Уральский государственный экономический университет

Администрирование информационных систем

02.03.03 -Математическое обеспечение и администрирование информационных систем, направленность (профиль) -разработка и администрирование информационных систем

http//vikchas.ru

Лабораторная работа 2.
Тема «**Администрирование серверных и персональных ОС: автоматизация базовых операций в PowerShell**»

Часовских Виктор Петрович
доктор технических наук, профессор кафедры
ШИиКМ, ФГБОУ ВО «Уральский государственный
экономический университет

Екатеринбург 2026

Лабораторная работа 2

Цель: освоить автоматизацию типовых задач администратора (управление пользователями, правами, службами, аудитом, диагностикой) средствами PowerShell.

Общие сведения Windows 10, PowerShell и безопасность через автоматизацию

Windows 10 — персональная операционная система Microsoft, построенная на ядре NT. Архитектурно она разделена на пользовательское пространство (User Mode) и пространство ядра (Kernel Mode). Для администратора и специалиста по безопасности ключевым инструментом управления является **PowerShell** — объектно-ориентированная командная оболочка и язык сценариев, интегрированная непосредственно в ОС.

PowerShell в контексте Windows 10 выполняет несколько ключевых ролей:

- **Управление системой** — создание пользователей, групп, настройка служб, реестра, файловой системы целиком через скрипты без графического интерфейса.
- **Автоматизация безопасности** — назначение прав доступа (DACL), настройка аудита (SACL), анализ журналов событий, управление политиками.
- **Диагностика и мониторинг** — получение событий безопасности, проверка текущего состояния системы, формирование отчётов.
- **Воспроизводимость** — скрипт фиксирует конфигурацию в виде кода, который можно запустить на любом числе машин и получить идентичный результат.

Политика выполнения скриптов (ExecutionPolicy) — первый рубеж контроля: по умолчанию установлен режим Restricted, запрещающий запуск любых сценариев. Для лабораторных и административных задач используется RemoteSigned или Bypass. Это демонстрирует, что PowerShell изначально проектировался с учётом модели безопасности Windows.

Модуль Microsoft.PowerShell.LocalAccounts предоставляет командлеты для управления локальными пользователями и группами (New-LocalUser, Add-LocalGroupMember), модуль Microsoft.PowerShell.Security — для работы с ACL (Get-Acl, Set-Acl), а командлет Get-WinEvent обеспечивает доступ к журналу событий Windows. Совокупность этих инструментов делает PowerShell основной платформой для автоматизированного управления безопасностью на рабочей станции Windows 10.

Модели безопасности Windows через призму автоматизации

Локальные пользователи и группы

Модель безопасности Windows строится на понятии **субъекта безопасности** (Security Principal) — любой сущности, которой может быть назначен идентификатор безопасности (**SID**). Каждый локальный пользователь и группа получают уникальный SID при создании. Именно SID, а не имя учётной записи, хранится в правах доступа — это важно понимать при автоматизации.

Локальные пользователи объединяются в группы для упрощения управления правами. Windows 10 имеет встроенные группы с предопределёнными привилегиями: Administrators, Users, Guests, Remote Desktop Users и другие. Назначение пользователя в группу автоматически предоставляет ему все права, закреплённые за этой группой.

С точки зрения автоматизации через PowerShell управление пользователями и группами сводится к трём операциям: **создать, настроить, проверить**. Идемпотентный скрипт перед созданием пользователя проверяет его существование через Get-LocalUser, и только при отсутствии объекта выполняет New-LocalUser. Это гарантирует отсутствие ошибок при повторном запуске.

NTFS DACL — управление доступом

DACL (Discretionary Access Control List) — список управления доступом на усмотрение владельца — определяет, какие субъекты безопасности имеют какие права на объект файловой системы NTFS. DACL состоит из записей **ACE (Access Control Entry)**, каждая из которых содержит: субъект (SID), тип (Allow/Deny), права (Read, Write, FullControl и др.) и флаги наследования.

Windows обрабатывает ACE в строгом порядке: сначала явные Deny, затем явные Allow, затем унаследованные. Это означает, что запрещающая запись всегда приоритетнее разрешающей, если они явные.

При автоматизации критически важно проверять существование ACE перед добавлением — иначе каждый запуск скрипта будет добавлять дублирующиеся записи, что засоряет DACL и усложняет анализ безопасности.

SACL — аудит доступа

SACL (System Access Control List) — системный список управления доступом — контролирует не разрешение, а **запись событий аудита**. Когда субъект обращается к объекту и это обращение соответствует записи в SACL, Windows генерирует событие в журнале **Security**. SACL может аудировать успешный доступ (Success), неудачный (Failure) или оба типа.

Работа с SACL требует привилегии SeSecurityPrivilege, которой по умолчанию обладают только члены группы Administrators. В PowerShell для чтения SACL используется параметр -Audit командлета Get-Acl:

Без настроенного SACL никакие обращения к файлам и папкам не будут регистрироваться в журнале, даже если политика аудита включена.

События безопасности

Журнал **Security** в Windows Event Log фиксирует все события, связанные с безопасностью, при условии что соответствующая политика аудита включена. Ключевые идентификаторы событий (Event ID):

Event ID	Событие
4720	Создание учётной записи пользователя
4722	Активация учётной записи
4728	Добавление пользователя в группу
4624	Успешный вход в систему

Event ID	Событие
4625	Неудачная попытка входа
4663	Попытка доступа к объекту (требует SACL)
4670	Изменение прав доступа к объекту

PowerShell предоставляет мощный инструмент анализа — Get-WinEvent с фильтрацией по хэш-таблице:

POWERSHELL

```
Get-WinEvent -FilterHashtable @{
    LogName   = 'Security'
    Id        = 4720, 4728
    StartTime = (Get-Date).AddHours(-1)
}
```

Это позволяет автоматизировать мониторинг: скрипт после выполнения конфигурационных операций может сразу верифицировать, что нужные события были зафиксированы, — тем самым подтверждая не только факт выполнения команды, но и её регистрацию системой безопасности.

Идемпотентность в скриптах

Идемпотентность — свойство операции, при котором её однократное и многократное применение дают одинаковый результат. В контексте PowerShell-скриптов это означает: **скрипт можно запустить сколько угодно раз — система всегда будет приведена в одно и то же целевое состояние, без ошибок и без нежелательных побочных эффектов**.

Паттерн реализации: «Проверь — Действуй»

Универсальный шаблон идемпотентной операции в PowerShell:

POWERSHELL

```
# Паттерн: Check → Act
$existing = Get-LocalUser -Name "LabUser" -ErrorAction SilentlyContinue

if ($null -eq $existing) {
    New-LocalUser -Name "LabUser" -Password (
        ConvertTo-SecureString "Pass@123" -AsPlainText -Force)
    Write-Host "[СОЗДАН] LabUser"
} else {
    Write-Host "[ПРОПУСК] LabUser уже существует"
}
```

Ключевой элемент — -ErrorAction SilentlyContinue, который превращает ошибку отсутствия объекта в $null, пригодный для проверки условием. Этот же паттерн применяется для групп, ACE в DACL, правил в SACL и любых других объектов системы.

Идемпотентность ACE в DACL

Для прав доступа паттерн усложняется: нужно не просто проверить наличие субъекта, но и убедиться что конкретная комбинация (Identity + Rights + Type) уже существует:

POWERSHELL

```
function Set-AceIdempotent {
    param($Path, $Identity, $Rights, $Type)

    $acl      = Get-Acl $Path
    $exists   = $acl.Access | Where-Object {
        $_.IdentityReference -like "*$Identity*" -and
        $_.FileSystemRights  -eq $Rights          -and
        $_.AccessControlType -eq $Type
    }
```

```
    if (-not $exists) {
        $rule = New-Object Security.AccessControl.FileSystemAccessRule(
                $Identity, $Rights, $Type)
        $acl.AddAccessRule($rule)
        Set-Acl $Path $acl
        Write-Host "[ACE ДОБАВЛЕН] $Identity -> $Rights ($Type)"
    } else {
        Write-Host "[ПРОПУСК] ACE уже существует: $Identity"
    }
}
```

Концепция целевого состояния

Идемпотентные скрипты естественным образом реализуют **декларативный подход**: администратор описывает не шаги («создай пользователя, затем добавь в группу»), а **желаемое состояние системы** («должен существовать пользователь X с такими-то параметрами, состоящий в группе Y»). Скрипт сам определяет, какие действия нужны для достижения этого состояния — создать недостающее, пропустить существующее, исправить отличающееся.

Такой подход используется в профессиональных инструментах управления конфигурацией — DSC (Desired State Configuration), Ansible, Terraform. PowerShell-скрипт с паттерном «Проверь — Действуй» — это та же концепция, реализованная вручную.

Проверка результата

Выполнение команды и достижение целевого состояния — не одно и то же. Командлет может завершиться без ошибки, но произвести не тот эффект: пользователь создан, но заблокирован; ACE добавлен с неверными флагами наследования; аудит настроен, но политика аудита в системе отключена. **Верификация результата** — это явная проверка фактического состояния системы после выполнения операций, независимая от кода самих операций.

Профессиональный скрипт автоматизации содержит отдельный блок проверок, который запускается после всех конфигурационных действий и формирует итоговый отчёт.

Итоговый отчёт

После выполнения всех проверок скрипт формирует сводный отчёт:

POWERSHELL

```
$passed = ($results | Where-Object { $_ }).Count
$failed = ($results | Where-Object { -not $_ }).Count
$total  = $results.Count

Write-Host "`n========================================"
Write-Host "  ИТОГ ВЕРИФИКАЦИИ"
Write-Host "  Всего : $total"
Write-Host "  PASS  : $passed" -ForegroundColor Green
Write-Host "  FAIL  : $failed" -ForegroundColor $(
    if ($failed -gt 0) { "Red" } else { "Green" })
Write-Host "========================================"

if ($failed -eq 0) {
    Write-Host "  Система приведена в целевое состояние ✓" -ForegroundColor
Green
} else {
    Write-Host "  Обнаружены отклонения от целевого состояния ✗" -
ForegroundColor Red
}
```

Связь идемпотентности и верификации

Идемпотентность и верификация — взаимодополняющие механизмы. **Идемпотентность** гарантирует, что скрипт безопасно выполнится повторно.

Верификация гарантирует, что система действительно находится в целевом состоянии. Вместе они формируют замкнутый цикл управления конфигурацией:

Для выполнения лабораторной работы необходимо изучить применение:

- PowerShell как основной инструмент администрирования Windows: cmdlets, модули, конвейеры, объекты.
- Модель безопасности Windows через призму автоматизации: локальные пользователи/группы, NTFS DACL, SACL (аудит), события безопасности.
- Идемпотентность в скриптах: скрипт должен безопасно выполняться многократно и приводить систему в целевое состояние.
- Проверка результата как обязательный шаг: вывод объектов, фильтрация, экспорт в CSV/JSON.

Контрольные вопросы

1. Чем `Get-LocalUser` лучше `net user` с точки зрения дальнейшей автоматизации?
2. Что значит «идемпотентный скрипт» и как это реализовать при создании пользователя?
3. Как по событию в журнале безопасности (Event ID) понять, была ли попытка доступа успешной или нет?
4. Почему в серверной среде PowerShell предпочтительнее графических оснасток?

Задание 1. Автоматизированное создание пользователей и управление группами

Цель: написать идемпотентный скрипт, который создаёт пользователей и приводит членство в группах к целевому состоянию.

Целевое состояние:

- Пользователь `lab_admin`: в группе «Администраторы», включён.
- Пользователь `lab_user`: в группе `lab_developers`, включён.
- Группа `lab_developers` существует.

Скрипт (сохраните как `setup-users.ps1`):

powershell

```powershell
# setup-users.ps1
$adminName = "lab_admin"
$userName = "lab_user"
$devGroup = "lab_developers"
$strongPassword = ConvertTo-SecureString "Str0ngP@ssw0rd!2024" -AsPlainText -Force

# Создаём группу, если нет
if (-not (Get-LocalGroup -Name $devGroup -ErrorAction SilentlyContinue)) {
    New-LocalGroup -Name $devGroup -Description "Группа разработчиков для лабы"
}

# lab_admin: создаём или убеждаемся, что он в Администраторах
if (-not (Get-LocalUser -Name $adminName -ErrorAction SilentlyContinue)) {
    New-LocalUser -Name $adminName -Password $strongPassword -FullName "Lab Admin" -Description "Админ для лабы"
}
if (-not ((Get-LocalGroupMember -Group "Администраторы" | Select-Object -ExpandProperty Name) -contains $adminName)) {
    Add-LocalGroupMember -Group "Администраторы" -Member $adminName
}
Set-LocalUser -Name $adminName -Enabled $true

# lab_user: создаём и добавляем в lab_developers
if (-not (Get-LocalUser -Name $userName -ErrorAction SilentlyContinue)) {
    New-LocalUser -Name $userName -Password $strongPassword -FullName "Lab User" -Description "Пользователь для лабы"
}
if (-not ((Get-LocalGroupMember -Group $devGroup | Select-Object -ExpandProperty Name) -contains $userName)) {
    Add-LocalGroupMember -Group $devGroup -Member $userName
}
Set-LocalUser -Name $userName -Enabled $true

Write-Host "Целевое состояние пользователей и групп достигнуто." -ForegroundColor Green
```

Проверка (включите в отчёт):

powershell

```powershell
Get-LocalUser lab_admin, lab_user | Select-Object Name, Enabled
Get-LocalGroupMember "Администраторы" | Where-Object {$_.Name -like "*lab_admin*"}
```

```
Get-LocalGroupMember $devGroup
```

Что сдать: файл скрипта, скриншоты вывода проверок, краткий комментарий: «скрипт идемпотентен: повторный запуск не создаёт дубликатов и не ломает состояние».

В серверной среде такие скрипты запускаются многократно (CI/CD, конфигурация серверов, восстановление). Поэтому нельзя просто «создать пользователя» — нужно сначала проверить, существует ли он, и только потом действовать. Это и есть автоматизация с контролем состояния.

Задание 2. Автоматизация управления правами и аудита (40 минут)

Цель: автоматизировать настройку NTFS-прав и аудит, а также проверку результата.

Исходное состояние:

- Папка `C:\LabData` существует.
- `lab_admin` — полный доступ.
- `lab_developers` — изменение и чтение.
- «Пользователи» — явно запрещены (explicit deny) к этой папке.
- Для `lab_user` включён аудит создания, чтения, удаления файлов.

Скрипт (сохраните как `setup-permissions.ps1`):

powershell

```
# setup-permissions.ps1
$folder = "C:\LabData"
$adminName = "lab_admin"
$devGroup = "lab_developers"

if (-not (Test-Path $folder)) {
    New-Item -Path $folder -ItemType Directory -Force | Out-Null
}

$acl = Get-Acl -Path $folder

# Удаляем унаследованные права и отключаем наследование (Copy)
$acl.SetAccessRuleProtection($true, $false)

# lab_admin — FullControl
```

```
$ruleAdmin = New-Object
System.Security.AccessControl.FileSystemAccessRule($adminName, "FullControl",
"ContainerInherit,ObjectInherit", "None", "Allow")
$acl.AddAccessRule($ruleAdmin)

# lab_developers — Modify + ReadAndExecute
$ruleDev = New-Object
System.Security.AccessControl.FileSystemAccessRule($devGroup,
"Modify,ReadAndExecute", "ContainerInherit,ObjectInherit", "None", "Allow")
$acl.AddAccessRule($ruleDev)

# Пользователи — явный запрет (Deny)
$ruleDeny = New-Object
System.Security.AccessControl.FileSystemAccessRule("Пользователи",
"FullControl", "ContainerInherit,ObjectInherit", "None", "Deny")
$acl.AddAccessRule($ruleDeny)

Set-Acl -Path $folder -AclObject $acl

# Аудит: lab_user — Create, Read, Delete
$auditRule = New-Object
System.Security.AccessControl.FileSystemAuditRule($env:USERNAME,
"CreateFiles,ReadData,Delete", "Success", "ContainerInherit,ObjectInherit",
"None")
# Примечание: для полноценного аудита лучше использовать lab_user, но в
рамках лабы можно использовать текущего пользователя для демонстрации.
$aclAudit = Get-Acl $folder
$aclAudit.AddAuditRule($auditRule)
Set-Acl -Path $folder -AclObject $aclAudit

Write-Host "Права и аудит настроены." -ForegroundColor Green
```

Проверка:

powershell

```
(Get-Acl C:\LabData).Access | Select-Object IdentityReference,
FileSystemRights, AccessControlType
(Get-Acl C:\LabData).Audit | Select-Object IdentityReference, AuditFlags
```

Что сдать: скрипт, вывод проверок, скриншот свойств папки (вкладка «Безопасность») с отображением явного запрета и правил аудита.

NTFS-права в PowerShell управляются через объекты ACL/ACE. Это даёт точный контроль (вплоть до явных запретов и аудита), который невозможно получить только через графический интерфейс. В серверной среде это стандарт: права задаются скриптами и проверяются автоматически.

Задание 3. Автоматизация управления службами и мониторинг

Цель: научиться управлять службами и писать скрипты, которые не просто выполняют действие, а принимают решение на основе состояния.

Исходное состояние: служба «Диспетчер печати» (Spooler) должна быть запущена. Если она остановлена — запустить. Если уже запущена — вывести сообщение.

Скрипт (сохраните как `manage-spooler.ps1`):

```powershell
# manage-spooler.ps1
$serviceName = "Spooler"
$svc = Get-Service -Name $serviceName

if ($svc.Status -eq "Running") {
    Write-Host "Служба $serviceName уже работает." -ForegroundColor Green
} else {
    Start-Service -Name $serviceName
    Write-Host "Служба $serviceName запущена." -ForegroundColor Yellow
}

Get-Service -Name $serviceName | Select-Object Name, Status, StartType
```

Дополнительно (по желанию, для продвинутых): экспорт списка служб в CSV:

```powershell
Get-Service | Export-Csv -Path "C:\LabData\services.csv" -NoTypeInformation -Encoding UTF8
```

Что сдать: скрипт, скриншот вывода, CSV-файл.

Автоматизация — это не «выполнить команду», а «проверить состояние → принять решение → выполнить действие → подтвердить результат». Такой подход масштабируется на сотни серверов.

Задание 4. Автоматизированная диагностика и сбор событий

Цель: собрать данные для диагностики в виде структурированного вывода (CSV/JSON) и проанализировать их.

Действия:

1. Получить последние 10 событий из журнала «Система»:

```powershell
Get-WinEvent -LogName System -MaxEvents 10 | Select-Object TimeCreated, Id, LevelDisplayName, Message | Export-Csv C:\LabData\system-events.csv -NoTypeInformation -Encoding UTF8
```

2. Получить топ-5 процессов по CPU:

```powershell
Get-Process | Sort-Object CPU -Descending | Select-Object -First 5 Name, Id, CPU, WorkingSet | Export-Csv C:\LabData\top-cpu.csv -NoTypeInformation -Encoding UTF8
```

3. Проанализировать одно событие из CSV: что произошло, уровень (Error/Warning), насколько критично.

Что сдать: два CSV-файла, скриншот одного события с анализом (2–3 предложения).

В администрировании серверных ОС данные собираются скриптами и анализируются программно (SIEM, мониторинг, отчётность). CSV/JSON — это стандартный формат обмена данными между инструментами.

Структура отчёта:

- Цель и задачи.
- Ход работы: по каждому заданию — скрипт (фрагмент), вывод команд, скриншоты.
- Анализ: какие проверки сделаны, как обеспечена идемпотентность, какие данные экспортированы.
- Вывод: какие навыки автоматизации приобретены, как эти скрипты можно масштабировать на серверную среду.
- Ответы на конкретные вопросы.

Обобщения.

PowerShell — это не просто «консоль», а объектная среда. Поэтому в заданиях упор на объекты (`Get-Service`, `Get-Acl`, `Get-WinEvent`), конвейеры и экспорт в структурированные форматы.

- **Связь с серверными ОС:** эти же скрипты (с минимальными изменениями) работают в Windows Server, в том числе в Server Core, где нет графического интерфейса.
- **Безопасность:** пароли в скриптах — только для лабораторной работы. В реальной среде используют `Read-Host -AsSecureString`, хранилища секретов, управляемые учётные записи.