OverTheWire Bandit - My Learning Journey

About This Guide

This repository documents my personal experience solving the Bandit wargame challenges. Each level includes my approach, the challenges I faced, and what I learned.

Level 0: Getting Started

Challenge

Connect to the Bandit game server using SSH.

My Approach

```bash
ssh bandit0@bandit.labs.overthewire.org -p 2220
Password: bandit0
```

What I Learned

- Basic SSH connection syntax
- Using non-standard ports with `-p` flag
- Username format for remote connections

Notes

This level is about getting comfortable with SSH. Make sure you have an SSH client installed (built into

Linux/Mac, use PuTTY or Windows Terminal on Windows).

Level 0 → Level 1

Challenge

Find and read a file called `readme` in the home directory.

My Approach

```bash
ls          # List files in current directory
cat readme  # Display file contents
```

Password Location

The password is in the `readme` file in the home directory.

What I Learned

- `ls` - lists directory contents
- `cat` - concatenates and displays file contents
- Home directory is where you start when logging in

Key Takeaway

Always start by exploring your environment with `ls` to see what files are available.

Level 1 → Level 2

Challenge

Read a file with a special name: `-`

Problem I Encountered

Running `cat -` doesn't work because `-` is interpreted as stdin.

My Solution

```bash
cat ./-
# OR
cat < -
```

What I Learned

- Special characters in filenames need careful handling
- Using `./` prefix treats the dash as a filename
- The `.` represents current directory

Alternative Approaches

- `cat < -` (redirect from file)
- Use absolute path: `cat /home/bandit1/-`

Level 2 → Level 3

Challenge

Read a file named `spaces in this filename`

My Approach

```bash
# Method 1: Escaping spaces
cat spaces\ in\ this\ filename

# Method 2: Using quotes
cat "spaces in this filename"

# Method 3: Tab completion
cat spa[TAB]  # Terminal auto-completes with proper escaping
```

What I Learned

- Spaces in filenames must be escaped with backslash
- Quotes preserve the entire filename as one argument
- Tab completion automatically handles special characters

Best Practice

Use quotes when dealing with filenames containing spaces - it's cleaner and less error-prone.

Level 3 → Level 4

Challenge

Find a hidden file in the `inhere` directory.

My Approach

```bash
cd inhere
ls -a        # Show all files including hidden ones
cat .hidden  # Read the hidden file
```

What I Learned

- Files starting with `.` are hidden in Linux
- `ls -a` shows all files (including hidden)
- `ls -la` gives detailed listing of all files

Common Hidden Files

- `.bashrc` - bash configuration
- `.ssh` - SSH keys directory
- `.git` - Git repository data

Level 4 → Level 5

Challenge

Find the only human-readable file among many files in `inhere` directory.

My Approach

My Approach

bash

```bash
cd inhere
ls          # Shows files named -file00 through -file09
file ./*    # Check file types for all files

# Found that -file07 is ASCII text
cat ./-file07
```

What I Learned

- `file` command identifies file types
- "Human-readable" typically means ASCII text
- Wildcards (`*`) help check multiple files at once

Why Use `./`?

The `./` prefix is needed because filenames start with `-`, which would otherwise be interpreted as command options.

Level 5 → Level 6

Challenge

Find a file in `inhere` directory with specific properties:

- Human-readable
- 1033 bytes in size
- Not executable

- Not executable

My Approach

```
bash

cd inhere
find . -type f -size 1033c -readable ! -executable
# Result: ./maybehere07/.file2
cat ./maybehere07/.file2
```

What I Learned

- `find` is powerful for locating files by properties
- `-type f` specifies regular files
- `-size 1033c` means exactly 1033 bytes (c = bytes)
- `!` negates a condition (not executable)
- `-readable` checks if file is readable

Find Command Syntax

```
bash

find [path] [conditions]
```

Level 6 → Level 7

Challenge

Find a file somewhere on the entire server with:

- Owned by user bandit7
- Owned by group bandit6
- 33 bytes in size

My Approach

```bash
cd /  # Go to root directory to search entire server
find . -user bandit7 -group bandit6 -size 33c 2>/dev/null
# Result: ./var/lib/dpkg/info/bandit7.password
cat /var/lib/dpkg/info/bandit7.password
```

What I Learned

- Search from `/` (root) to scan entire system
- `2>/dev/null` redirects error messages (like "Permission denied")
- `-user` and `-group` filter by ownership
- File permissions can prevent access to most results

Error Redirection

- `2>` redirects stderr (error messages)
- `/dev/null` is a special file that discards all data

Level 7 → Level 8

Challenge

Find the password next to the word "millionth" in `data.txt`.

My Approach

```bash
grep millionth data.txt
```

What I Learned

- `grep` searches for patterns in files
- Format: `grep [pattern] [file]`
- grep returns entire lines containing the match

Useful grep Options

- `-i` - case insensitive search
- `-n` - show line numbers
- `-v` - invert match (show non-matching lines)
- `-c` - count matching lines

Level 8 → Level 9

Challenge

Find the only unique line in `data.txt` (appears exactly once).

My Approach

bash

```bash
sort data.txt | uniq -u
```

What I Learned

- `sort` arranges lines alphabetically
- `uniq` filters duplicate adjacent lines
- `uniq -u` shows only unique lines (appearing once)
- `|` (pipe) sends output from one command to another
- **Must sort before using uniq** (uniq only works on adjacent duplicates)

Command Pipeline Explained

1. `sort` groups identical lines together
2. `uniq -u` finds lines that appear only once

Level 9 → Level 10

Challenge

Find human-readable strings in a binary file, looking for ones starting with several `=` characters.

My Approach

```bash
strings data.txt | grep "==="
```

What I Learned

- `strings` extracts printable characters from binary files
- Binary files contain non-printable data
- Combining `strings` with `grep` filters results
- Multiple `=` signs help narrow down the search

Why strings?

Binary files can't be read with `cat`. The `strings` command extracts readable text sequences from any file type.

Level 10 → Level 11

Challenge

Decode base64 encoded data in `data.txt`.

My Approach

```bash
base64 -d data.txt
```

What I Learned

- Base64 is an encoding scheme (not encryption)

- `-d` flag decodes base64 data
- `-e` flag encodes to base64
- Base64 is commonly used for encoding binary data in text format

Base64 Basics

- Converts binary data to ASCII text
- Uses A-Z, a-z, 0-9, +, /
- Often ends with `=` padding

Level 11 → Level 12

Challenge

Decode ROT13 cipher in `data.txt`.

My Approach

```bash
cat data.txt | tr 'A-Za-z' 'N-ZA-Mn-za-m'
```

What I Learned

- ROT13 rotates each letter by 13 positions
- `tr` (translate) replaces characters
- ROT13 is its own inverse (applying it twice gives original text)

- Format: `tr [set1] [set2]`

ROT13 Explained

- A→N, B→O, C→P, ... M→Z
- N→A, O→B, P→C, ... Z→M
- Numbers and special characters unchanged

Level 12 → Level 13

Challenge

Reverse a hexdump and repeatedly decompress a file.

My Approach

```
bash
```

```
mkdir /tmp/mywork
cp data.txt /tmp/mywork
cd /tmp/mywork

# Reverse hexdump
xxd -r data.txt > data_bin

# Check file type and decompress accordingly
file data_bin  # Shows compression type

# Repeat: rename, decompress, check type
# Process: gzip → bzip2 → gzip → tar → tar → bzip2 → tar → gzip
```

Decompression Commands Used

```bash
# For gzip (.gz)
mv file file.gz
gunzip file.gz

# For bzip2 (.bz2)
mv file file.bz2
bunzip2 file.bz2

# For tar (.tar)
tar -xf file.tar
```

What I Learned

- `xxd -r` reverses a hexdump
- Files can be compressed multiple times

- Always check `file` command to know what to do next
- Different compression tools have different extensions
- Working in `/tmp` is safe for temporary files

Key Strategy

1. Check file type with `file`
2. Rename with appropriate extension
3. Decompress
4. Repeat until you get ASCII text

Level 13 → Level 14

Challenge

Use an SSH private key to log into the next level.

My Approach

```bash
ssh -i sshkey.private bandit14@localhost -p 2220
```

What I Learned

- `-i` specifies identity file (private key)
- Private keys enable passwordless authentication

- `localhost` refers to the same machine
- SSH keys are more secure than passwords

SSH Key Authentication

- Private key stays secret (like a password)
- Public key goes on servers
- Never share private keys

Level 14 → Level 15

Challenge

Submit the current level's password to port 30000 on localhost.

My Approach

```bash
# First get the password
cat /etc/bandit_pass/bandit14

# Connect and submit
nc localhost 30000
[paste password]
```

What I Learned

- `nc` (netcat) connects to network ports

- Can send data to specific ports
- Format: `nc [host] [port]`
- Password for level 14 is in `/etc/bandit_pass/bandit14`

Netcat Basics

- Swiss army knife of networking
- Can act as client or server
- Useful for testing network services

Level 15 → Level 16

Challenge

Submit password to port 30001 using SSL/TLS encryption.

My Approach

```bash
openssl s_client -connect localhost:30001
[paste password for bandit15]
```

What I Learned

- `openssl s_client` creates SSL/TLS connections
- SSL encrypts data transmission
- `-connect` specifies host:port

- Regular netcat doesn't support encryption

SSL/TLS Explained

- Secures communication between client and server
- Prevents eavesdropping
- Used by HTTPS websites

Level 16 → Level 17

Challenge

Find which port (31000-32000) speaks SSL and gives credentials.

My Approach

bash

```
# Scan for open ports
nmap -p 31000-32000 localhost

# Test SSL ports found
openssl s_client -connect localhost:31790
[paste password]
```

Handling the SSH Key

bash

```
# Save the returned private key
mkdir /tmp/mykeys
nano /tmp/mykeys/bandit17.key
[paste the private key]

# Set proper permissions
chmod 600 /tmp/mykeys/bandit17.key

# Use it to login
ssh -i /tmp/mykeys/bandit17.key bandit17@localhost -p 2220
```

What I Learned

- `nmap` scans for open ports
- `-p` specifies port range
- SSH keys need restrictive permissions (600)
- `chmod 600` makes file readable/writable only by owner

Port Scanning Ethics

- Only scan systems you have permission to test
- Port scanning can be detected
- Used by both security professionals and attackers

Level 17 → Level 18

Challenge

Find the difference between two password files.

My Approach

```bash
diff passwords.old passwords.new
```

What I Learned

- `diff` compares files line by line
- `<` indicates lines from first file
- `>` indicates lines from second file
- Changed line in passwords.new is the password

Diff Output Format

```
42c42
< old line
---
> new line
```

Level 18 → Level 19

Challenge

Login despite .bashrc logging you out immediately.

My Approach

My Approach

```bash
# Method 1: Execute command without shell
ssh bandit18@bandit.labs.overthewire.org -p 2220 cat readme

# Method 2: Use different shell
ssh bandit18@bandit.labs.overthewire.org -p 2220 -t /bin/sh
cat readme
```

What I Learned

- `.bashrc` runs automatically on login
- Can execute commands via SSH without interactive shell
- `-t` forces pseudo-terminal allocation
- Different shells available: bash, sh, zsh, etc.

Bypassing .bashrc

When `.bashrc` contains malicious commands, you can:

1. Run single command via SSH
2. Use alternative shell
3. Specify `--norc` flag (in some scenarios)

Level 19 → Level 20

Challenge

Use a setuid binary to read bandit20's password.

My Approach

```bash
ls -la  # Check file permissions
./bandit20-do cat /etc/bandit_pass/bandit20
```

What I Learned

- SetUID bit allows running programs as file owner
- Shown as `s` in permissions: `-rwsr-x---`
- Security risk if misused
- Used for programs needing elevated privileges

SetUID Explained

- Allows temporary privilege elevation
- User runs file as if they were the owner
- Common in system utilities (like `passwd`)

Level 20 → Level 21

Challenge

Create a listening port, then connect with the setuid binary.

My Approach

```bash
# Terminal 1: Create listener
echo "GbKksEFF4yrVs6il55v6gwY5aVje5f0j" | nc -l -p 12345 &

# Terminal 2: Connect with suconnect
./suconnect 12345
```

What I Learned

- `nc -l` creates a listening server
- `-p` specifies port number
- `&` runs command in background
- Can communicate between two terminals
- The binary reads from our listener and responds with next password

Netcat Listening

- `-l` means listen mode
- Port number should be > 1024 (non-privileged)
- Can pipe data directly to netcat

Level 21 → Level 22

Challenge

Examine a cron job to find the password.

My Approach

```bash
cd /etc/cron.d
ls
cat cronjob_bandit22
# Shows: @reboot and * * * * * both run a script

cat /usr/bin/cronjob_bandit22.sh
# Script copies password to /tmp file

cat /tmp/t7O6lds9S0RqQh9aMcz6ShpAoZKF7fgv
```

What I Learned

- Cron runs scheduled tasks automatically
- `@reboot` runs at system startup
- `* * * * *` means every minute
- Cron jobs are defined in `/etc/cron.d/`
- Scripts can be read even if we can't execute them

Cron Time Format

```
* * * * * command
| | | | |
| | | | └─ Day of week (0-7)
| | | └─── Month (1-12)
```

```
│ │ └───────── Day of month (1-31)
│ └────────── Hour (0-23)
└─────────── Minute (0-59)
```

Level 22 → Level 23

Challenge

Understand and exploit a cron job script.

My Approach

```bash
cat /etc/cron.d/cronjob_bandit23
cat /usr/bin/cronjob_bandit23.sh

# Script creates filename from: echo I am user $myname | md5sum | cut -d ' ' -f 1
# We need to run this AS bandit23

myname=bandit23
echo I am user $myname | md5sum | cut -d ' ' -f 1
# Output: 8ca319486bfbbc3663ea0fbe81326349

cat /tmp/8ca319486bfbbc3663ea0fbe81326349
```

What I Learned

- Can simulate script execution with different variables
- MD5 creates hash of input
- `cut` extracts specific fields

- `cut` extracts specific fields
- `-d ' '` sets delimiter as space
- `-f 1` takes first field

Script Analysis Strategy

1. Read and understand the script
2. Identify variables and their values
3. Simulate execution with target user's context
4. Find where output is stored

Level 23 → Level 24

Challenge

Create a script that gets executed by a cron job to retrieve the password.

My Approach

```
bash
```

```
# Create working directory
mkdir /tmp/mythings
cd /tmp/mythings

# Create script to copy password
cat > myscript.sh << EOF
#!/bin/bash
cat /etc/bandit_pass/bandit24 > /tmp/mythings/password
EOF

chmod 777 myscript.sh
chmod 777 /tmp/mythings

# Copy to cron directory
cp myscript.sh /var/spool/bandit24/

# Wait about a minute
sleep 60

# Read password
cat /tmp/mythings/password
```

What I Learned

- Scripts in `/var/spool/bandit24/` get executed by bandit24
- Need write permissions on output directory
- Cron jobs run with owner's privileges
- Scripts are deleted after execution
- `chmod 777` gives full permissions (read, write, execute for all)

Permission Numbers

- 4 = read (r)
- 2 = write (w)
- 1 = execute (x)
- 7 = 4+2+1 = rwx

Level 24 → Level 25

Challenge

Brute force a 4-digit PIN by connecting to port 30002.

My Approach

bash

```
mkdir /tmp/brute24
cd /tmp/brute24

# Create brute force script
cat > brute.sh << EOF
#!/bin/bash
password="UoMYTrfrBFHyQXmg6gzctqAwOmw1IohZ"

for pin in {0000..9999}; do
    echo "\$password \$pin"
done | nc localhost 30002 > result.txt
EOF

chmod +x brute.sh
./brute.sh

# Find the correct response
grep -v "Wrong" result.txt | grep -v "Please"
```

What I Learned

- Brute forcing tests all possibilities
- `{0000..9999}` generates all 4-digit combinations
- Piping multiple attempts to netcat at once
- `grep -v` excludes lines matching pattern
- 10,000 attempts (0000-9999)

Optimization Note

This script sends all attempts at once, which is faster than connecting 10,000 times separately.

Level 25 → Level 26

Challenge

Escape from a restricted shell that uses `more` command.

My Approach

```bash
# First, make terminal window VERY small (vertically)
ssh -i bandit26.sshkey bandit26@localhost -p 2220

# When 'more' is active (shows partial text):
# Press 'v' to enter vim

# In vim:
:set shell=/bin/bash
:shell

# Now you have a proper shell
cat /etc/bandit_pass/bandit26
```

What I Learned

- Can check user's shell: `getent passwd bandit26`
- `more` paginates output for large files
- `v` in `more` opens vim at current line
- Vim can execute shell commands

- `:set shell` changes vim's shell
- `:shell` spawns a shell from vim
- Small terminal forces `more` to paginate

More Command Tricks

- `v` - open in editor
- `q` - quit
- Space - next page
- Works only when content exceeds terminal height

Level 26 → Level 27

Challenge

Use the setuid binary after escaping the restricted shell.

My Approach

```bash
# After getting shell from previous level
./bandit27-do cat /etc/bandit_pass/bandit27
```

What I Learned

- Same setuid concept as level 19
- Must first escape restricted shell

- Then can use privileges of bandit27

Quick Reminder

This level combines skills from previous levels:

1. Escape restricted shell (Level 25)
2. Use setuid binary (Level 19)

Level 27 → Level 28

Challenge

Clone a git repository and find the password.

My Approach

```bash
mkdir /tmp/git27
cd /tmp/git27

git clone ssh://bandit27-git@localhost:2220/home/bandit27-git/repo
cd repo
cat README
```

What I Learned

- `git clone` copies a repository
- SSH can be used for git operations

- Format: `ssh://user@host:port/path`
- Password is same as current level's user

Git Clone Basics

- Creates local copy of repository
- Downloads all files and history
- Can clone via HTTPS, SSH, or local path

Level 28 → Level 29

Challenge

Find password in git history (it was censored in current version).

My Approach

```bash
```

```
mkdir /tmp/git28
cd /tmp/git28

git clone ssh://bandit28-git@localhost:2220/home/bandit28-git/repo
cd repo

cat README.md  # Shows censored password

# Check commit history
git log

# View older commit
git show [commit-hash-with-password]
# OR
git checkout [commit-hash]
cat README.md
```

What I Learned

- Git stores all historical versions
- `git log` shows commit history
- `git show` displays specific commit
- `git checkout` switches to previous commits
- Sensitive data in git history is still accessible

Important Security Lesson

Never commit passwords to git, even temporarily! They remain in history forever.

Level 29 → Level 30

Challenge

Find password in a different git branch.

My Approach

```bash
mkdir /tmp/git29
cd /tmp/git29

git clone ssh://bandit29-git@localhost:2220/home/bandit29-git/repo
cd repo

cat README.md  # Says "no passwords in production"

# List all branches
git branch -a

# Check dev branch
git checkout dev
cat README.md
```

What I Learned

- Git branches enable parallel development
- `git branch -a` shows all branches (including remote)
- `git checkout` switches branches
- Common branch names: master/main, dev, staging, production
- Sensitive data often in dev branches

- Sensitive data often in development branches

Git Branching

- Separate lines of development
- Changes isolated until merged
- Remote branches prefixed with `remotes/origin/`

Level 30 → Level 31

Challenge

Find password in git tags.

My Approach

```bash
```

```
mkdir /tmp/git30
cd /tmp/git30

git clone ssh://bandit30-git@localhost:2220/home/bandit30-git/repo
cd repo

# Check usual places (nothing useful)
git log
git branch -a

# Check tags
git tag
# Shows: secret

git show secret
```

What I Learned

- Git tags mark specific points in history
- Often used for version releases (v1.0, v2.0)
- `git tag` lists all tags
- `git show [tag]` displays tag contents
- Tags can contain messages

Tags vs Branches

- Tags: Fixed points (releases, milestones)
- Branches: Moving lines of development

Level 31 → Level 32

Challenge

Push a file to the remote git repository.

My Approach

bash

```bash
mkdir /tmp/git31
cd /tmp/git31

git clone ssh://bandit31-git@localhost:2220/home/bandit31-git/repo
cd repo

cat README.md  # Instructions to push key.txt

# Create required file
echo "May I come in?" > key.txt

# Check .gitignore
cat .gitignore  # Shows *.txt is ignored

# Remove or edit .gitignore
rm .gitignore

# Add, commit, push
git add key.txt
git commit -m "Add key"
git push origin master
```

What I Learned

- `.gitignore` prevents files from being tracked
- Must remove .gitignore rule to track ignored files
- `git add -f` can force-add ignored files
- `git push` uploads commits to remote
- Remote repository can execute hooks on push

Git Push Workflow

1. `git add` - stage changes
2. `git commit` - save changes locally
3. `git push` - upload to remote

Level 32 → Level 33

Challenge

Escape from an uppercase shell.

My Approach

```
bash
```

```
# After login, everything is converted to uppercase

# Try special shell variable
$0

# This spawns a new shell (sh)
# Now check who you are
whoami  # Shows bandit33

# Get password
cat /etc/bandit_pass/bandit33
```

What I Learned

- `$0` contains name of current shell
- Executing `$0` starts a new shell instance
- Shell variables aren't converted to uppercase
- SetUID bit gave us bandit33 privileges
- Creative thinking needed for escape challenges

Shell Variables

- `$0` - shell name
- `$1, $2...` - script arguments
- `$$` - current process ID
- `$?` - last command exit status

Level 33 → Level 34

Final Level

Congratulations! You've completed all the current Bandit levels.

What I Learned From This Journey

1. **Linux Command Line**: Proficiency with essential commands
2. **File Permissions**: Understanding ownership and permissions
3. **Networking**: Using tools like netcat and SSH
4. **Scripting**: Creating bash scripts to automate tasks
5. **Git**: Working with version control
6. **Problem Solving**: Breaking down complex challenges
7. **Security Concepts**: SetUID, cron jobs, privilege escalation

Skills Acquired

- File manipulation and searching
- Text processing with grep, awk, cut
- Compression and encoding
- Network connections
- Git version control
- Shell scripting
- Creative problem-solving

Next Steps

- Try other OverTheWire wargames (Natas, Leviathan, Krypton)
- Practice on CTF platforms (HackTheBox, TryHackMe)
- Learn more advanced Linux administration
- Study cybersecurity fundamentals

Tips for Success

1. **Read Man Pages**: `man [command]` is your friend
2. **Experiment**: Try different approaches
3. **Take Notes**: Document what you learn
4. **Be Patient**: Some levels are frustrating - that's normal
5. **Google Wisely**: Look for concepts, not direct answers
6. **Understand Why**: Don't just copy commands, understand them

Resources Used

- OverTheWire Bandit: https://overthewire.org/wargames/bandit/
- Linux man pages
- Bash documentation
- Git documentation

Acknowledgments

This walkthrough represents my personal learning journey through the Bandit wargame. Each challenge taught me valuable skills in Linux system administration and security.

Remember: The goal is to learn, not just to get passwords. Take time to understand each concept!