

PROJECT 3: CSCI 3403

Devin Murray: demu2748@colorado.edu

Jacob Christiansen: jach7037@colorado.edu

Nick Price: npr2175@colorado.edu

BANDIT PASSWORDS

1. boJ9jbbUNNfktD78OOpsqOltutMc3MY1
2. CV1DtqXWVFXTvM2F0k09SHz0YwRINYA9
3. UmHadQclWmgdLOKQ3YNngjWxGoRMb5luK
4. plwrPrtPN36QITSp3EQaw936yaFoFgAB
5. koReBOKulDDepwhWk7jZC0RTdopnAYKh
6. DXjZPULLxYr17uwoI01bNLQbtFemEgo7
7. HKBPTKQnlay4Fw76bEy8PVxKEDQRKTzs
8. cvX2JJJa4CFALtqS87jk27qwqGhBM9pIV
9. UsvVyFSfZZWbi6wgC7dAFyFuR6jQQUhR
10. truKLdjsbJ5g7yyJ2X2R0o3a5HQQJFuLk
11. IFukwKGSFW8MOq3IRFqrxE1hxTNEbUPR
12. 5Te8Y4drgCRfCx8ugdwuEX8KFC6k2EUu
13. 8ZjyCRiBWFYkneahHwxCv3wb2a1ORpYL
14. ssh -i sshkey.private bandit14@localhost
15. BfMYroe26WYalil77FoDi9qh59eK5xNr
16. cluFn7wTiGryunymYOu4RcffSxQluehd
17. -----BEGIN RSA PRIVATE KEY-----
MIIeogIBAAKCAQEAvmOkuifmMg6HL2YPIOjon6iWfbp7c3jx34YkYWqUH57SUdyJ
imZzeyGC0gtZPGujUSxiJSWl/oTqexh+cAMTSMIOJf7+BrJObArndx9Y7YT2bRPQ
Ja6Lzb558YW3FZl87ORiO+rW4LCDCNd2IUvLE/GL2GWyuKN0K5iCd5TbtJzEkQTu
DSt2mcNn4rhAL+JFr56o4T6z8WWAW18BR6yGrMq7Q/kALHYW3OeKePQAZL0VUYbW
JGTi65CxbCnzc/w4+mQqYvmzpwWtMAzJTzAzQxNbkR2MBGySxDLrjg0LWN6sK7wNX
x0YVztz/zblKpJfU1jHS+9EbVNj+D1XFOJuaQIDAQABAoIBABagpxpM1aoLWfvD
KHcj10nqcoBc4oe11aFYQwik7xfW+24pRNUDE6SFthOar69jp5RILwD1NhPx3iBI
J9nOM8OJOVToum43UOS8YxF8WwhXriYGnc1sskbwpXOUDc9uX4+UESzH22P29ovd
d8WErY0gPxun8pbJLmxkAtWNhpMvfe0050vk9TL5wqbu9AlbssgTcCXkMQnPw9nC
YNN6DDP2lbcBrvgT9YCNL6C+ZKufD52yOQ9qOkwFTEQpjtF4uNtJom+asvlpmS8A
vLY9r60wYSvmZhNqBUrj7lyCtXmLu1kkd4w7F77k+DjHoAXyxcUp1DGL51sOmama
+TOWWgECgYEA8JtPxP0GRJ+IQkX262jM3dElkza8ky5molwUqYdsx0NxHgRRhORT
8c8hAuRBb2G82so8vUHk/fur85OEfc9TncnCY2crpoqsgghifKLxrlgtT+qDpfZnx
SatLdt8GfQ85yA7hnWWJ2Mx3F3NaeSDm75Lsm+tBbAiyC9P2jGRNtMSKcgyEAYpHd
HCctNi/FwjulhttFx/rHYKhLidZDFYeiE/v45bN4yFm8x7R/b0iE7KaszX+Exdvt
SghaTdcG0Knyw1bpJVyusavPzpaJMjdJ6tcFhVAbAjm7enCivGCSx+X3l5SiWg0A
R57hJglezliVjv3aGwHwvZvtszK6zV6oXFAu0ECgYABjo46T4hyP5tJi93V5HDI
TtieK7xRVxUl+iU7rWkGAXFpMLFteQEsRr7PJ/lemmEY5eTDAFmLy9FL2m9oQWCg
R8VdwSk8r9FGLS+9aKcV5PI/WEKIwgXinB3OhYimtiG2Cg5JCqlZFHxD6MjEGOIu

-----END RSA PRIVATE KEY-----

- 26.5czgV9L3Xx8JP0yRbXh6lQbmIOWvP | 6Z

NATAS PASSWORDS

1. gtVrDuiDfck831PqWsLEZy5gyDz1clto
 - a. It said the password could be found on the page so I inspected the page. It was a comment in the div. I checked the console first. Literally the first thing I did.
2. ZluruAthQk7Q2MqmDeTiUij2ZvWy2mBi
 - a. It said right-clicking was locked but my inspect window was still open from one and it was literally in the same spot.
3. sJIJNW6ucpu6HPZ1ZAchaDtwd7oGrD14
 - a. Initially, I kept opening this image, it was black but it obviously shouldn't have been there. Overthewire haha. Eventually, I checked the source file and found users.txt, the password was in there.
4. Z9tkRkWmpt9Qr7XrR5jWRkgOU901swEZ
 - a. I solved this the same way except. I copy and pasted the "not even google will help you" message into google and while natas was the top result I learned what disallowed webpages are and how they're normally kept in robots.txt. When I went to the disallowed directory I found another users.txt.
5. iX6lOfmpN7AYOQGPwtn3fXpbaJVJcHfq
 - a. I tried to use mtmproxy but the internet said I was stupid and should be using burp. I got the community version and tried looking at the index.php because upon refreshing level 4 the url added index to itself. Eventually, there's this thing called a referer. It's pretty much shitty access control by referring "verified" links and redirects. I just had to change a 4 to a 5.
6. aGoY4q2Dc6MgDq4oL4YtoKtyAg9PeHa1
 - a. It was the same thing as the previous level except it said I'm not logged in. I just looked at everything and it actually took me many minutes to see the logged tag. I've been taking CS classes for a couple of years now so I know 1 means yes and 0 means no. It was 0, logged in = no.
7. 7z3hEENjQtflzgnt29q7wAvMNFZdh0i9
 - a. I tested abcd thinking I could translate that pretty easy, count bits and whatnot if I had to. It was wrong. Turns out there's a view source code link so uh duh. Nothing seems to be happening that I can manipulate but it includes a .inc file. I don't know exactly what that is but I actually tried like 10 different directories before I got the includes. I googled html include directories but I kept trying things like src and scripts. Html things.
8. DBfUBfqQG69KvJvJ1iAbMolpwSNQ9bWe


- a. More source code, it says where the answer is as a hint. Again it took me a while to figure out that the php should equal the location. I was trying it from the home page, about page, even as trying to enter the level at the /etc/ directory.
9. W0mMhUcRRnG8dcghE4qvK3JA9IGt8nDI
 - a. I tested the same way as 7 but upon looking I saw the secret being encoded. I googled the return line but the results were just natas8. So it's probably not industry-standard encryption. I did a hex2bin online decoder on the secret 3d3d516343746d4d6d6c315669563362 then a string reverse then a base64 decoder. It was an arduous journey copy and pasting and googling while the answer was right there.
10.
 - a. Again testing with abcd reveals nothing. It is the obvious view source code link that holds the solution. Upon looking at the source code I could see an answer key is dictionary.txt. I tried natas, natas9/10, and then I played with it because it's pretty big. I spent a couple minutes typing in a b c and d respectively and quickly scrolling through the output looking for some random string. Eventually, I looked up the pass through function. Apparently, you can just run a cat. From there I googled where are natas passwords held and found /etc/natas_webpass/natasX. Took about 30 mins all told. Huge pain in the ass. How was I supposed to know where the password was


PORTSWIGGER:


SQL Injections(5):

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




LAB SQL injection UNION attack, finding a column containing text >>  Solved

LAB SQL injection UNION attack, retrieving data from other tables >>  Solved



LAB SQL injection UNION attack, retrieving multiple values in a single column >>  Solved

LAB SQL injection attack, querying the database type and version on Oracle >>  Solved

XSS Attacks (5):

- LAB** Reflected XSS into HTML context with nothing encoded >>  Solved
- LAB** Reflected XSS with event handlers and href attributes blocked >>  Solved
- LAB** Reflected XSS in canonical link tag >>  Solved
- LAB** Reflected XSS in a JavaScript URL with some characters blocked >>  Solved
- LAB** Reflected XSS with AngularJS sandbox escape without strings >>  Solved

CSRF Attacks (2):

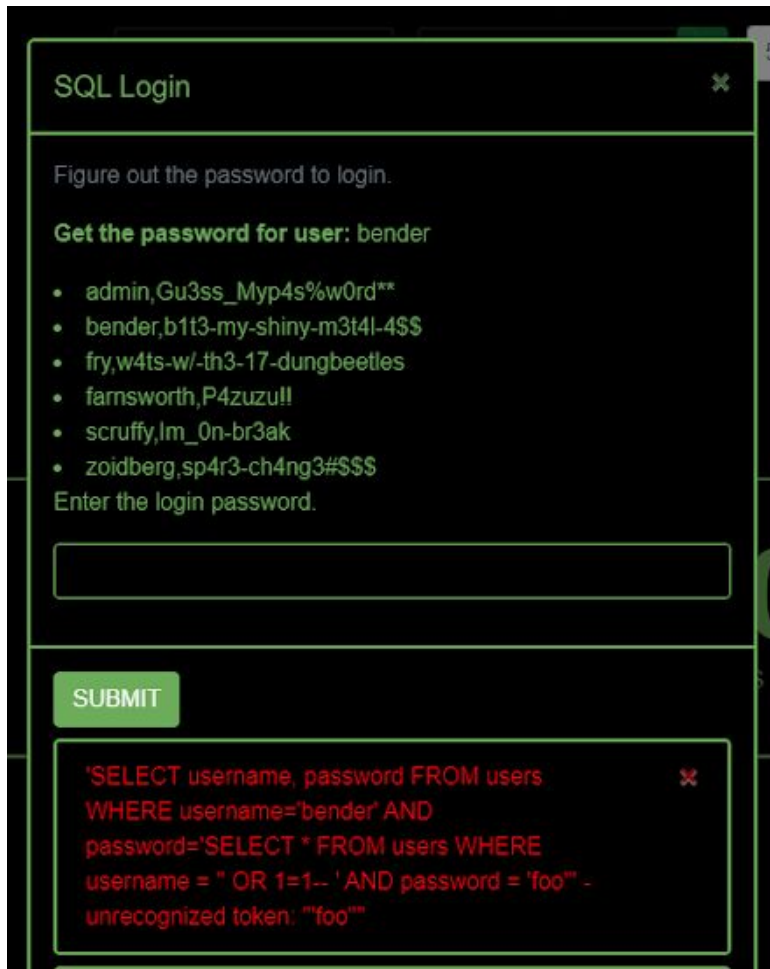
- LAB** CSRF vulnerability with no defenses >>  Solved
- LAB** CSRF where token validation depends on request method >>  Solved

REAL WORLD (AIS):

ID used: f866cacf-7c48-4970-8694-514a9500a49e

Input Validation

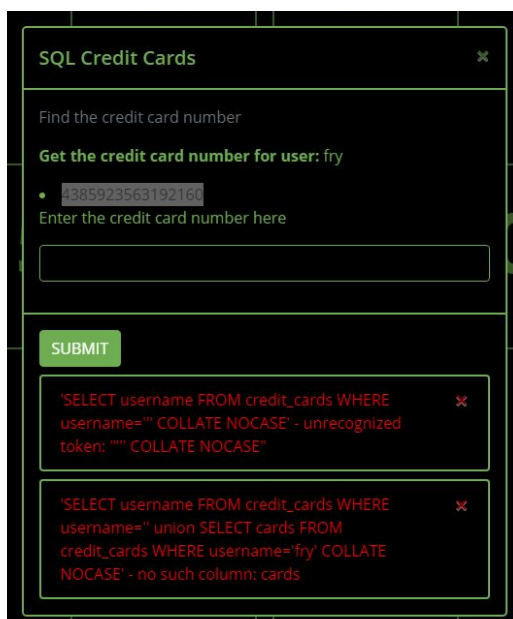
1. This is a sql exploit. Since I knew it was a sql exploit I just went straight to Google. I actually tried a select statement on my own before checking the internet for examples. I spent about about 10 minutes using a variation of `SELECT password/* FROM users/pg_shadow WHERE user = '/'bender'/'`. I actually found an example using ' or '1'='1 but initially I didn't know what it was so I didn't think it was necessary. it took me about another 5 minutes to get the right combination of the only input I actually needed.



2. I approached this one similar to the XSS stuff from portswigger, trying combinations of inputs to really see what was going on. Some simple alerts wouldn't work because of AIS's "anti-XSS" implementation, but quickly I realized that the image started disappearing, and I could take advantage of that. It took me a while of googling and looking back on portswigger to get a correct input - I noticed that Chrome wasn't showing pop-ups, which threw me off a bit before I switched to Firefox. I eventually entered "><svg onload=alert(1234)>" as a test, and got a response. Shortly after I figured out that I could simply enter the cookie into the alert, and get the answer: "><svg onload=alert(document.cookie)>", which gave the response: admin_sess_id=flag{false-p4cket-5niff3r=8in}

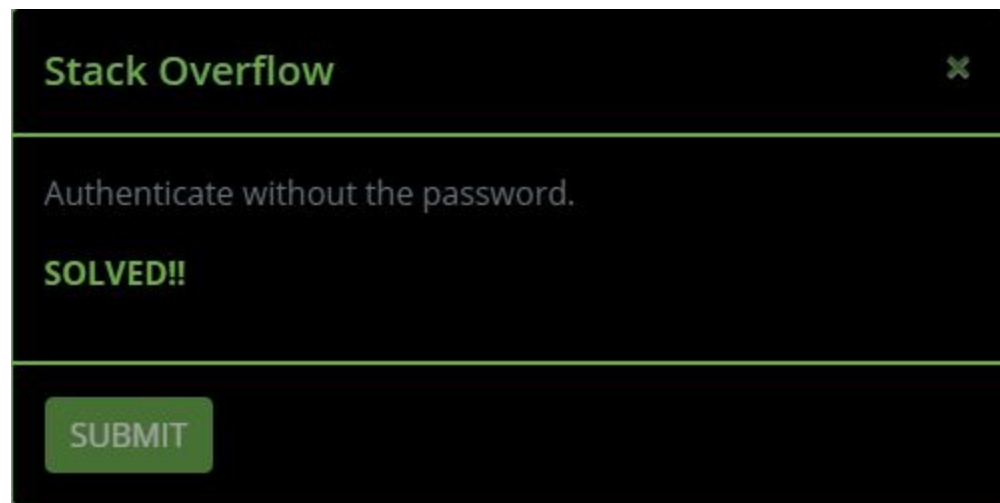


3. SQL Credit Cards: Similar to #1, this problem is also a relatively simple SQL exploit. Firstly, by inputting ' or '1'='1', get a "That's not the right number" error, along with the SQL error response of 'SELECT username FROM credit_cards WHERE username='' COLLATE NOCASE' - unrecognized token: '' COLLATE NOCASE". This gives us almost enough information to craft a query to send out and receive the information needed. I can craft "SELECT [credit card item name] FROM credit_cards WHERE username='fry". It doesn't take much guessing to figure out that [credit card item number] is just "card". Now you can run the command with a SQL union between the exploit and the query: " ' union SELECT card FROM credit_cards WHERE username='fry ", and our credit card number is displayed!







Exploitation:







1. Firstly, it's obvious that this code can be exploited with a buffer overflow due to its use of the now-deprecated "gets" function. I decided right away to start testing the code in my own environment, so I could see how the variables/functions reacted to different password inputs. I realized that the input for the password was limited at 12 chars, which meant that (because of 'gets') an added char would overflow to the stack's next variable. That next variable was authenticated! So for example, if you submit uuuuuuuuuuuuu (13 u's) as the password, it will overflow, and then authenticated will equal u, or 117 according to the ASCII table. This method worked with different chars in our testing.



EXTRA CREDIT PROBS (10):

We went back and did some extra portswigger problems, during our attempts to solve AIS problems, and after the fact. All completed blindly.

LAB	Stored XSS into HTML context with nothing encoded >>	 Solved
LAB	SQL injection vulnerability in WHERE clause allowing retrieval of hidden data >>	 Solved
LAB	SQL injection attack, listing the database contents on non-Oracle databases >>	 Solved
LAB	SQL injection attack, listing the database contents on Oracle >>	 Solved

LAB	DOM XSS in document.write sink using source location.search »	 Solved
LAB	DOM XSS in document.write sink using source location.search inside a select element »	 Solved
LAB	DOM XSS in innerHTML sink using source location.search »	 Solved
LAB	DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded »	 Solved
LAB	Reflected DOM XSS »	 Solved
LAB	Stored DOM XSS »	 Solved

DIVISION OF LABOR:

Devin was able to complete both Bandit and Natas before Jacob and Nick had a chance to start working on the project. They went back and completed some of those challenges on their own in order to be prepared for the rest of the project, and to double check Devin's solutions. Then Jacob completed Portswigger. Finally, everyone came together to complete the hack.ainfosec.com challenges & extra credit.