

Very Basic Linux Exploits (20 points)

Levels 1 - 25:

- **Bandit1:** boJ9jbbUNNfktd78OOpsqOltutMc3MY1
 - Doing the command “cat readme” immediately provided the password for bandit1. This took a few seconds with Ash working on it.
- **Bandit2:** CV1DtqXWVFXTvM2F0k09SHz0YwRINYA9
 - Attempting the command “cat -” resulted in nothing happening, along with similar methods such as “vim -”. Some research into accessing non-standard filenames revealed that the full address of the file should be entered, not just the local name. “cat /home/bandit/-” provided the password for bandit2. This took 5 minutes with Ash working on it.
- **Bandit3:** UmHadQcIWmgdLOKQ3YNgjWxGoRMb5luK
 - Since this file has spaces in it, the target file was surrounded with speech-marks, and so “cat “spaces in this filename”” was able to immediately provide the password for bandit3. This took a few seconds with Ash working on it.
- **Bandit4:** plwrPrtPN36QITSp3EQaw936yaFoFgAB
 - First I switched to the “inhere” directory, then entered “ls -la” to provide a more verbose list of files present, which revealed the “.hidden” file present in the directory. Entering “cat ./hidden” provided the password. This took approximately a minute with Ash working on it.
- **Bandit5:** koReBOKuIDDepwhWk7jZC0RTdopnAYKh
 - First I switched to the “inhere” directory, then kept running “cat ./file0x” where x was the number at the end of the file, from 1 to 7 which eventually provided a legible output. This took approximately a minute with Ash working on it.
- **Bandit6:** DXjZPULLxYr17uwoI01bNLQbtFemEgo7
 - Looking at the documentation for the “find” command revealed that a “-size” parameter can be passed and that if the number provided had a “c” after it, then that specified how many characters the file was. Since a character is a byte, I entered “find inhere/ -size 1033c” and that provided one file with the address “inhere/maybehere07/.file2”. Entering “cat inhere/maybehere07/.file2” provided the password. This took about a few minutes with Ash working on it.
- **Bandit7:** HKBPTKQnlay4Fw76bEy8PVxKEDQRKTzs
 - A similar approach was taken for this problem, where more parameters were provided, and I moved to the root directory in order to search the entire server. First I moved to the root directory with “cd ../..”. Then I entered “find -size 33c -user bandit7 -group bandit6”, and the only result which didn’t give a permission error was ./var/lib/dpkg/info/bandit7.password, so I did the “cat” command on that file address and that provided the password. This took about a minute with Ash working on it.
- **Bandit8:** cvX2JJJa4CFALtqS87jk27qwqGhBM9pIV

- I used the `grep` command with a wildcard to find this password. The command `"grep "millionth*" data.txt"` provided the line with the password immediately. This took about a minute with Ash working on it.
- **Bandit9:** UsvVyFSfZZWbi6wgC7dAFyFuR6jQQUhr
 - I first used the `"uniq -u"` command suggested on the problem page, however running this on the file provided hundreds of passwords. Referencing the documentation for the `"uniq"` command then revealed that it revealed the unique elements in a list provided the repeated lines were consequent. As a result, `"sort data.txt | uniq -u"` returned the desired password. This took about 5 minutes with Ash working on it.
- **Bandit10:** truKLdjsbJ5g7yyJ2X2R0o3a5HqJfULk
 - Looking into the documentation for the `"strings"` command suggested seemed to indicate it was suited for this challenge, since it converted characters into human-readable strings, and so a stab in the dark command was entered: `"strings data.txt | grep "===*"`. This worked. This took Ash a few minutes.
- **Bandit11:** IFukwKGsFW8MOq3IRFqrxE1hxTNEbUPR
 - The documentation for the `base64` command stated the `"-d"` option decoded base64 encoded text. So, the command `"base64 -d data.txt"` was tried and returned the password. This took Ash about a minute.
- **Bandit12:** 5Te8Y4drGCRfCx8ugdwuEX8KFC6k2EUu
 - The text from `data.txt` was copied into an online caesar cipher decoder, with a shifting of 13 provided. This revealed the password, and took Ash a few minutes to solve.
- **Bandit13:** 8ZjyCRiBWFYkneahHwxCV3wb2a1ORpYL
 - In order to undo the hexdump the command `"xxd -r data.txt > tmp/tmpdir/undump"` where `tmpdir` was the directory created by `"mktemp -d"`. Once I did that, I navigated to the directory, and then ran `"file undump"` to see which kind of compression had been run on the file. I then repeatedly decompressed it and checked the file type until it registered as an ASCII plaintext file, which contained the password. This took Ash about 15 minutes.
- **Bandit14:** 4wcYUJFw0k0XLShIDzztnTBHqXU3b3e
 - Looking into the documentation for the `"ssh"` command, the `"-i"` command provides a way to include a private key. As a result, the command `"ssh bandit14@localhost -i sshkey.private"` successfully logged me into bandit14. From there, `"cat /etc/bandit_pass/bandit14"` revealed the password. This took Ash about 15 minutes.
- **Bandit15:** BfMYroe26WYalil77FoDi9qh59eK5xNr
 - After looking at the documentation for the suggested command `"nc"`, the command `"cat /etc/bandit_pass/bandit14 | nc localhost 30000"` was attempted. This worked, and took Ash about 5 minutes.
- **Bandit16:** cluFn7wTiGryunymYOu4RcfSxQluehd
 - This required looking into the docs for `openssl` and `s_client`, which were both suggested commands for this level. After some research, the command `"openssl`

s_client localhost:30001” was tried, which showed all the security information for the ssl connection, and then I pasted the current password into the blank area provided and it returned the password. This took Ash about 20 minutes.

- **Bandit17:** xLYVMN9WE5zQ5vHacb0sZEVqbrp7nBTn
 - Looking into the docs for the “nmap” command showed that it accepted a range of port values that you may want to search through, so I entered “nmap localhost -p 31000-32000” which showed that there were 2 ports listening, and only one labelled as open. I attempted the same process as the one listed in bandit16 with the listed port, 31790, with success. This took Ash about 10 minutes.
- **Bandit18:** kFbF3eYk5BPBRzwtbbfE887SVc5Yd
 - After looking into the docs for the suggested command “diff”, I attempted running “diff passwords.old passwords.new”. This yielded that one password had been substituted for another in the new password file, so that was the solution.
- **Bandit19:** lueksS7Ubh8G3DCwVzrTd8rAVOwq3M5x
 - I tried numerous times to Ctrl+x in order to pause the .bashrc file, with no success. I also looked into different ways of trying to disable the .bashrc file from executing, which I also stopped pursuing as it seemed like an invalid way to go about the problem. I then turned to address the ssh component of the issue, and tried instead to ssh from the previous bandit user, which also didn’t work. When looking for a way to hide from the system that an ssh connection was being used, I realised after looking at the docs that inline commands could be passed from the ssh connection attempt. As a result, I attempted “ssh bandit18@bandit.labs.overthewire.org -p 2220 cat readme” when connecting, and then entered the connection password and that printed the next password to the screen before disconnecting me. This took about 45 minutes with Ash working on it.
- **Bandit20:** GbKksEFF4yrVs6il55v6gwY5aVje5f0j
 - Running “./bandit20-do” printed “Run a command as another user”. Using this information, I ran “./bandit20-do cat /etc/bandit_pass/bandit20”. This provided the password for the next level. This took Ash about 1 minute.
- **Bandit21:** gE269g2h3mw3pwgrj0Ha9Uoqen1c9DGr
 - First I ran “nmap localhost -p-” to see which ports were being currently used. Then I incorrectly kept running “./suconnect 30003 &”, and then “nc localhost -p 30003”. It took a good while of me repeatedly trying small variations on that command before I realised I was doing the connection in the wrong order, and so instead entering “cat /etc/bandit_pass/bandit20 | nc -l localhost -p 30003 &”, followed by “./suconnect 30003” which ended up returning the right password. This took Ash about 45 minutes.
- **Bandit22:** Yk7owGAcWjwMVRwrTesJEwB7WVOiLLI
 - First I entered “cd /etc/cron.d/” to see what cron jobs were available. This revealed a file by the name of “cronjob_bandit22”, which I opened to see ran a script called “/usr/bin/cronjob_bandit22.sh”. I opened this script as well, showing

that bandit22 regularly saves the password to a tmp file. Opening the listed file provided the password. This took Ash about a minute

- **Bandit23:** jc1udXuA1tiHqjlsL8yaapX5XIAI6i0n
 - Repeating the same process as above to find the bandit23 cron shell script, it can be seen that the script takes the current user's name (which would be bandit23 when running), and processes it to generate a tmp file. By manually carrying out the command in the script with the bandit23 username, the tmp file where the password is saved can be generated. Opening the file provides the password. This took Ash about 5 minutes
- **Bandit24:** UoMYTrfrBFHyQXmg6gzctqAwOmw1lohZ
 - First I created a tmp directory using "mktemp -d" to create my script, since the cron script deleted the shell script after executing it. Then, I created the shell script with "touch cronny.sh". In order to make sure everyone has permission to execute it, I also do a chmod 777 on the file and the directory. I then wrote the actual script in vim, where my initial attempt had the script try to cat the password out to the terminal, before realising that the output would only happen in bandit24, not bandit23. After this, I instead had the script cat the password to a file in the tmp directory. In order to execute the script, I executed the command "cp cronny.sh /var/spool/bandit24", waited a minute, then read the password. This took Ash about 20 minutes to complete.
- **Bandit25:** uNG9O58gUE7snukf3bvZ0rxhtnjzSGzG
 - For this challenge, I tried connecting to the daemon manually to see how it worked, and saw that it required the level password and a 4-digit pin separated by a space. So, I tried to have a for-loop iterate through pins {0000..9999}, using the command "echo "\$level_pass \$pin" | nc localhost 30002". However, this method was highly inefficient, since it waited for the connection to timeout, which took 20 seconds per attempt, which was not practical. I then tried to manually decrease the timeout time for the connection, however even that could only be brought down to 1 second using the "-w 1" option for netcat. This would result in the code requiring a little under 3 hours to run at most, which was also considered a poor solution. After a lot of playing around with different ways of feeding the pins to the connection, the final solution was achieved by creating a text file with the script with every pin combination, and then running the command "nc localhost 30002 < combinations.txt". This took Ash about 4 hours to solve
- **Bandit26:** 5czgV9L3Xx8JPOyRbXh6lQbmIOWvPT6Z
 - I began by trying to log into bandit26 using the sshkey, only to be logged out immediately. At first I tried to get more information by providing the "-v" option to the ssh connection, to show a more verbose output and understand what was happening. This however proved useless as it didn't state what script was running instead of bash. After some research online, I was able to discover that the command "cat /etc/shells" displays valid login shells. Running this command showed an option that seemed like it was our target, /usr/bin/showtext. Performing cat on this file confirmed those suspicions since it showed that this

```
bandit25@bandit:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/screen
/usr/bin/tmux
/usr/bin/showtext
bandit25@bandit:~$ cat /usr/bin/showtext
#!/bin/sh

export TERM=linux

more ~/text.txt
exit 0
bandit25@bandit:~$
```

```
--More-- (50%)
```

A screenshot of a terminal window with a black background. The window title bar at the top shows standard macOS window controls (red, yellow, and green buttons) and the text "Terminal". Inside the terminal, a 5x10 grid of ASCII art characters is displayed, forming the word "HELLO". The characters are white and use a variety of symbols including pipes, parentheses, backslashes, and forward slashes to create a pixelated effect. The grid is as follows:

At the bottom right of the terminal window, the text "1,3" and "All" are visible, likely indicating the current line and column positions.

From here, we can now reference the docs for “vi” to see what options we have for navigating. One such option that catches the eye is the `:r` command, which

allows you to read in a file. Executing “:r /etc/bandit_pass/bandit26” then pastes the contents of that file into this one:



As can be seen, the password is now visible and exploited. This took Ash about 2 ½ hours to complete.

Go Further! (Very Basic Linux Exploits) (8 points)

- **Bandit27:** 3ba3118a22e93127a4ed485be72ef5ea
 - This one required me to revise how I went about solving bandit26, since it required that I now have the shell actually function in order for me to get the password for the next level. I repeated the steps as above to get to the vim editor the same, but then decided to go to the official docs as opposed to the first “cheat sheet” pages that came up, and read through the docs at <http://vimdoc.sourceforge.net/html/doc/vimindex.html>. Looking through here showed a vast number of commands available, including “:shell”. However, running this just returned me to “more”. So I ran the command “:set shell=/bin/bash”, which reassigned the shell that it tried to connect to, and so running “:shell” again connected me to the correct shell, and from there I ran the command “./bandit27-do cat /etc/bandit_pass/bandit27” to get the password. This took 1 hour with Ash working on it.
- **Bandit28:** 0ef186ac70e04ea33b4c1853d2526fa2
 - For this level, I began by creating a tmp directory with the command “mktemp -d”. Then, I ran the command “git clone ssh://bandit27-git@localhost/home/bandit27-git/repo” from the tmp directory, which copied all the files over from the git. Included with the repo was a README file, which I opened with cat to reveal the password. This took about a minute with Ash working on it.
- **Bandit29:** bbc96594b4e001778eee9975372716b2
 - Retrying the steps above to find the password do not work for this level, since the password section of the readme now reads “xxxxxxxxxx”. Referring to a git command sheet showed one potential command that would be useful, “git log”. Looking up the documentation for the command, reveals the option “-p” to include changes that have been made, so I ran “git log -p” which showed the password before it was removed. This took Ash about 10 minutes

- **Bandit30:** 5b90576bedb2cc04c86a9e924ce42faf
 - Once again, trying the same trick from the previous level doesn't quite work, since the place that usually listed the password now says "<no passwords in production!>". The only place the password could now be hiding, is in another branch of the repo. Running "git branch -a" tells git to list every branch there is, which reveals a remote branch called "dev". Once I checkout to the dev branch, re-running "git log -p" shows the password we were looking for. This took Ash about 20 minutes.
- **Bandit31:** 47e603bb428404d265f59c42920d81e5
 - Trying both "git log -p" and "git branch -a" turn up nothing, so again we look to other commands to try and see what we can get. This time, I tried the command "git tag", which revealed a "secret" tag. Trying to see what this secret was with "git show secret" showed the password. This took Ash about 15 minutes.
- **Bandit32:** 56a9bf19c63d650ce78e6ec0354ee45e
 - To no surprise, none of the previous methods work for this level, so we revert back to "cat README.md". The file seems to suggest that I need to push a file in order to progress, so I enter "vim key.txt" and entered "May I come in?" like the README described. Then, running "git add key.txt" is met with a message saying that a .gitignore file is causing our file not to be uploaded. Running "cat *.gitignore" shows "*.txt", which indicates that our .gitignore is blocking all .txt files. Removing this file and then trying again shows the password in the validation script. This took about 30 minutes with Ash working on it
- **Bandit33:** c9c3199ddf4121b10cf581a98d51caee
 - This level has the shell convert everything into uppercase, rendering most commands useless. However, it does show that the "sh" shell is being run, so I revert back to a standard shell and run "man sh" and browse the documents to see what non-text commands I can run. When I reach the "Special Parameters" section of the docs, I see the option for "\$", which "Expands to the process ID of the invoked shell. A subshell retains the same value of \$ as its parent". Running this with "0" enters a standard dash shell, which allows me to cat the password for the next level.

PortSwigger Specific Attacks (10 points)

5 SQL Labs:

- 1.) **SQL injection vulnerability in WHERE clause allowing retrieval of hidden data:**
 - a.) In this lab you see that you can modify html attributes which call upon the sql database and change the requirements of the query by using a comment. This comments out the hidden codes and allows to show non released products from the database
- 2.) **SQL injection vulnerability allowing login bypass:**

- a.) In this lab you see that by injecting the name into the username with the comment at the end by passes the password security checks letting anyone sign in.

3.) SQL injection UNION attack, determining the number of columns returned by the query

- a.) In this lab we see that since we need the number of columns to perform a union attack that we have to find the number of columns. We do this by using Union Select Null, Null --. If the number of rows doesn't match the number of NULLs you have got the size + 1.

4.) SQL injection UNION attack, finding a column containing text:

- a.) In this lab since we have figured out the number of columns we can now search columns for strings. So if we search each column for a string it will be returned when it is in the column. Ex: ' UNION SELECT NULL, '00Jnlh', NULL -- was the answer cause the string was found in there

5.) SQL injection UNION attack, retrieving data from other tables:

- a.) In this lab you see that you can add union finds to existing queries to extract information from other databases such as username and passwords. Then you can use this information to log in.

SQL Attacks took around 1 hour. The toughest part was figuring out where to exploit the querying which turned out to be in the href references to the database itself in the search tab. In there you could directly modify the query results to get other info.

Screenshots:

The image displays two screenshots from a web security lab. The left screenshot shows the 'WEB SECURITY ACADEMY' interface with a lab titled 'SQL injection vulnerability in WHERE clause allowing retrieval of hidden data'. It indicates the lab is 'Solved' and shows a search bar with the text 'Accessories' OR 1=1--'. Below the search bar are filters for 'All', 'Accessories', 'Clothing, shoes and accessories', 'Corporate gifts', and 'Food & Drink'. The right screenshot shows a browser's developer console with the 'Elements' panel open, displaying the HTML structure of the search results. The search filters section is highlighted, showing the href attribute for the 'Accessories' filter: `Accessories == $0`. The console also shows the 'Console' panel with the message 'What's New'.



SQL injection vulnerability allowing login bypass

LAB

Solved



[Back to lab description >>](#)

Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning >>](#)

[Home](#) | [Hello, administrator!](#) | [Log out](#)

WE LIKE TO SHOP



SQL injection UNION attack, determining the number of columns returned by the query

LAB Solved



[Back to lab description >>](#)

Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning >>](#)

[Home](#) | [Account login](#)

WE LIKE TO SHOP

Accessories' UNION SELECT NULL,NULL,NULL--

Refine your search:

[All](#) [Accessories](#) [Corporate gifts](#) [Food & Drink](#) [Pets](#) [Toys & Games](#)

```
Elements Console Sources Network Performance Memory >>
<!doctype html>
<html>
  <head>...</head>
  <body>
    <div theme="ecommerce">
      <script src="/resources/js/labHeader.js"></script>
      <div id="labHeader">...</div>
      <section class="maincontainer">
        <div class="container is-page">
          <header class="navigation-header">...</header>
          <section class="ecommerce-pageheader">...</section>
          <section class="ecommerce-pageheader">...</section>
          <section class="search-filters">
            <label>Refine your search:</label>
            <a href="#">All</a>
            <a href="/filter?category=Accessories">Accessories</a> == $0
            <a href="/filter?category=Corporate+gifts">Corporate gifts</a>
            <a href="/filter?category=Food+%26+Drink">Food & Drink</a>
            <a href="/filter?category=Pets">Pets</a>
            <a href="/filter?category=Toys+%26+Games">Toys & Games</a>
          </section>
          <table class="is-table-numbers">...</table>
        </div>
      </section>
    </div>
  </body>
</html>
```

html body div section.maincontainer div.container.is-page section.search-filters a

Styles Event Listeners DOM Breakpoints Properties Accessibility

Filter :hov .cls +

Console What's New

top Filter Default levels



SQL injection UNION attack, finding a column containing text

[Back to lab description >>](#)

LAB Solved



Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning >>](#)

[Home](#) | [Account login](#)

WE LIKE TO
SHOP 

Accessories' UNION SELECT NULL,'00JnIh', NULL --

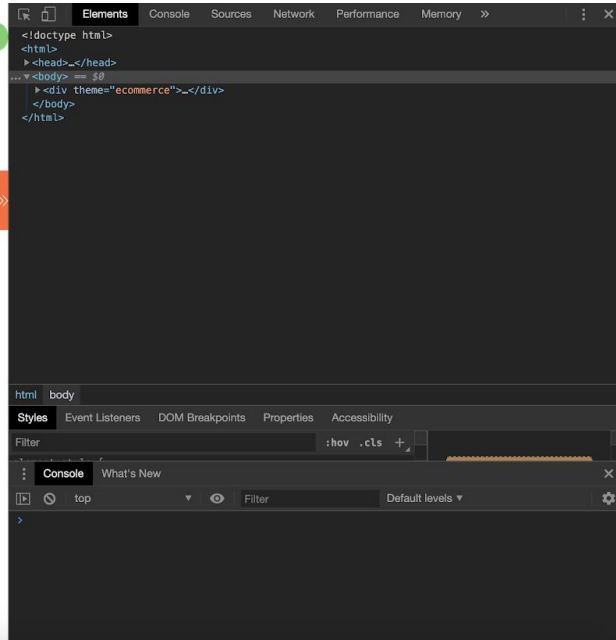
Refine your search:

[All](#) [Accessories](#) [Corporate gifts](#) [Lifestyle](#) [Tech gifts](#) [Toys & Games](#)

Six Pack Beer Belt

\$78.21

[View details](#)





SQL injection UNION attack, retrieving data from other tables

LAB

Solved



[Back to lab description >>](#)

Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning >](#)

[Home](#) | [Hello, administrator!](#) | [Log out](#)

WE LIKE TO
SHOP 

Refine your search:

[All](#)

[Accessories](#)

[Food & Drink](#)

[Gifts](#)

[Lifestyle](#)

[Pets](#)

Cheshire Cat Grin

We've all been there, found ourselves in a situation where we find it hard to look interested in

5 XSS Attacks:

1.) Reflected XSS into HTML context with nothing encoded:

- a.) For this lab we can see that you can execute scripts in HTML contexts by placing the script into the search box. The script is executed when doing this. Solution
`<script> Alert('Hello') </script>`

2.) Stored XSS into HTML context with nothing encoded

- a.) For this lab it was very similar to execution number one. We submit the `<script> Alert('Hello') </script>` into the comments to be stored and executed whenever someone loads that particular comment.

3.) DOM XSS in document.write sink using source location.search

- a.) For this exercise I ended up having to use the answer. However it makes a lot of sense. If we understand which element is being updated by an input or search result we can modify the element directly from the search itself. By adding `"><svg onload=alert(1)>` it completes the html element as well as running the alert command when the image loads.

4.) DOM XSS in innerHTML sink using source location.search

- a.) For this lab you can see that by searching and html tagged element with a error such as `` when it doesnt find the image source it will trigger the alert error. This could be used to call or modify things within a webpage other than just running a alert.

5.) Reflected XSS into attribute with angle brackets HTML-encoded

- a.) For this lab you had to inject alerts into the search. When the search is conducted the html takes what's inside the search box and queries it via the url.

The Xss attacks took around 1 hour as well. Things that worked were first really trying to understand where the search results were going before developing a plan on how the injected code would work.



Reflected XSS into HTML context with nothing encoded

[Back to lab description »](#)

LAB Solved

Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning »](#)

[Home](#)

0 search results for "

Search the blog...

Search

[< Back to Blog](#)



Stored XSS into HTML context with nothing encoded

[Back to lab description »](#)

LAB Solved

Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning »](#)

[Home](#)

Thank you for your comment!

Your comment has been submitted.

[< Back to blog](#)



DOM XSS in document.write sink using source location.search

LAB Solved

[Back to lab description >>](#)

Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning >>](#)

[Home](#)

0 search results for "'><svg onload=alert(1)>'

Search

[< Back to Blog](#)

```
Elements Console Sources Network Performance Memory >>
<!doctype html>
<html>
  <head></head>
  <body>
    <div theme="blog">
      <script src="/resources/js/labHeader.js"></script>
      <div id="labHeader"></div>
      <section class="maincontainer">
        <div class="container is-page">
          <header class="navigation-header"></header>
          <section class="blog-header"></section>
          <section class="search"></section>
          <script></script>
          </svg>
          <div class="is-linkback"></div>
        </div>
      </section>
    </div>
  </body>
</html>

html body div section.maincontainer div.container.is-page img
Styles Event Listeners DOM Breakpoints Properties Accessibility
Filter :hov .cls +
Console What's New
top Filter Default levels 1 hidden
```



DOM XSS in innerHTML sink using source location.search

LAB Solved

[Back to lab description >>](#)

Congratulations, you solved the lab!

[Share your skills!](#)

[Continue learning >>](#)

[Home](#)

0 search results for '<img'>

Search

[< Back to Blog](#)

Congratulations, you solved the lab!

[Share your skills!](#)
[Continue learning >>](#)
[Home](#)

0 search results for "'onmouseover='alert(1)'"

[< Back to Blog](#)

A Real Challenge:

- **Ash's ID:** fd219804-8e34-4cfa-ad97-ee30939d47f3
- **Jason's ID:** 3d01ef79-9144-485f-83db-841ea76da772

If we just type into the login box just a single quote, we can see a SQL query returned back. Since now we know that strings are not properly escaped in the password box, we can assume that we can end any string and also add any SQL code that we want to be executed after that.

If we then put the following code into the box and submit:

' or '1'='1

The SQL query that the server runs will return a list of usernames and they're accompanying passwords.

Input Validation: SQL Injection (50pts)

Figure out the password to login.

Get the password for user: zoidberg

admin,Gu3ss_Myp4s%w0rd**

bender,b1t3-my-shiny-m3t4l-4\$\$

fry,w4ts-w/-th3-17-dungbeetles

farnsworth,P4zuzu!!

```
scruffy,lm_0n-br3ak
zoidberg,sp4r3-ch4ng3#$$$
Enter the login password.
```

We can see from our SQL query that the password for username “zoidberg” is in the last line.

Figure out the password to login.

SOLVED!!

```
SUBMIT
'SELECT username, password FROM users WHERE username='zoidberg' AND
password="" - unrecognized token: ""
```

Cross Site Scripting (75pts):

Starting the assignment was really just a search effort through the Inspect Element page. After about 20 minutes we found that the flags were stored within a cookie and so from there, we just created a basic XSS payload to alert us of the cookie.

By looking through our cookies, we can see that the flags are stored in “admin_sess_id”, and by using a basic XSS payload below to alert us the cookie, we are able to retrieve the flag below:

```
<img src=http://what/file.dunno"onerror=alert(document.cookie);>
```

```
admin_sess_id=flag{James/T.*Kirk=+ermin@l}
```

Submit Score

Application

Manifest

Service Workers

Clear storage

Storage

Local Storage

Session Storage

IndexedDB

Web SQL

Cookies

https://hack.ainfosec.co

Cache

Cache Storage

Application Cache

Background Services

Background Fetch

Background Sync

Notifications

Payment Handler

Periodic Background Sync

Push Messaging

Filter

Only blocked

Name	Value	D...	P...	E...	S...	H...	S...	S...	P...
csrftoken	wLN06yN6cemsm...	h...	/	2...	73		✓	L...	M...
sessionid	db5a01j7cyljbe3w...	h...	/	2...	41	✓	✓	L...	M...
admin_sess_...	flag_should_be_he...	h...	/	S...	47				M...

flag_should_be_here%20%F0%9F%A4%94

Console

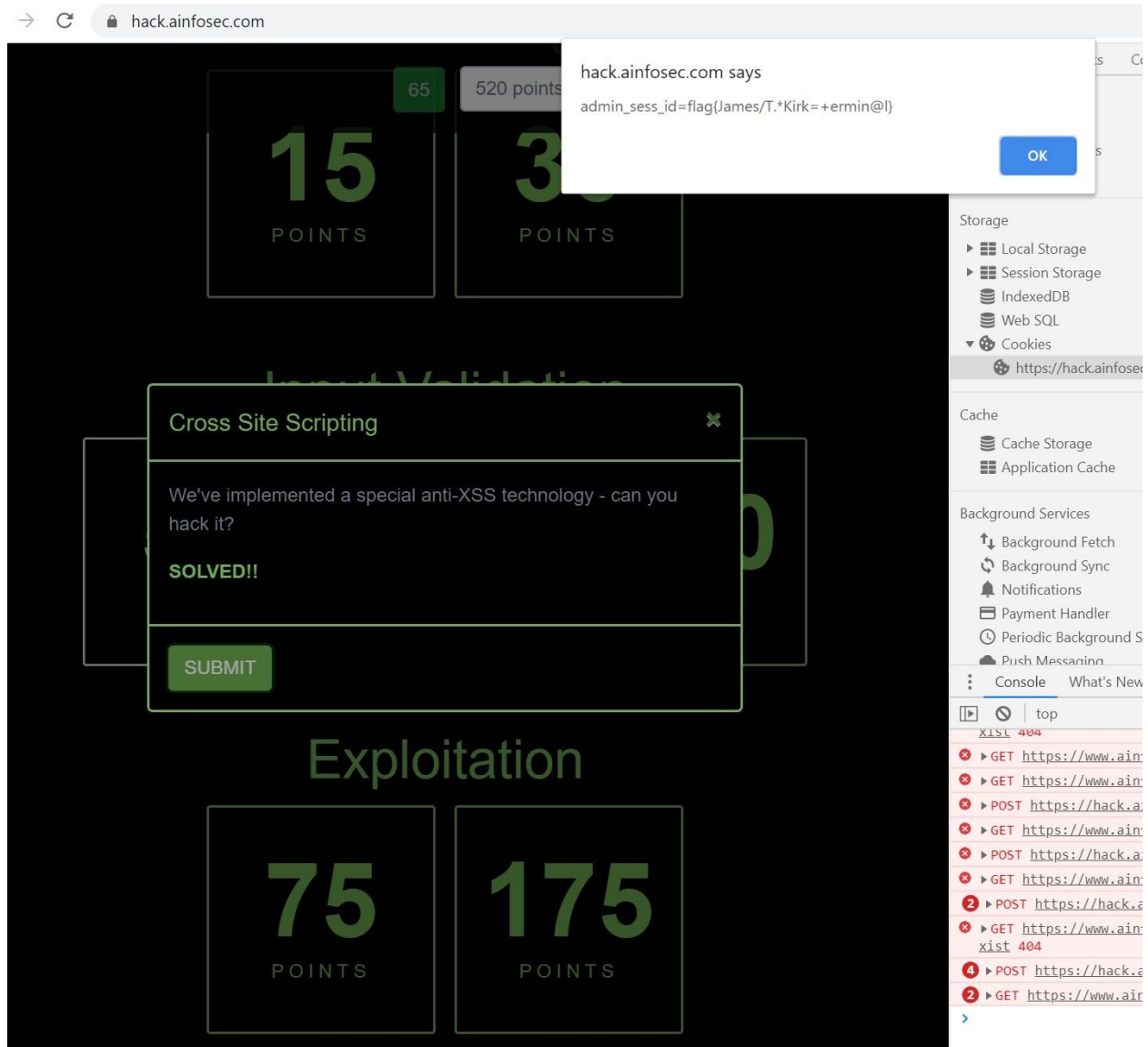
What's New

top

Filter

Default levels

If you get something other than a 200 as a response then you didn't solve it...



SQL Credit Cards (100pts):

Basically we started with running a single quote in the text box and this gave us the Query structure and table names to the relevant information that we are trying to access. After we have the relevant table names, we just expanded the SQL query to pull the credit card information we want from user 'scruffy'

Here is the SQL injection code:

```
' and FALSE union SELECT card FROM credit_cards WHERE username='scruffy'
```

And the progression we made is below. It was relatively straightforward because we had SQL experience and took maybe about 15 minutes.

Networking

SQL Credit Cards

Find the credit card number

Get the credit card number for user: scruffy

Enter the credit card number here

SUBMIT

'SELECT username FROM credit_cards WHERE
username="" COLLATE NOCASE' - unrecognized
token: "" COLLATE NOCASE'

Networking

SQL Credit Cards

Find the credit card number

Get the credit card number for user: scruffy

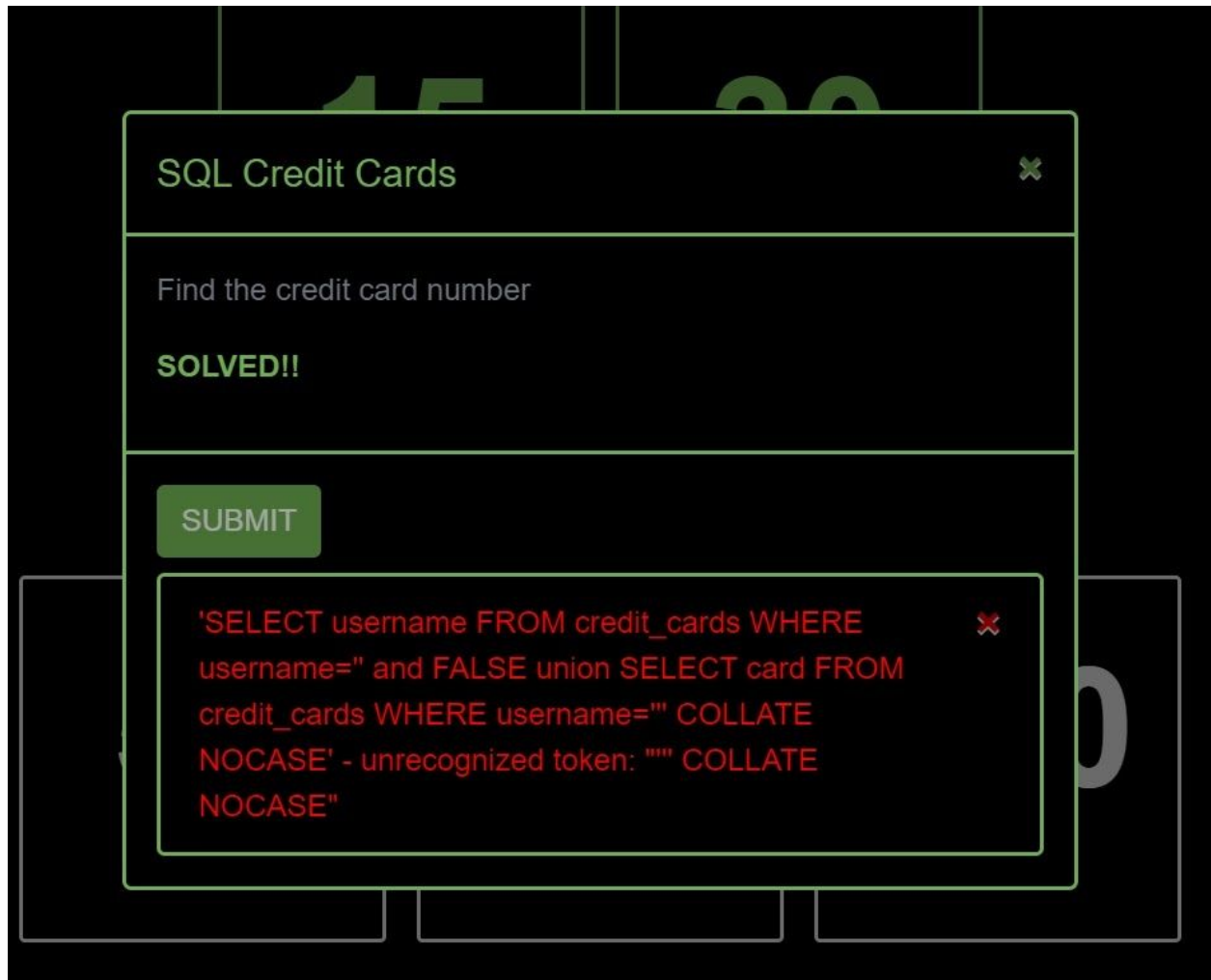
- 4987327898009549

Enter the credit card number here

SUBMIT

```
'SELECT username FROM credit_cards WHERE  
username=" and FALSE union SELECT card FROM  
credit_cards WHERE username="" COLLATE  
NOCASE' - unrecognized token: "" COLLATE  
NOCASE"
```

Exploitation



PROGRAMMING SECTION:

Brutal Force (15pts):

Basically we just wanted to find a number between 0-10000. So I just wrote a simple loop to increment *i* by 1 each time up to 10000 and submit the value of *i* until the correct answer was reached. We just wrote this function in the console and ran it. Simple. Straight Forward.


```
function brutalForce() {  
  for (let i = 0; i < 10000; i++) {
```

brutalForce();

We wrote a program to basically brute force our way into finding the value of the 7-character alphanumeric. Basically we created a string of all characters possible in an alphanumeric and since our score is above 0 if we get a single character correct irrelevant of position, we used this to create a list of characters that were used. Once we had a list of characters that we know are in the answer, we basically just looped each used alphanumeric through each position to until the score was above 0, and recorded the position that this occurred. After this runs, we have not only every alpha numeric that was in the final answer, but also it's corresponding position and thus we have our final answer.

codeshareExpires in 24 hoursSave Code

```
1 function charsUsed(chars, index = 0, usedChars = []) {
2   if (typeof chars === 'undefined') {
3     chars = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'.split('');
4   }
5   if (index >= chars.length) {
6     console.log(usedChars);
7     return Promise.resolve(usedChars);
8   }
9   let ch = chars[index];
10  let code = ch.repeat(7);
11  return new Promise(resolve => {
12    CodeBreaker_submit(code)
13      .then(score => {
14        console.log(code, score);
15        if (score > 0) {
16          usedChars.push(ch);
17        }
18        charsUsed(chars, index + 1, usedChars).then(resolve);
19      });
20  });
21 }
22
23 function findCharsPos(usedChars = [], index = 0, pos = 0, result = []) {
24   if (index >= usedChars.length) {
25     console.log(result);
26     return Promise.resolve(result);
27   }
28   if (pos >= 7) {
29     return findCharsPos(usedChars, index + 1, 0, result);
30   }
31   let ch = usedChars[index];
32   let code = '-'.repeat(pos) + ch + '-'.repeat(6 - pos);
33   return new Promise(resolve => {
34     CodeBreaker_submit(code)
35       .then(score => {
36         console.log(code, score);
37         if (score > 0) {
38           result[pos] = ch;
39         }
40         findCharsPos(usedChars, index, pos + 1, result).then(resolve);
41       });
42   });
43 }
44
45 function solutionCodeBreaker() {
46   charsUsed()
47     .then(usedChars => {
48       return findCharsPos(usedChars);
49     })
50     .then(result => {
51       let code = result.join('');
52       console.log(code);
53       CodeBreaker_submit(code);
54     });
55 }
56
57 solutionCodeBreaker()
```

 Slack
Slack brings the
wherever you a

Client-side Protections:

- **Super Admin (10 points):**
 - Solving this one was very trivial, as clicking on the button says “Javascript validation failed. You are not super admin. is_super_admin = false”. As a result, I could go over into the console and enter “is_super_admin = true” and then click submit, to solve the problem. This took Ash about a minute.

Super Admin

Are you admin tho?
You must be an admin to proceed.

SUBMIT

JavaScript validation failed. You are not super admin.
is_super_admin = false

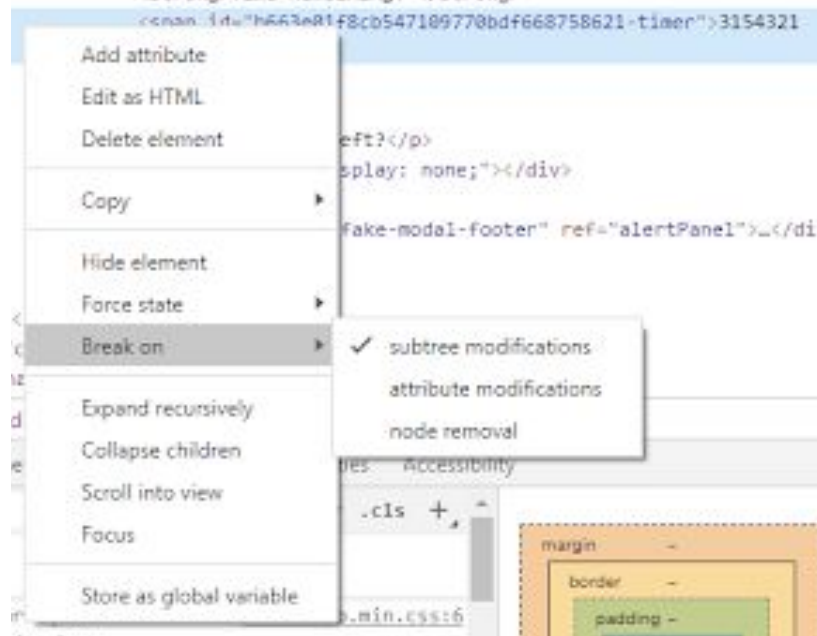
```
> is_super_admin = true  
< true
```

- **Timer (50 points):**

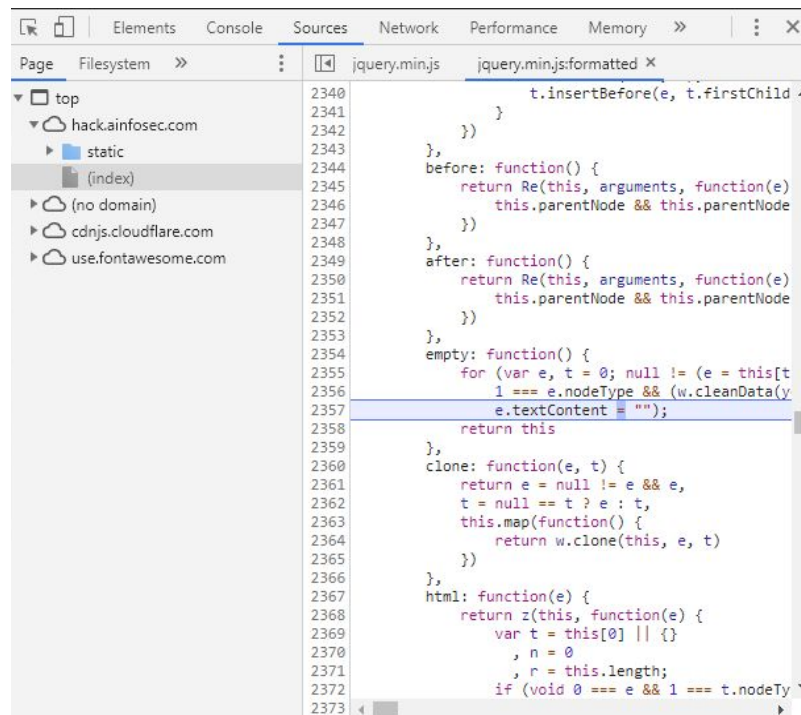
- This one required a little more insight into, and took some time to actually solve. For this challenge, I began by inspecting the timer object in Chrome:

```
<span id="b663e01f8cb547109770bdf668758621-timer">3155361  
</span> == $0
```

Any attempt to modify this object was unsuccessful, and so I tried some more unconventional methods to try and modify the timer element. As a result, I right clicked the element and tried messing around with different ways to have the code that updated the element pause, and the option that worked was Break on -> subtree modifications:



This then had the code pause when in some script that formatted the timer appearance:



Once I had reached this point, I kept clicking "Step over next function call", until I reached some code that seemed like something I could work with, where there was a variable called "seconds" that I could modify.


```

var timerId = setInterval(function() {
    display.text(seconds);
    seconds = seconds - 1;

    if (tick_cb) {
        tick_cb(seconds);
    }

    if (seconds <= 0) {
        HackerChallenge.endTimer(display);
    }
}, 1000);

display.data("_timerId", timerId);

return timerPromise;
};

```

Then, I could use the console to manually override the seconds variable:

```

> seconds=1

< 1

```

Once I entered that, I clicked on resume script execution until the Time Remaining said 0, and then hit submit. This took Ash about 30 minutes.

- **Paid Content (100 points):**

- I began by inspecting the submit button, and then clicking on it to see what happened in the console. The message that appeared was this:

```

✖ Failed to load resource: the server responded with a status of 400 (Bad Request) /challenge/submit-answer/:1
⚠ DevTools failed to parse SourceMap: chrome-extension://ekhagklcjbdpajgpjgmbionohlpd
bjgc/browser-polyfill.js.map
⚠ DevTools failed to parse SourceMap: https://hack.ainfosec.com/static/hackerchallenge/external/js/bootstrap.bundle.min.js.map
⚠ DevTools failed to parse SourceMap: https://hack.ainfosec.com/static/hackerchallenge/external/css/bootstrap.min.css.map
✖ POST https://hack.ainfosec.com/challenge/submit-answer/ 400 (Bad Request) jquery.min.js:2
>

```

Expanding the bad POST request shows the following info:

```
✖ POST https://hack.ainfosec.com/challenge/submit-answer/ 400 (Bad Request) jquery.min.js:2
  send @ jquery.min.js:2
  ajax @ jquery.min.js:2
  HackerChallenge.submitAnswer @ hackerchallenge.js:53
  tag.submitAnswer @ /static/hackerchallengeodal.tag.html.js:71
  1 @ jquery.min.js:2
  c @ jquery.min.js:2
  setTimeout (async)
  (anonymous) @ jquery.min.js:2
  u @ jquery.min.js:2
  add @ jquery.min.js:2
  (anonymous) @ jquery.min.js:2
  Deferred @ jquery.min.js:2
  then @ jquery.min.js:2
  tag.submit @ /static/hackerchallengeodal.tag.html.js:46
  (anonymous) @ riot%2Bcompiler.min.js:2
```

This reveals that the function we are probably interested in, is `HackerChallenge.submitAnswer`. Luckily, the console provides a direct link to the line of code that we are also interested in:

```
48   HackerChallenge.submitAnswer = function(challenge_id, answer) {
49     if (Array.isArray(answer) || typeof answer !== "string") {
50       answer = JSON.stringify(answer);
51     }
52
53     return $.ajax({
54       url: HackerChallenge.submitAnswerUrl,
55       method: "POST",
56       dataType: "json",
57       data: {
58         csrfmiddlewaretoken: HackerChallenge.csrfToken,
59         challenge_id: challenge_id,
60         answer: answer
61       }
62     });
63   };
```

Placing a breakpoint on that line and then clicking the submit button then allows us to control what happens instead of the ajax call, and so we can see what this “answer” is that is being submitted. Entering “answer” into the console shows us a very long dictionary, but one field in particular interests us:

```
"PaidContent", "paid": false}"
```

Seeing this, we now know to run the command: `answer =`

`answer.replace('"paid":false', '"paid":true');`. Then, we resume the debugger and the request goes through. This took Ash about 30 minutes.

Overthewire Wargames:

Level 0 & 0->1:

Basically here we just need to open a terminal and ssh into the website that was given using the following line:

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

So basically here we would want to ls to see what files are available and when we read the 'readme' file that shows up, it contains the password for the next round. It seems simple but we ran into several problems. The first problem was we tried to solve this on windows and were inspecting the elements like crazy only to realize it wasn't going anywhere. After spending about half an hour on this problem we figured that we needed to do the assignment on a linux based system with a terminal. It was a rough start.

```
For support, questions or comments, contact us through IRC on  
irc.overthewire.org #wargames.
```

```
Enjoy your stay!
```

```
bandit0@bandit:~$ ls  
readme  
bandit0@bandit:~$ cat readme  
boJ9jbbUNNfktd780OpsqOltutMc3MY1  
bandit0@bandit:~$
```

Level 2 Password: boJ9jbbUNNfktd780OpsqOltutMc3MY1

Level 1->2:

We first ssh into the website given to us for level 2. We enter our password from the previous phase and it works. We see a folder '-' so we basically just went into the directory and there was the password in the last file for our next phase. Clockwork.

For support, questions or comments, contact
irc.overthewire.org #wargames.

Enjoy your stay!

```
bandit1@bandit:~$ ls
-
bandit1@bandit:~$ ./-
-bash: ./-: Permission denied
bandit1@bandit:~$ ./-
-bash: ./-: Permission denied
bandit1@bandit:~$ cat ./-
CV1DtqXWVFXTvM2F0k09SHz0YwRINYA9
bandit1@bandit:~$ █
```

Password for phase 3: CV1DtqXWVFXTvM2F0k09SHz0YwRINYA9

Phase 2->3:

We ssh'd into the link for phase 3 and enter the password. When we typed the ls command, we got something really weird from the command "spaces in this file name". We've never really seen this before and were really confused if this was an error or what it was. After being confused we thought that is actually returned something so it must be related to names of directories. When we tried to enter the directory as one file it didn't work, I guess our assumption that this was a single directory was wrong. After a bunch of trial and error, we figured the formatting of the directories, (each one had a space in front of it) and from there we were able to navigate through the directories. This took about 20 minutes.

```
bandit2@bandit:~$ cat spaces \
> \ in
cat: spaces: No such file or directory
cat: ' in': No such file or directory
bandit2@bandit:~$ cat spaces\ in\ this\ filename
UmHadQcIWmgdLOKQ3YNgjWxGoRMb5luK
bandit2@bandit:~$
```

Password for phase 3: UmHadQcIWmgdLOKQ3YNgjWxGoRMb5luK

Phase 3->4:

We ssh'd into phase 4 with the password from the previous phase. When we went into the directory, there were no files whatsoever. This took us a couple minutes to figure out but we came to the conclusion that the answer had to be somewhere in the directory, there was simply no where else it could be. We googled the flag to show hidden files and it worked. The hidden file was shown and we were able to cat into it for the password into the next phase. This took about 10 minutes to figure out.

Enjoy your stay!

```
bandit3@bandit:~$ ls
inhere
bandit3@bandit:~$ cat inhere
cat: inhere: Is a directory
bandit3@bandit:~$ cd inhere
bandit3@bandit:~/inhere$ ls
bandit3@bandit:~/inhere$ ls -al
total 12
drwxr-xr-x 2 root    root    4096 Oct 16  2018 .
drwxr-xr-x 3 root    root    4096 Oct 16  2018 ..
-rw-r----- 1 bandit4 bandit3   33 Oct 16  2018 .hidden
bandit3@bandit:~/inhere$ cat .hidden
pIwrPrtPN36QITSp3EQaw936yaFoFgAB
bandit3@bandit:~/inhere$
```

Password for phase 4: plwrPrtPN36QITSp3EQaw936yaFoFgAB

Phase 4->5:

So same process, we ssh into the phase and enter the password. When we went into the "inhere" directory and used the ls command there were a bunch of weird files. When we tried to access the files it just wouldn't work. Basically we knew that we had to do something with one of the files here so we tried to use different flags to see if we could gather additional information but it lead to a dead end. So in our last attempt to just try to open every file, the 7th file actually opened and gave us our password. This took about 20 minutes.

```
bandit4@bandit:~/inhere$ cat ./-file00
bandit4@bandit:~/inhere$ cat ./-file01
bandit4@bandit:~/inhere$ cat ./-file02
bandit4@bandit:~/inhere$ cat ./-file03
bandit4@bandit:~/inhere$ cat ./-file04
bandit4@bandit:~/inhere$ cat ./-file05
bandit4@bandit:~/inhere$ cat ./-file06
bandit4@bandit:~/inhere$ cat ./-file07
koReBOKuIDDepwhWk7jZC0RTdopnAYKh
bandit4@bandit:~/inhere$
```

Password for phase 5: koReBOKuIDDepwhWk7jZC0RTdopnAYKh

Phase 5->6:

Once we ssh'd in and entered the directory, we were really confused for about 15 minutes trying various things (mostly trying to brute force and seeing if we'd get lucky) but that ended up failing and we considered the hints that was provided. Know that the file we were looking for was human readable, 1033 bytes in size, and not executable, we looked up the flags we needed to be able to find the file. Probably took us half an hour in total.

```
-executable
Matches files which are executable and directories which are
searchable (in a file name resolution sense). This takes into
account access control lists and other permissions artefacts which
the -perm test ignores. This test makes use of the access(2) sys-
tem call, and so can be fooled by NFS servers which do UID mapping
(or root-squashing), since many systems implement access(2) in the
client's kernel and so cannot make use of the UID mapping informa-
tion held on the server. Because this test is based only on the
result of the access(2) system call, there is no guarantee that a
file for which this test succeeds can actually be executed.
```

```
-size n[cwbkMG]
    File uses n units of space, rounding up. The following suffixes
    can be used:

    `b'   for 512-byte blocks (this is the default if no suffix is
          used)

    `c'   for bytes

    `w'   for two-byte words

    `k'   for Kilobytes (units of 1024 bytes)

    `M'   for Megabytes (units of 1048576 bytes)

    `G'   for Gigabytes (units of 1073741824 bytes)
```

```
bandit5@bandit:~/inhere$ find . -type f -readable ! -executable -size 1033c
./maybehere07/.file2
bandit5@bandit:~/inhere$ cat ./maybehere07/.file02
cat: ./maybehere07/.file02: No such file or directory
bandit5@bandit:~/inhere$ cat ./maybehere07/.file2
DXjZPULLxYr17uwoI01bNLQbtFemEgo7
```

Password for Phase 6: DXjZPULLxYr17uwoI01bNLQbtFemEgo7

Phase 6-7:

This problem took about 20 minutes mostly because we were looking up the flags we needed in order to filter out what we were looking for on the server. Even though we had a lot of trial and error trying to find the right flags and formatting, we consistently made progress and felt confident because it was such a natural progression from the last question.

Enjoy your stay!

```
bandit6@bandit:~$ ls
bandit6@bandit:~$ find / -user bandit7 -group bandit6 -size 33c 2>/dev/null
/var/lib/dpkg/info/bandit7.password
bandit6@bandit:~$ cat /var/lib/dpkg/info/bandit7.password
HKBPTKQnIay4Fw76bEy8PVxKEDQRKTzs
bandit6@bandit:~$
```

Password for Phase 7: HKBPTKQnIay4Fw76bEy8PVxKEDQRKTzs

Phase 7->8:

This question was a breath of fresh air because we were expecting the difficulty to ramp up, but we were pretty familiar already with commands dealing with opening and parsing through files so this took maybe about 5 minutes.

Enjoy your stay!

```
bandit7@bandit:~$ HKBPTKQnIay4Fw76bEy8PVxKEDQRKTzs
-bash: HKBPTKQnIay4Fw76bEy8PVxKEDQRKTzs: command not found
bandit7@bandit:~$ ls
data.txt
bandit7@bandit:~$ cat data.txt | grep millionth
millionth      cvX2JJJa4CFALtqS87jk27qwqGhBM9pIV
bandit7@bandit:~$
```

Password for Phase 8: cvX2JJJa4CFALtqS87jk27qwqGhBM9pIV

Phase 8->9:

We loved how this problem was so similar in progression to the previous question and because we were familiar with files on linux, it was pretty intuitive for us to pipe and use sort and uniq flags to narrow down the text that only occurs once and is unique. We did try to only use the unique -u flag but we got a ton of results so we knew that we needed to sort it after that. Overall Maybe about 10 minutes.

```
bandit8@bandit:~$ ls
data.txt
bandit8@bandit:~$ cat data.txt | sort | uniq -u
UsvVyFSfZZWbi6wgC7dAFyFuR6jQQUhR
bandit8@bandit:~$ █
```

Password for Phase 9: UsvVyFSfZZWbi6wgC7dAFyFuR6jQQUhR

Phase 9->10:

Again, really loving dealing with files. A smooth transition from the previous two questions. We knew that we wanted strings to be returned and we knew that we wanted to search the plain text data set that matches the equal symbol. It took about 5 minutes of trial and error but eventually we got returned a small list of entries, with only one line that contained something that looked like our password for the next phase.

```
bandit9@bandit:~$ ls
data.txt
bandit9@bandit:~$ strings data.txt | grep '='
2===== the
===== password
>t=      yP
rV~dHm=
===== isa
=FQ?P\U
=      F[
pb=x
J;m=
=)$=
===== truKLdjsbJ5g7yyJ2X2R0o3a5HQJFuLk
iv8!=
bandit9@bandit:~$
```

Password for Phase 10: truKLdjsbJ5g7yyJ2X2R0o3a5HQJFuLk

Phase 10->11:

Basically just ls'ing and reading the file, and knowing that the file contained something that was base64 encoded from the hint provided, we knew that we had to do some sort of decoding. The string that was provided in the file was too long and was not the correct answer to the next phase so we searched up the command to base64 decode in the terminal. A simple search and we were able to find the password and move on to the next phase. It was pretty cool because this was really familiar to use because of the first project we did where we set up server to client side encrypted communications.

Enjoy your stay!

```
bandit10@bandit:~$ ls
data.txt
bandit10@bandit:~$ cat data.txt
VGhlIHBhc3N3b3JkIGlzIElGdWt3S0dzRlc4TU9xM01SRnFyeEUxaHhUTkViWBSCg==
bandit10@bandit:~$ cat data.txt | base64 --decode
The password is IFukwKGsFW8MOq3IRFqrxE1hxTNEbUPR
bandit10@bandit:~$
```

Password for Phase 11: IFukwKGsFW8MOq3IRFqrxE1hxTNEbUPR

Phase 11->12:

So this question really threw us off because of a lot of new terminology, i.e we didnt know what a ROT was. After googling we found out that it was basically a cipher that chooses a letter 13 positions down from its relative position. The string in the file is basically rot'd 13 times for every letter, and probably says The Password is : blah blah blah. We just used a ROT13 decoder online and typed in the string in the file and got our next password. <https://rot13.com/>

Enjoy your stay!

```
bandit11@bandit:~$ ls
data.txt
bandit11@bandit:~$ cat data.txt
Gur cnffjbeq vf 5Gr8L4qetPEsPk8htqjhRK8XSP6x2RHh
bandit11@bandit:~$
```

rot13.com

[About ROT13](#)

Gur cnffjbeq vf 5Gr8L4qetPEsPk8htqjhRK8XSP6x2RHH



ROT13 ▼



The password is 5Te8Y4drgCRfCx8ugdwuEX8KFC6k2EUu

Password for phase 12: The password is 5Te8Y4drgCRfCx8ugdwuEX8KFC6k2EUu

Phase 12->13:

This was basically really complicated but interesting familiar from Computer Systems - CSCI2400. It took us about 2 hours of work and a ton of googling. Looking up the commands took a bit of effort but it was harder going through the progression because each command unlocked another part.


```
bandit12@bandit:/tmp/jason$ file data8.bin
data8.bin: gzip compressed data, was "data9.bin", last modified: Tue Oct 16 12:00:23 2018, max compression, from Unix
bandit12@bandit:/tmp/jason$ zcat data8.bin > data8_zcat
bandit12@bandit:/tmp/jason$ file data8_zcat
data8_zcat: ASCII text
bandit12@bandit:/tmp/jason$ cat data8_zcat
The password is 8ZjyCRiBWFYkneahHwxCv3wb2a1ORpYL
bandit12@bandit:/tmp/jason$
```

Password for next Phase: 8ZjyCRiBWFYkneahHwxCv3wb2a1ORpYL