sum`, `curl`, `jq`, `osquery`, `OpenSCAP` и `ShellCheck`.

На Windows полезны `PowerShell`, `Get-WinEvent`, `wevtutil`, Windows Defender cmdlets, Sysmon/Sysinternals, а в Windows 11 и Windows Server — WSL как нормальный bridge между Linux-пайплайнами и Windows telemetry.

Для нового рабочего места минимум такой: bash 5.x, jq 1.7/1.8+, curl, coreutils, gawk, grep, sed, openssh-client, git, shellcheck, python3 для редких задач, где bash не должен изображать XML-парсер. Для Windows-части удобно иметь WSL 2, Windows Terminal, PowerShell 7+, Sysmon, Defender module и права на чтение нужных журналов.

ГОТОВ ИДТИ ДАЛЬШЕ?

Команды подготовки рабочей среды

```
# Ubuntu/Debian baseline для лаборатории
sudo apt update
sudo apt install -y bash coreutils gawk grep sed jq curl git shellcheck openssh-client

# WSL: проверка дистрибутивов и версии
wsl.exe -l -v
wsl.exe --status

# Проверка, что bash видит Windows bridge
powershell.exe -NoProfile -Command 'Get-Date'
cmd.exe /c ver
```

В 2026 году Windows telemetry уже нельзя игнорировать: Windows 11, Windows Server 2025, Defender, Sysmon и PowerShell дают много полезных сигналов. Bash здесь выступает не заменой PowerShell, а клеем: собрать JSON, прогнать через jq, положить в единый отчет и сравнить с Linux-артефактами.

Базовый шаблон безопасного bash-скрипта

Большая часть ошибок в security bash-скриптах не в regex, а в эксплуатационных мелочах: забыли кавычки вокруг переменной, перезаписали файл, съели ошибку в пайпе, оставили временный файл с секретом, не проверили зависимость, неправильно обработали пустой input. Поэтому каждый скрипт в ZIP начинается с общего каркаса.

scripts/00_lib_common.sh — общий каркас

```
#!/usr/bin/env bash
# Common helpers for Bash Security Ops scripts.
# Source this file from other scripts; do not execute it directly.

set -Eeuo pipefail
IFS=$'\n\t'
export LC_ALL=C
umask 077

ts() { date -u '+%Y-%m-%dT%H:%M:%SZ'; }
log() { printf '[%s] %s\n' "$(ts)" "$*" >&2; }
die() { log "ERROR: $*"; exit 1; }
require() { command -v "$1" >/dev/null 2>&1 || die "required command not found: $1"; }
ensure_dir() { mkdir -p -- "$1"; }

safe_name() {
  # Keep filenames boring and portable for reports/artifacts.
  printf '%s' "$1" | tr -c 'A-Za-z0-9._=-' '_'
}

need_authorization() {
  cat >&2 <<'EOF'
[scope] Run this only on systems, logs, accounts, hosts or files you own/administer
        or where you have explicit written authorization.
EOF
}
```

Базовое понимание:

- `set -Eeuo pipefail` ломает привычку молча продолжать после ошибки в середине pipeline.
- `IFS=$'\n\t'` снижает риск случайного split по пробелу в имени файла.
- `umask 077` делает новые артефакты приватными по умолчанию.
- `require` валидирует зависимости до выполнения основной логики.
- `need_authorization` напоминает про scope прямо в CLI, особенно перед сетевыми проверками.

Важно.

Strict mode не магия. Команды вроде `grep pattern file || true` в triage-скриптах допустимы, потому что отсутствие совпадений — не авария.

Важный навык — отличать «нет находок» от «сломался сбор данных».

НУ, МЕН, ЧТО К ПРАТИКЕ?

JSON, XML и текст: практический парсинг security data

JSONL alert aggregation через jq

Security tools часто отдают JSONL: один alert на строку. Для SIEM-export, EDR-output, WAF events или cloud audit logs удобно нормализовать поля в CSV и отдельно посчитать severity/source/IP. jq хорош тем, что не ломается от порядка ключей и умеет stream-like обработку без regex по JSON.

Фрагмент: нормализация JSONL в CSV

```
ensure_dir "$OUT"

log "normalizing JSONL alerts: $IN"

jq -r '
  select(type == "object") |
  [
    (.timestamp // .time // .ts // ""),
    ((.severity // .level // "unknown") | tostring | ascii_downcase),
    (.source // .sensor // .product // ""),
    (.event_id // .rule_id // .signature_id // ""),
    (.user // .username // .account // ""),
    (.src_ip // .source_ip // .client_ip // ""),
    (.message // .msg // .description // "")
  ] | @csv
' "$IN" > "$OUT/normalized.csv"

jq -sr '
  map(select(type == "object"))
  | group_by((.severity // .level // "unknown") | tostring | ascii_downcase)
  | map([ (.[0].severity // .[0].level // "unknown" | tostring | ascii_downcase), length ])
  | .[] | @csv
' "$IN" > "$OUT/severity_summary.csv"

jq -sr '
  map(select(type == "object"))
  | map(.src_ip // .source_ip // .client_ip // empty)
  | map(select(. != "" and . != null))
  | group_by(.)
  | map([ .[0], length ])
  | sort_by(.[1]) | reverse
  | .[] | @csv
' "$IN" > "$OUT/top_src_ip.csv"
```

Запуск и быстрый просмотр

```
bash scripts/01_json_aggregate_alerts.sh examples/sample_alerts.jsonl out/json
column -s, -t < out/json/severity_summary.csv
column -s, -t < out/json/top_src_ip.csv
```

XML IOC extraction без regex-героизма

XML лучше парсить XML-парсером. Regex может сработать на красивом sample, но развалится на namespace, переносах, атрибутах и вложенных элементах. В скрипте используется bash-обертка и Python stdlib `xml.etree.ElementTree`: это честный компромисс, когда нужен portable XML parsing без установки тяжелого стека.

Фрагмент: извлечение IOC из XML

```
XML=$1
[[ -r "$XML" ]] || die "XML file is not readable: $XML"
require python3

python3 - "$XML" <<'PY'
import csv, re, sys, xml.etree.ElementTree as ET
path = sys.argv[1]
text_tags = {"ip", "ipv4", "ipv6", "domain", "host", "hostname", "url", "uri", "hash", "sha1", "sha256", "md5"}
rx = {
    "ipv4": re.compile(r"\b(?:\d{1,3}\.){3}\d{1,3}\b"),
    "sha256": re.compile(r"\b[a-fA-F0-9]{64}\b"),
    "sha1": re.compile(r"\b[a-fA-F0-9]{40}\b"),
    "md5": re.compile(r"\b[a-fA-F0-9]{32}\b"),
    "url": re.compile(r"https?://[^\s'\"<>]+"),
    "domain": re.compile(r"\b(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}\b"),
}

def local(tag):
    return tag.rsplit('}', 1)[-1].lower()

root = ET.parse(path).getroot()
rows = []
for el in root.iter():
    tag = local(el.tag)
    value = (el.text or "").strip()
    if not value:
        continue
    if tag in text_tags:
        rows.append((tag, value, "tag"))
    else:
        for typ, pattern in rx.items():
            for m in pattern.findall(value):
                rows.append((typ, m, "regex"))

seen = set()
w = csv.writer(sys.stdout)
w.writerow(["type", "value", "method"])
for row in rows:
    if row in seen:
        continue
    seen.add(row)
    w.writerow(row)
PY
```

Текстовые логи: grep/awk там, где они сильны

Сырые текстовые логи по-прежнему живы: nginx, sshd, syslog, audit excerpts, appliance exports. Хороший подход — не пытаться «понять всё», а сделать несколько high-signal выборок: failed auth, successful logins, denied/panic/segfault, top IP, HTTP status counts.

Фрагмент: triage auth/syslog/nginx-like логов

```
IP='([0-9]{1,3}\.){3}[0-9]{1,3}'

log "triaging: $LOGFILE"

grep -Eai "failed password|authentication failure|invalid user|sudo:.*authentication failure" \
  "$LOGFILE" > "$OUT/auth_failures.txt" || true

grep -Eaio "$IP" "$OUT/auth_failures.txt" \
  | sort | uniq -c | sort -nr > "$OUT/top_failed_auth_ips.txt" || true

grep -Eai "accepted password|accepted publickey|session opened" \
  "$LOGFILE" > "$OUT/successful_logins.txt" || true

grep -Eai "segfault|oom-killer|kernel:.*denied|apparmor=DENIED|audit.*denied|permission denied" \
  "$LOGFILE" > "$OUT/system_security_signals.txt" || true

# Web log mode: count status codes if the line contains a quoted request and numeric code.
awk 'match($0, /" [0-9]{3} /) { code=substr($0, RSTART+2, 3); c[code]++ }
     END { for (k in c) print k, c[k] }' "$LOGFILE" \
  | sort -k2,2nr > "$OUT/http_status_counts.txt" || true

log "done: $OUT"
```

Linux log triage: auth.log, journalctl и audit-сигналы

На современных Linux-системах нужно уметь работать и со старыми файлами `/var/log/auth.log` или `/var/log/secure`, и с `journalctl`, и с audit-событиями. В incident response часто нет времени поднимать полноценный pipeline: важно быстро понять, были ли failed logons, sudo failures, новые sessions, service failures, AppArmor/SELinux denials, kernel anomalies.

Linux log triage one-liners

```
# Последний час по systemd journal, high-signal grep
journalctl --since '1 hour ago' -o short-iso \
  | grep -Eai 'failed|invalid user|sudo|denied|segfault|oom|service failed'

# По конкретному unit
journalctl -u ssh.service --since '24 hours ago' -o short-iso

# Auditd summary, если установлен audit framework
aureport --summary 2>/dev/null || true
ausearch -m USER_LOGIN,USER_AUTH -ts recent 2>/dev/null || true
```

scripts/04_journalctl_watch.sh — journal watcher

```
  esac
done

require journalctl
PATTERN='failed|invalid user|authentication failure|sudo|denied|segfault|oom|panic|apparmor|audit|usb|new session|service
started|service failed'
args=(--since "$SINCE" -o short-iso)
[[ -n "$UNIT" ]] && args+=(-u "$UNIT")
[[ $FOLLOW -eq 1 ]] && args+=(-f)

journalctl "${args[@]}" | grep -Eai --line-buffered "$PATTERN" || true
```

- Для production лучше писать результаты в отдельный каталог с timestamp и hostname.
- Не путайте отсутствие вывода с отсутствием события: проверьте права чтения journal и ротацию логов.
- Для auditd держите правила отдельно от triage-скрипта; handbook не меняет audit rules автоматически.

Windows Event Logs из bash: WSL, PowerShell, Sysmon, Defender

Windows-часть сделана как bridge: bash запускается в WSL или Git Bash, дергает `powershell.exe`, получает JSON и дальше обрабатывает его как обычный файл. Это удобно, когда команда привыкла к Linux-пайплайнам, но endpoint telemetry живет в Windows Event Log.

Базовые event IDs для triage: 4624 successful logon, 4625 failed logon, 4688 process creation при включенном audit process creation, 7045 service installed, PowerShell Operational 4104 script block logging, Sysmon 1 process creation и Sysmon 3 network connection. Набор не исчерпывающий, но дает хороший старт для workstation/server review.

Фрагмент: Get-WinEvent queries из WSL

```
PS=$(cat <<'PS1'
param([int]$Hours, [string]$OutDir)
$start = (Get-Date).AddHours(-$Hours)
$queries = @(
  @{Name='security_failed_logons_4625'; Log='Security'; Id=4625},
  @{Name='security_logon_4624'; Log='Security'; Id=4624},
  @{Name='process_creation_4688'; Log='Security'; Id=4688},
  @{Name='service_installed_7045'; Log='System'; Id=7045},
  @{Name='powershell_operational_4104'; Log='Microsoft-Windows-PowerShell/Operational'; Id=4104},
  @{Name='sysmon_process_1'; Log='Microsoft-Windows-Sysmon/Operational'; Id=1},
  @{Name='sysmon_network_3'; Log='Microsoft-Windows-Sysmon/Operational'; Id=3}
)
foreach ($q in $queries) {
  try {
    $events = Get-WinEvent -FilterHashtable @{LogName=$q.Log; Id=$q.Id; StartTime=$start} -ErrorAction Stop |
      Select-Object TimeCreated, Id, ProviderName, MachineName, Message
    $path = Join-Path $OutDir ($q.Name + '.json')
    $events | ConvertTo-Json -Depth 4 | Out-File -Encoding utf8 $path
  } catch {
    $path = Join-Path $OutDir ($q.Name + '.error.txt')
    $_.Exception.Message | Out-File -Encoding utf8 $path
  }
}
PS1
)

powershell.exe -NoProfile -NonInteractive -Command "$PS" -Hours "$HOURS" -OutDir "$(wslpath -w "$OUT" 2>/dev/null || printf
'%s' "$OUT")"
log "done: $OUT"
```

Фрагмент: Defender и Sysmon status

```
PS=$(cat <<'PS1'
param([string]$OutDir)
try {
  Get-MpComputerStatus | ConvertTo-Json -Depth 4 | Out-File -Encoding utf8 (Join-Path $OutDir 'defender_status.json')
  Get-MpPreference | ConvertTo-Json -Depth 4 | Out-File -Encoding utf8 (Join-Path $OutDir 'defender_preference.json')
} catch {
  $_.Exception.Message | Out-File -Encoding utf8 (Join-Path $OutDir 'defender_error.txt')
}
try {
  Get-Service -Name Sysmon64,Sysmon -ErrorAction SilentlyContinue |
    Select-Object Name, Status, StartType, ServiceName |
    ConvertTo-Json -Depth 3 | Out-File -Encoding utf8 (Join-Path $OutDir 'sysmon_service.json')
} catch {
```

```
  $_.Exception.Message | Out-File -Encoding utf8 (Join-Path $OutDir 'sysmon_error.txt')
}
PS1
)

powershell.exe -NoProfile -NonInteractive -Command "$PS" -OutDir "$(wslpath -w "$OUT" 2>/dev/null || printf '%s' "$OUT")"
log "done: $OUT"
```

Важно.

В Windows Server и корпоративной среде часть журналов может быть недоступна без elevated privileges или GPO-настроек аудита. Скрипт пишет `.error.txt`, чтобы не маскировать отсутствие прав под «нет событий».

ИНТЕРЕСНО? ИДЕМ ДАЛЬШЕ!

Сетевой baseline и authorized port scanning

Для быстрой проверки baseline не всегда нужен Nmap. Если задача — убедиться, что на пяти известных хостах не появился неожиданный 8080/8443/3306, достаточно простого TCP connect через bash `/dev/tcp`. Это не stealth scan, не bypass, не mass scanning. Это аккуратный authorized check для своей инфраструктуры.

Фрагмент: authorized TCP connect scanner

```
require timeout
ensure_dir "$(dirname -- "$OUT")"
printf 'timestamp,host,port,status\n' > "$OUT"

while IFS= read -r host; do
  [[ -z "$host" || "$host" == \#* ]] && continue
  while IFS= read -r port; do
    [[ -z "$port" || "$port" == \#* ]] && continue
    if timeout "$TIMEOUT_SEC" bash -c "</dev/tcp/$host/$port" 2>/dev/null; then
      status=open
    else
      status=closed_or_filtered
    fi
    printf '%s,%s,%s,%s\n' "$(ts)" "$host" "$port" "$status" | tee -a "$OUT"
  done < "$PORTS"
done < "$HOSTS"

log "done: $OUT"
```

Пример запуска

```
# hosts.txt
10.10.10.5
10.10.10.6

# ports.txt
22
80
443
8080

bash scripts/06_port_baseline_scan.sh --hosts hosts.txt --ports ports.txt --out out/ports.csv
awk -F, '$4=="open" {print}' out/ports.csv
```

Замечение.

Для полноценного discovery, service/version detection и NSE используйте Nmap в согласованном scope. Bash-сканер в архиве нужен как минимальный, понятный и легко читаемый baseline tool.

НУ, А ТЕПЕРЬ ЧТО ПОСЕРЬЕЗНЕЕ! LET'S GO!

Filesystem baseline для форензики и контроля изменений

Baseline файловой системы полезен, когда нужно зафиксировать known-good состояние: свежий сервер после установки, golden image, критичная директория с конфигами, webroot, каталог systemd units. Потом можно сравнить текущие SHA-256 с baseline и увидеть добавленные, удаленные или измененные файлы.

Фрагмент: создание и проверка SHA-256 baseline

```
require sort
require sha256sum
ensure_dir "$(dirname -- "$BASELINE")"

make_manifest() {
  local root=$1
  find "$root" -xdev \
    \( -path "$root/proc" -o -path "$root/sys" -o -path "$root/dev" -o -path "$root/run" \) -prune \
    -o -type f -readable -print0 2>/dev/null \
    | sort -z \
    | xargs -0 sha256sum 2>/dev/null
}

case "$MODE" in
  init)
    make_manifest "$ROOT" > "$BASELINE"
    log "baseline created: $BASELINE"
    ;;
  check)
    [[ -r "$BASELINE" ]] || die "baseline is not readable: $BASELINE"
    tmp=$(mktemp)
    trap 'rm -f "$tmp"' EXIT
    make_manifest "$ROOT" > "$tmp"
    if diff -u "$BASELINE" "$tmp"; then
      log "no changes detected"
    else
      log "changes detected"
      exit 1
    fi
    ;;
  *) usage; exit 2 ;;
esac
```

Команды эксплуатации

```
# Создать baseline
bash scripts/07_filesystem_baseline.sh init /etc out/etc.sha256

# Проверить позже
bash scripts/07_filesystem_baseline.sh check /etc out/etc.sha256 > out/etc.diff 2>&1

# Хранить baseline лучше вне проверяемой системы или хотя бы read-only.
```

- Создавайте baseline только на заведомо чистой системе, иначе вы зафиксируете компрометацию как норму.
- Исключайте псевдо-FS: `/proc`, `/sys`, `/dev`, `/run`.
- Для критичных систем baseline должен быть подписан или храниться в защищенном хранилище.

Linux security audit: пользователи, права, SSH, SUID и PATH

Read-only audit script не чинит систему, а собирает snapshot: UID 0 аккаунты, shell-пользователи, NOPASSWD в sudoers, SSH effective config, password aging policy, world-writable директории без sticky bit, SUID/SGID sample, listening sockets. Это быстро превращается в checklist для hardening ticket.

Фрагмент: Linux security audit snapshot

```
 hostnamectl 2>/dev/null || hostname
  [[ -r /etc/os-release ]] && cat /etc/os-release
  uname -a

  section "uid_0_accounts"
  awk -F: '($3 == 0) { print }' /etc/passwd

  section "users_with_shells"
  awk -F: '($7 !~ /(nologin|false)$/) { print $1 ":" $3 ":" $6 ":" $7 }' /etc/passwd

  section "sudoers_nopasswd"
  grep -RInE 'NOPASSWD|!authenticate' /etc/sudoers /etc/sudoers.d 2>/dev/null || true

  section "ssh_effective_config_highlights"
  if command -v sshd >/dev/null 2>&1; then
    sshd -T 2>/dev/null | grep -E
'^(permitrootlogin|passwordauthentication|pubkeyauthentication|challengeresponseauthentication|usepam|maxauthtries|allowusers|
allowgroups) ' || true
  else
    grep -RInE '^(PermitRootLogin|PasswordAuthentication|PubkeyAuthentication|MaxAuthTries|AllowUsers|AllowGroups)' /etc/ssh
2>/dev/null || true
  fi

  section "password_policy"
  grep -Ev '^\s*(#|$)' /etc/login.defs 2>/dev/null | grep -E 'PASS_MAX_DAYS|PASS_MIN_DAYS|PASS_WARN_AGE|UMASK|ENCRYPT_METHOD'
|| true

  section "world_writable_dirs_sample"
  find / -xdev -type d -perm -0002 -not -perm -1000 -print 2>/dev/null | head -200

  section "suid_sgid_sample"
  find / -xdev \( -perm -4000 -o -perm -2000 \) -type f -print 2>/dev/null | sort | head -300

  section "listening_ports"
  if command -v ss >/dev/null 2>&1; then
    ss -tulpen 2>/dev/null || ss -tulpen
  else
    netstat -tulpen 2>/dev/null || true
  fi
} | tee "$OUT/audit.txt" >/dev/null

log "done: $OUT/audit.txt"
```

Запуск audit snapshot

```
bash scripts/08_linux_security_audit.sh out/audit_$(hostname)_$(date -u +%Y%m%d)
less out/audit_*/audit.txt
```

Замечение.

Для compliance-grade оценки используйте OpenSCAP, CIS Benchmarks, osquery packs или коммерческий posture management. Bash-аудит здесь — быстрый предварительный слой, который удобно читать глазами и вклеивать в issue.

Password: проверка утекших паролей без раскрытия секрета

Проверка паролей через Have I Been Pwned Pwned Passwords использует k-anonymity: наружу уходит только первые 5 символов SHA-1, а сравнение суффикса выполняется локально. Это не делает хранение plaintext password file безопасным, но существенно снижает риск по сравнению с отправкой полного пароля или полного хеша.

Фрагмент: HIBP k-anonymity lookup

```
if [[ "$line" == *:* ]]; then
    account=${line%%:*}
    password=${line#*:}
  fi
  sha=$(printf '%s' "$password" | sha1sum | awk '{print toupper($1)}')
  prefix=${sha:0:5}
  suffix=${sha:5}
  count=$(curl -fsS --retry 2 "https://api.pwnedpasswords.com/range/$prefix" \
    | awk -F: -v s="$suffix" 'toupper($1)==s {gsub(/\r/, "", $2); print $2}')
  printf '%s,%s,%s\n' "$account" "$prefix" "${count:-0}"
  sleep 1
done < "$IN"
```

Пример запуска

```
# Формат входа: user:password или просто password
printf 'alice:correct horse battery staple\n' > passwords.txt
bash scripts/09_password_hibp_audit.sh passwords.txt > out/hibp_results.csv
awk -F, 'NR>1 && $3>0 {print "exposed:", $1, "count:", $3}' out/hibp_results.csv
```

Важно.

Не запускайте такой audit без юридического и организационного разрешения. Для enterprise-процесса лучше интеграция на этапе смены пароля или offline check against approved corpus, а не сбор пользовательских паролей в файл.

Malware triage: strings, хеши и VirusTotal без лишней утечки данных

Первичная triage-процедура для подозрительного файла обычно начинается с read-only действий: `file`, `sha256sum`, `strings`, entropy-ish признаки, размер, timestamp, наличие URL/IP/domain-like строк. Если нужен внешний reputation, безопаснее сначала искать по хешу. Upload файла в публичный сервис может раскрыть конфиденциальные данные или показать adversary, что образец найден.

Минимальная локальная triage без исполнения файла

```
SAMPLE=./suspicious.bin
file "$SAMPLE"
sha256sum "$SAMPLE"
strings -a -n 6 "$SAMPLE" \
  | grep -Eai 'https?://|[a-z0-9.-]+\.[a-z]{2,}|powershell|cmd\.exe|/bin/sh' \
  | sort -u | head -200
```

Фрагмент: VirusTotal API v3 hash lookup

```
printf 'hash,malicious,suspicious,harmless,undetected,type_description,meaningful_name\n'
while IFS= read -r h; do
  [[ -z "$h" || "$h" == \#* ]] && continue
  response=$(curl -fsS --retry 2 --header "x-apikey: $VT_API_KEY" \
    "https://www.virustotal.com/api/v3/files/$h" || true)
  if [[ -z "$response" ]]; then
    printf '%s,,,,,,lookup_failed\n' "$h"
  else
    jq -r --arg h "$h" '
      .data.attributes as $a |
      [
        $h,
        ($a.last_analysis_stats.malicious // 0),
        ($a.last_analysis_stats.suspicious // 0),
        ($a.last_analysis_stats.harmless // 0),
        ($a.last_analysis_stats.undetected // 0),
        ($a.type_description // ""),
        ($a.meaningful_name // "")
      ] | @csv
    ' <<< "$response"
  fi
  sleep 16  # conservative for public API rate limits; tune for your plan
done < "$IN"
```

- Не исполняйте образец на рабочей машине.
- Не загружайте файл в публичный sandbox без approval.
- Для повторяемой triage сохраняйте hash, имя, размер, source, время обнаружения и analyst notes.

Отчетность, автоматизация и эксплуатация

Скрипт полезен только тогда, когда его результат можно прочитать через неделю. Поэтому каждый сценарий пишет CSV/TXT/JSON в out-директорию, а отчетный скрипт собирает HTML из артефактов. Для регулярного запуска используйте cron/systemd timer на Linux и Task Scheduler на Windows; для production добавьте логирование в syslog/journald/SIEM и контроль exit code.

Фрагмент: HTML report generator

```
IN_DIR=${1:-out}
OUT=${2:-report.html}
[[ -d "$IN_DIR" ]] || die "input directory not found: $IN_DIR"
```

```
html_escape() {
  sed -e 's/&/\&amp;/g' -e 's/</\&lt;/g' -e 's/>/\&gt;/g'
}

{
  cat <<EOF
<!doctype html>
<html lang="ru"><head><meta charset="utf-8">
<title>Bash Security Ops Report</title>
<style>
body{font-family:Arial, sans-serif;max-width:1080px;margin:32px auto;line-height:1.45;color:#111}
pre{background:#f4f4f4;padding:12px;border-radius:8px;overflow:auto;border:1px solid #ddd}
h1,h2{color:#222}.meta{color:#666}.warn{background:#fff4d6;padding:10px;border-left:4px solid #c90}
</style></head><body>
<h1>Bash Security Ops Report</h1>
<p class="meta">Generated: $(ts)</p>
<p class="warn">Review context before acting. A signal is not automatically an incident.</p>
EOF
  find "$IN_DIR" -type f \( -name '*.txt' -o -name '*.csv' -o -name '*.json' \) | sort | while IFS= read -r f; do
    echo "<h2>$(basename "$f" | html_escape)</h2>"
    echo '<pre>'
    head -300 "$f" | html_escape
    echo '</pre>'
  done
  echo '</body></html>'
} > "$OUT"

log "report written: $OUT"
```

Automation notes

```
# Linux cron пример: daily local audit в 08:00
0 8 * * * /opt/bash-secops/scripts/08_linux_security_audit.sh /var/reports/secops/$(hostname)-$(date -u +\%Y\%m\%d)

# systemd timer лучше для production: отдельный service + timer, журналирование через journald.

# Windows Task Scheduler идеологически: запуск Git Bash или WSL с путем к скрипту.
# Для сложной Windows-логики проще запускать PowerShell напрямую, а bash использовать для обработки JSON.
```

Hardening и QA самих bash-скриптов

Обфускация bash-кода редко дает настоящую защиту. Если скрипт запускается у пользователя, его можно прочитать, трассировать, подменить окружение или снять аргументы из process list. Для легитимной защиты важнее другое: минимальные права, ownership root:root, chmod 750/700, read-only директория, контроль хеша, подпись артефакта, секреты только через environment/secret manager, review через ShellCheck и тестовый запуск на sample data.

QA и hardening commands

```
# Проверка синтаксиса всех скриптов в пакете
bash scripts/99_selftest_syntax.sh

# ShellCheck, если установлен
find scripts -name '*.sh' -print0 | xargs -0 shellcheck -x

# Базовая защита installed scripts
sudo chown -R root:root /opt/bash-secops
sudo chmod -R go-rwx /opt/bash-secops
sudo find /opt/bash-secops/scripts -name '*.sh' -exec chmod 750 {} \;

# Контроль целостности пакета
find /opt/bash-secops -type f -print0 | sort -z | xargs -0 sha256sum > bash-secops.manifest.sha256
```

- Не храните API keys в файле скрипта. Используйте environment variables, secret store или CI/CD secrets.
- Не передавайте секреты в command-line аргументах, если их можно увидеть через process list.
- Логи ошибок должны идти в stderr, а машинно-читаемый вывод — в stdout или отдельный файл.
- Любой destructive mode должен требовать явный флаг вроде `--apply`; в этом handbook такие режимы сознательно не добавлены.

Состав ZIP-пакета к данному мини handbook

К handbook приложен отдельный ZIP. Он задуман так, чтобы его можно было распаковать в lab-директорию и сразу запускать безопасные примеры. Все скрипты имеют комментарии, понятный usage и общий disclaimer в README.

Структура архива

```
bash_security_ops_release_1_1/
├── README.md
├── scripts/
│   ├── 00_lib_common.sh
│   ├── 01_json_aggregate_alerts.sh
│   ├── 02_xml_ioc_extract.sh
│   ├── 03_text_log_triage.sh
│   ├── 04_journalctl_watch.sh
│   ├── 05_windows_event_triage_wsl.sh
│   ├── 06_port_baseline_scan.sh
│   ├── 07_filesystem_baseline.sh
│   ├── 08_linux_security_audit.sh
│   ├── 09_password_hibp_audit.sh
│   ├── 10_vt_hash_lookup.sh
│   ├── 11_defender_sysmon_wsl.sh
│   ├── 12_report_html.sh
│   └── 99_selftest_syntax.sh
├── cheatsheets/
│   ├── 01_bash_security_cli.md
│   ├── 02_jq_yq_awk_grep.md
│   ├── 03_linux_env_vars.md
│   ├── 04_windows_env_vars_and_wsl.md
│   └── 05_security_one_liners.md
└── examples/
    ├── sample_alerts.jsonl
    ├── sample_iocs.xml
    ├── auth.log
    ├── hosts.txt
    ├── ports.txt
    ├── passwords.txt
    └── sample_tree/
```

Приложение A: ежедневные defensive one-liners

Этот раздел — рабочая шпаргалка. Команды короткие, но их ценность в том, что они быстро дают первый слой наблюдаемости. Используйте их как scratchpad: сначала посмотреть глазами, потом уже автоматизировать в отдельный скрипт.

Linux defensive one-liners

```
# 1. Топ источников failed SSH за последние ротации логов
zgrep -hai 'Failed password' /var/log/auth.log* /var/log/secure* 2>/dev/null \
  | grep -Eo '([0-9]{1,3}\.){3}[0-9]{1,3}' \
  | sort | uniq -c | sort -nr | head -20

# 2. Учетки с UID 0
awk -F: '$3==0 {print $1 ":" $3 ":" $6 ":" $7}' /etc/passwd

# 3. Shell-пользователи, которые могут логиниться интерактивно
awk -F: '$7 !~ /(nologin|false)$/ {print $1 ":" $6 ":" $7}' /etc/passwd

# 4. Сервисы, которые слушают сеть
ss -tulpen 2>/dev/null || netstat -tulpen

# 5. Изменения в /etc за последние 48 часов
find /etc -xdev -type f -mtime -2 -ls 2>/dev/null

# 6. Мирозаписываемые директории без sticky bit
find / -xdev -type d -perm -0002 -not -perm -1000 -print 2>/dev/null

# 7. SUID/SGID sample для review
find / -xdev \( -perm -4000 -o -perm -2000 \) -type f -print 2>/dev/null | sort
```

Windows / WSL defensive one-liners

```
# Windows: failed logons за сутки
powershell.exe -NoProfile -Command \
  "Get-WinEvent -FilterHashtable @{LogName='Security';Id=4625;StartTime=(Get-Date).AddHours(-24)} | Select
TimeCreated,Id,ProviderName,Message"

# Windows: последние service install события
powershell.exe -NoProfile -Command \
  "Get-WinEvent -FilterHashtable @{LogName='System';Id=7045;StartTime=(Get-Date).AddDays(-7)} | Select TimeCreated,Message"

# Windows: Defender status в JSON
powershell.exe -NoProfile -Command "Get-MpComputerStatus | ConvertTo-Json -Depth 4"

# WSL: конвертация путей и открытие директории в Explorer
wslpath -w "$PWD"
explorer.exe .
```

Для регулярного использования one-liner лучше превратить в скрипт с явными параметрами, обработкой ошибок и понятным output format. One-liner хорош для discovery, но плох для долгосрочного сопровождения.

Приложение B: переменные окружения Linux и Windows

Переменные окружения часто оказываются недооцененным источником ошибок и находок. Через PATH можно случайно выполнить не ту бинарь; через TMPDIR — оставить чувствительные временные файлы; через SSH_AUTH_SOCK — понять, доступен ли agent; через SUDO_USER — восстановить исходного пользователя после sudo.

Linux environment variables

```
# Linux / Unix variables
HOME            # home directory пользователя
USER, LOGNAME   # имя пользователя
SHELL           # login shell
PATH            # порядок поиска команд; проверяйте writable directories
PWD, OLDPWD     # текущая и предыдущая директории
LANG, LC_ALL    # locale; для парсинга часто ставят LC_ALL=C
TMPDIR          # temp path; не храните секреты без контроля прав
SSH_AUTH_SOCK   # путь к SSH agent socket
SUDO_USER       # исходный пользователь после sudo
SUDO_COMMAND    # команда, выполненная через sudo
XDG_RUNTIME_DIR # runtime dir для user services
```

Windows environment variables

```
# Windows variables, полезные в WSL/Git Bash/PowerShell automation
USERNAME                # текущий пользователь
USERPROFILE             # профиль пользователя
COMPUTERNAME            # имя хоста
USERDOMAIN              # домен или локальная authority
APPDATA, LOCALAPPDATA   # roaming/local app data
PROGRAMDATA             # shared application data
TEMP, TMP               # временные директории
SystemRoot, windir      # обычно C:\Windows
PROCESSOR_ARCHITECTURE  # архитектура процесса
PSModulePath            # путь поиска PowerShell modules
Path                    # Windows PATH
```

Быстрая проверка PATH на Linux: распечатайте директории по одной и посмотрите, есть ли writable path до системных директорий. Это не полноценный exploit check, а hygiene review.

PATH hygiene check

```
printf '%s' "$PATH" | tr ':' '\n' | nl -ba
printf '%s' "$PATH" | tr ':' '\n' | while read -r d; do
  [ -n "$d" ] && [ -d "$d" ] && [ -w "$d" ] && echo "writable: $d"
done
```

Приложение C: эксплуатационные ошибки и диагностика

Даже хороший bash-скрипт ломается, если не учитывать среду. Ниже — короткий troubleshooting list, который экономит часы: он помогает понять, проблема в данных, правах, зависимостях, locale, shell compatibility или Windows/WSL boundary.

- Скрипт «ничего не нашел»: проверьте входной файл, права чтения, ротацию логов, временное окно и то, что grep-паттерн действительно совпадает с локализованными сообщениями.
- JSON parsing падает: прогоните jq empty file.json или jq -c . file.jsonl | head; часто проблема в multiline JSON или в смешанном формате.
- XML parsing падает: проверьте well-formed XML. Если файл на самом деле HTML или broken export, не пытайтесь чинить это regex-пайпом.
- Скрипт работает вручную, но не из cron: почти всегда отличается PATH, working directory, locale или права. В cron задавайте абсолютные пути.
- Windows events не читаются из WSL: проверьте, что доступен powershell.exe, что путь OutDir корректно конвертируется через wslpath, и что у пользователя есть права на нужный журнал.
- Baseline постоянно меняется: возможно, вы сканируете runtime/cache/log директорию. Для baseline выбирайте стабильный scope или явно исключайте volatile paths.

Preflight/debug commands

```
# Диагностический preflight для bash-скрипта
set -x                        # включить трассировку временно, не с секретами
bash -n script.sh             # syntax check
shellcheck -x script.sh       # static analysis, если установлен
command -v jq curl awk sed    # проверить зависимости
printf 'PATH=%s\nPWD=%s\n' "$PATH" "$PWD"
locale | sort
id
umask
```

Важно.

Не оставляйте set -x включенным в production, если скрипт работает с токенами, паролями, API keys или чувствительными путями. Трассировка легко утаскивает секреты в CI logs, journald или terminal scrollback.

Источники и ориентиры для дальнейшей работы

Ниже — не «библиография ради библиографии», а практический список мест, куда стоит идти за деталями поведения команд, новых версий и edge cases. Для production-скриптов всегда проверяйте текущую документацию конкретной ОС, дистрибутива и версии утилиты.

- GNU Bash Reference Manual — https://www.gnu.org/software/bash/manual/bash.html
- jq Manual — https://jqlang.org/manual/
- yq documentation — https://mikefarah.gitbook.io/yq/
- Microsoft Learn: Windows Subsystem for Linux — https://learn.microsoft.com/en-us/windows/wsl/
- Microsoft Learn: Install WSL — https://learn.microsoft.com/en-us/windows/wsl/install
- Microsoft Learn: wevtutil — https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/wevtutil
- Microsoft Learn: Sysmon overview — https://learn.microsoft.com/en-us/windows/security/operating-system-security/sysmon/overview
- Microsoft Sysinternals: Sysmon — https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon
- Microsoft Defender PowerShell module — https://learn.microsoft.com/en-us/powershell/module/defender/
- journalctl manual — https://man7.org/linux/man-pages/man1/journalctl.1.html
- Red Hat Security Hardening: Linux Audit — https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/security_hardening/auditing-the-system_security-hardening
- osquery documentation — https://osquery.readthedocs.io/
- ShellCheck — https://www.shellcheck.net/
- VirusTotal API v3 — https://docs.virustotal.com/reference/overview
- Have I Been Pwned API / Pwned Passwords — https://haveibeenpwned.com/api/v3
- Nmap Reference Guide — https://nmap.org/book/man.html
- OpenSCAP documentation — https://github.com/OpenSCAP/openscap/blob/main/docs/manual/manual.adoc
- Paul Troncone, Carl Albing. Cybersecurity Ops with bash / Bash и кибербезопасность — как один из тематических ориентиров по структуре защитных операций.