# Work & documentation notes of various wargames

Galen Rowell

July 24, 2020

# 1 Bandit

## 1.1 Levels

### 1.1.1 bandit0

**Password to enter:** *bandit0*
**Challenge:** Solved using the **ssh** command, which included use of flags to set user & port.

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

### 1.1.2 bandit1

**Password to enter:** *boJ9jbbUNNfktd78OOpsqOltutMc3MY1*
**Challenge:** Reading a file named '-', this was problematic due to many common shell commands using '-' to prefix an option or flag.

```
cat ./-
```

### 1.1.3 bandit2

**Password to enter:** *CV1DtqXWVFXTvM2F0k09SHz0YwRINYA9*
**Challenge:** With spaces in a filename, shell programs will interpret the input as several arguments (instead of one space-delimited string). This issue can be solved two ways.

```
cat 'spaced filename'
```

```
cat spaced\ filename
```

### 1.1.4 bandit3

**Password to enter:** *UmHadQclWmgdLOKQ3YNgjWxGoRMb5luK*
**Challenge:** The file is prepended by a '.', which causes it to be hidden from most views. The *-A* flag for **ls** will show all hidden files except '.' & '..', which are part of the directory itself.

```
ls -A1
```

### 1.1.5 bandit4

**Password to enter:** *pIwrPrtPN36QITSp3EQaw936yaFoFgAB*
**Challenge:** The file is hidden in one of ' /inhere/-file0,9'. They contain special characters that interfere with the terminal environment. The use of **file** aids as it lists the encoding of a given file.

```
file ./*
```

### 1.1.6 bandit5

**Password to enter:** *koReBOKuIDDepwhWk7jZC0RTdopnAYKh*
**Challenge:** The file is within one of many sub-folders, with human readable encoding and a file size of '1033' bytes. The use of **ls** with the recursive flag *-R*, combined with **grep** to select the file with the given size solves this problem.

```
ls -Al -R | grep --color -C 5 -e '1033'
```

### 1.1.7 bandit6

**Password to enter:** *DXjZPULLxYr17uwoI01bNLQbtFemEgo7*
**Challenge:** The file is somewhere on the server, so we should search recursively from the root of the drive. We are given the owner name, group name and size of the file, which we can plug into **find** to find the file.

```
find / -group bandit6 -size 33c 2>&1 | grep -v "Permission denied"
```

*Note: The use of a terminal redirect and **grep** remove the output of excessive file permission warnings*

### 1.1.8 bandit7

**Password to enter:** *HKBPTKQnIay4Fw76bEy8PVxKEDQRKTzs*
**Challenge:** This level is a simple grep search for the word *millionth* in a large keyword text file.

### 1.1.9 bandit8

**Password to enter:** *cvX2JJa4CFALtqS87jk27qwqGhBM9plV*
**Challenge:** The password is the only line that occurs once within an unordered text file.

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

*Note: The -u flag of **uniq** ensures only lines of 1 occurrence are printed*

### 1.1.10 bandit9

**Password to enter:** *UsvVyFSfZZWbi6wgC7dAFyFuR6jQQUhR*
**Challenge:** The given file is a binary encoded file, IE. it is not in plaintext or easy to read. **strings** will only print human-readable strings from a given input, and the use of **grep** will limit the output to a manageable size.

```
strings data.txt | grep -Ee [=]+
```

*Note: The [=]+ pattern of **grep** searches for one or more occurances of = in each line, EG. =, ==== or ========*

### 1.1.11 bandit10

**Password to enter:** *truKLdjsbJ5g7yyJ2X2R0o3a5HQJFuLk*
**Challenge:** The file is encoded in base 64, which can be encoded and decoded using the **base64** program.

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

### 1.1.12 bandit11

**Password to enter:** *IFukwKGsFW8MOq3IRFqrxE1hxTNEbUPR*
**Challenge:** The file is encoded in a ROT-13 cipher, meaning that all letters in the alphabet have been shifted 13 places. **tr** is a unix program which is used the **tr**anslate various sets of text.

```
cat data.txt | tr "n-za-mN-ZA-M" "a-zA-Z"
```

### 1.1.13 bandit12

**Password to enter:** *5Te8Y4drgCRfCx8ugdwuEX8KFC6k2EUu*
**Challenge:** The given file is a hexdump of a binary file, which is a compressed **gzip** file. The **gzip** file is itself compressed many times with **gzip, bzip2** & **tar**. One of the best ways to discover what encoding a file has is to run **file** on the given file, as well as visual inspection with **less**.

The methodology used to solve the level was to inspect the file encoding using **file**, find the appropriate decompression program, then repeat until the end result was the final plain-text.

**gzip** has a 'unix-pipe' program version named **zcat**.
**bzip2** has a 'unix-pipe' program version named **bzcat**.
**tar** acts like a 'unix-pipe' program with the arguments **tar xO**.
The use of these *unix-pipe* versions allow use to pipe the input through *std-in* and have the decompressed output sent to *std-out*.

to uncompress a hexdump

```
xxd -r data.txt a.bin
```

to test the file encoding/type from std-in

```
file -
```

the series of decompression required

```
xxd -r data.txt a.bin
zcat a.bin | bzcat | zcat | tar xO | tar xO | bzcat | tar xO | zcat
```

### 1.1.14 bandit13

**Password to enter:** *8ZjyCRiBWFYkneahHwxCv3wb2a1ORpYL*
**Challenge:** This is a small challenge regarding SSH keys, a private key to access the next level is given. **ssh**'s **-i** flag uses the given key to authenticate the connection.

```
ssh bandit14@localhost -i sshkey.private
```

### 1.1.15 bandit14

**Password to enter:** *4wcYUJFw0k0XLShlDzztnTBHiqxU3b3e*
**Challenge:** This passwords for the next level is retreieved by sending the password for the current level to port 30,000 of the machine *(IE. localhost)*. This was solved by using a dated, but universal, shell program called **n**et **c**at.

```
1 nc localhost 30000
2   4wcYUJFw0k0XLShlDzztnTBHiqxU3b3e
3   Correct!
4   BfMYroe26WYalil77FoDi9qh59eK5xNr
```

*Note: Line #2 was entered manually, lines #3-4 were a 'response' from port 30,000*

### 1.1.16 bandit15

**Password to enter:** *BfMYroe26WYalil77FoDi9qh59eK5xNr*
**Challenge:** This challenge covers the use of **openssl**, and it's broad uses as a cryptographic tool for certificate and key generation/management. **openssl** has many sub-commands, of particular note is the **s_c lient** sub-command.

```
cat bandit15 | openssl s_client -connect localhost:30001 -ign_eof
```

*Note: the flag '-ign_eof' is used to allow the piping from cat into the session, alternatively the text can be manually input.*

### 1.1.17 bandit16

**Password to enter:** *cluFn7wTiGryunymYOu4RcffSxQluehd*
**Challenge:** This challenge expands more upon basic TCP/IP technologies, and covers port scanning with the use of **nmap**. **nmap** is an extremely powerful and ubiquitous ip & port scanning tool. It can be used to detect open ports and what they are running to detecting the OS a given machine is running.

scan ports 31000 32000 of localhost and attempt to discover what they are running

```
nmap -sV -p 31000-32000 localhost
```

### 1.1.18 bandit17

**Password to enter:** *xLYVMN9WE5zQ5vHacb0sZEVqbrp7nBTn*
**Challenge:** This challenge is a simple use of **diff**, to find the password which is the one line of difference between the two files.

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

### 1.1.19 bandit18

**Password to enter:** *kfBf3eYk5BPBRzwjqutbbfE887SVc5Yd*
**Challenge:** The login shell environment for level 18 automatically logs the user out. This prevents users from entering in any commands but is solved by using a feature of **ssh**, in which a command can be specified (instead of invoking a login shell).

```
ssh [...various flags/options...] destination [command]
ssh bandit18@bandit.labs.overthewire.org -p 2220 /bin/sh
```

### 1.1.20 bandit19

**Password to enter:** *IueksS7Ubh8G3DCwVzrTd8rAVOwq3M5x*
**Challenge:** This covers the shell programs **setuid** & **setgid**.

According to Wikipedia: ***setuid*** *and* ***setgid*** *(short for "set user ID" and "set group ID") are Unix access rights flags that allow users to run an executable with the permissions of the executable's owner or group respectively and to change behaviour in directories. They are often used to allow users on a computer system to run programs with temporarily elevated privileges in order to perform a specific task*

In essence they allow a user to execute a file as if they were a different (possibly higher-privileged) user.

### 1.1.21 bandit20

**Password to enter:** *GbKksEFF4yrVs6il55v6gwY5aVje5f0j*
**Challenge:** This level covers LINUX jobs, which enable the pausing of an active process, sending it to the background and other various features. This allows us to do things like set a server running in the background and connect to it all from the same terminal. The main issue here is a self-imposed limitation of using one **ssh** connection, it would be far more trivial if we used multiple connections.

1. set up the server for the given executable to connect to, place it in the background
2. connect the executable to the server, place it in the background
3. bring the server into the **f**ore**g**round, send the password to our executable

```
1 nc -l -p 30101 &
2 ./suconnect 30101 &
3 fg 1
4   nc -l -b -p 30101
5   GbKksEFF4yrVs6il55v6gwY5aVje5f0j
6   Read: GbKksEFF4yrVs6il55v6gwY5aVje5f0j
7   Password matches, sending next password
8   gE269g2h3mw3pwgrj0Ha9Uoqen1c9DGr
```

*Note: Line #5 was entered by the user*
*Note:* ***nc*** *or* ***netcat*** *varies by system and has oddities which had to be worked around for the script to function*

### 1.1.22 bandit21

**Password to enter:** *gE269g2h3mw3pwgrj0Ha9Uoqen1c9DGr*
**Challenge:** The challenge requires simple inspection of a cron file, which publishes the password for the next level in an obscure */tmp/* file.

Cron is a system utility for scheduling repeated tasks & scripts to execute at given times. *'It typically automates system maintenance or administration—though its general-purpose nature makes it useful for things like downloading files from the Internet and downloading email..'*

### 1.1.23 bandit22

**Password to enter:** *Yk7owGAcWjwMVRwrTesJEwB7WVOiILLI*
**Challenge:** This level requires analysis into a small shell script, and it's file permissions. The script uses **md5sum** to generate the name of a file inside */tmp/* in which the password is placed. When run manually it copies the password for the current level, being a script owned by the user *bandit23* it will publish the password for that user intermittently.

### 1.1.24 bandit23

**Password to enter:** *jc1udXuA1tiHqjIsL8yaapX5XIAI6i0n*
**Challenge:** This level had several pitfalls and difficulties, it required analysis of a cron job/script (similar to *bandit22*), file permissions and shell *stdin & stdout* redirection. Similar to the previous level, a cron script calls another script every minute. This particular script executes each executable inside */var/spool/bandit24*, then deletes them. Notably, if the output of the script is a new file, the script itself must ensure it is readable to other users.

script.sh

```
#!/bin/sh
cat etc/bandit_pass/bandit24 > <absolute path to pass>
```

```
touch pass

chmod a+rx script.sh
chmod a+rw pass.txt

cp script.sh /var/spool/bandit24

cat pass.txt
```

*Note: Due to the 1 minute scheduling of the cron script, one must wait at the most a minute before the 'pass.txt' is updated*

### 1.1.25 bandit24

**Password to enter:** *UoMYTrfrBFHyQXmg6gzctqAwOmw1IohZ*
**Challenge:** This level required a brute-forcing attempt in order to receive the password for the next level. The daemon on port 30002 required the password of the current level and a four digit passcode. *A note on brute-forcing:* any operation that is repeated will add to the time complexity, sometimes by an extraordinary amount. While it may be easier to include the **n**et**c**at command inside the loop, doing so would greatly increase overall execution time.

one line for loops in bash

```
for i in {01..05}; do echo "$i"; done
```

you can pipe the output of a for loop

```
for i in {01..05}
do
  echo "$i"
done | nc localhost 30002 > output.txt 2>&1
```

using grep line numbers combined with bash math

```
$(( `grep -n 'Correct' $output | cut -d : -f 1` - 1 ))
#but you probably want to assign it to a variable
passcode=$(( `grep -n 'Correct' $output | cut -d : -f 1` - 1 ))
```

#### 1.1.26 bandit25

**Password to enter:** *uNG9O58gUE7snukf3bvZ0rxhtnjzSGzG*
**Challenge:** The user is provided a ssh private key to login to the the next user *bandit26*, and subsequently read the password. The challenge arises from the login shell used by *bandit26*.

The default shell can be modified by a user, common shells are **bash** (*/bin/bash*), **zsh** (*/usr/bin/zsh*) & **fish** (*/usr/local/bin/fish*).

All valid login shells installed on the system will be listed in */etc/shells*.

The login shell for *bandit26* is problematic, as it *almost* immediately exits.

login shell for bandit26 /usr/bin/showtext

```
1 #!/bin/sh
2
3 export TERM=linux
4
5 more ~/text.txt
6 exit 0
```

This login shell exits on line 6, providing an issue as we want an interactive shell. The use of **more** however, provides us with a small exploit.

**more** is used to view files within the terminal, and has several utilities built in. If the file is small and can be completely shown within the current terminal screen, then more will simply display it and immediately exit. This would cause the last line of the login shell to execute, and cancel the session.

The easiest work-around is to shrink the size of the terminal within the screen, ensuring the **more** cannot display *text.txt*, entering **more**s '*interactive*' display mode. From here we can enter **v** to enter the **Vim** editor.

From within **Vim**, we can set the shell to be opened (as opposed to using the default shell) and then open a proper shell.

How to start a specific shell from vim

```
:set shell=/bin/bash
:shell
```

#### 1.1.27 bandit26

**Password to enter:** *5czgV9L3Xx8JPOyRbXh6lQbmIOWvPT6Z*
**Challenge:** This level is trivial, as it combines the **SETUID** challenge of *bandit19* and the login shell challenge of *bandit25*. The password for the next level can be read with a small SETUID script.

#### 1.1.28 bandit27

**Password to enter:** *3ba3118a22e93127a4ed485be72ef5ea*
**Challenge:** This level is a simple *git clone* of a *repo*(sitory) hosted locally. The password is in plain-text within the repository.

```
git clone ssh://bandit27-git@localhost/home/bandit27-git/repo
```

### 1.1.29 bandit28

**Password to enter:** *0ef186ac70e04ea33b4c1853d2526fa2*
**Challenge:** This level followed the same format of cloning a git repository. There was a plain-text file with the credentials for the next level inside, but the password was censored.

There are several **git** commands to gather information about the current repo:

list all local & remote branches

```
git branch -a
```

show the commit history

```
git log
```

show your local status compared to the repository

```
git status
```

**git log** showed a recent commit was made to censor the password that was mistakenly committed. Usage of the following recovered the needed commit:

```
git reset --hard <commit number>
```

*Note: This resets the current index & working tree in a destructive manner and discards any unsaved changes. This is a non-issue as we aren't saving anything to the repo.* ***git revert*** *may be another appropriate option to look into*

### 1.1.30 bandit29

**Password to enter:** *bbc96594b4e001778eee9975372716b2*
**Challenge:** This level was completed in a related manner to the previous level. The password was in a separate branch named *dev*, where someone had committed the password.

change to a different branch

```
git checkout <branch name>
```

change to a **new** branch

```
git checkout -b <new branch name>
```

### 1.1.31 bandit30

**Password to enter:** *5b90576bedb2cc04c86a9e924ce42faf*
**Challenge:** This level involved git tagging. The tag is named *secret* and is not linked to any commit or object, it merely contains the password of the next level.

show all objects on the remote

```
git ls-remote
```

show all local tags

```
git tag
```

show more information about a given object

```
git show <object name>
```

### 1.1.32 bandit31

**Password to enter:** *47e603bb428404d265f59c42920d81e5*
**Challenge:** This level follows the same format of a git repository. The *README.md* file prompts us to push a file with a given text content to *remote/master*. When this is done the remote responds with the password.

my default way to add files to the index

```
git add -Av
```

However, there was no output from this command when it normally lists each file that it's added. We should check the **.gitignore** file.

contents of this specific .gitignore

```
*.txt
```

So we can force addition of the file with:

```
git add -vf key.txt
```

*Note: Another option could be to modify the .gitignore and include the file with !key.txt*

### 1.1.33 bandit32

**Password to enter:** *56a9bf19c63d650ce78e6ec0354ee45e*
**Challenge:** This level has a shell which translates all letters into uppercase before execution, the subshell also had the SETUID bit set, and executed under *bandit33*. We had to break out of this subshell, which provided difficulty as almost every command in Linux is lowercase.

Thankfully we could start a subshell, and while the subshell doesn't display any output we can copy the password for *bandit33* to a temporary file.

two different ways to launch a subshell with no letters

```
`$0`
$($0)
```

### 1.1.34 bandit33

**Password to enter:** *c9c3199ddf4121b10cf581a98d51caee*

This level has no challenge, it's the final level of the bandit wargame.

## Links & resources

1. The bandit wargame is run on a remote server, accessed by ssh. In order to write scripts to log what I executed and re-run/solve the level then a tool is needed to be able to feed the password during the handshake process. SSHpass is great for this: SSHPass tutorial
2. When scripting, it is often useful to have a temporary directory where files can be created & modified without the risk of littering such files about the filesystem. So a temporary directory (often in /tmp/) is useful, **mktemp** does this:

   move to the new temporary directory

   ```
   cd $(mktemp -d)
   ```

   store the new temporary directory path

   ```
   tmp_dir=$(mktemp -d)
   ```