zsombii - Easy cracme — Reverse Engineering Write-up

Challenge link: https://crackmes.one/crackme/693d89b50992a052ab2222d7
Author: *zsombii*
Write-up by: *SenorGPT*
Tools used: *JADX*

Platform	Difficulty	Quality	Arch	Language
Multiplatform	1.0	4.0	java	Java

Status: Complete
Goal: Document a clean path from initial recon → locating key-check logic → validation/reversal strategy

zsombii - Easy cracme — Reverse Engineering Write-up

---

1. Executive Summary

This write-up documents my reverse-engineering process for `Easy crackme` by `zsombii`. The target is a Java `.jar`, so my usual native tooling (*CFF Explorer*, *x64dbg*) isn't the right fit here. This one is all about decompiling *Java* bytecode and reasoning about the validation logic.

Using *JADX*, I recovered readable Java source and immediately identified the input gate: the program loops until the user supplies a key of **exactly 10 characters**, then passes that value into `checkValidity()`.

The validation itself boils down to a simple scoring rule: `keyPoints` increments once per character when `(char & 3) == 0` (i.e., the character's ASCII value is divisible by 4), and the key is accepted only when the final score is **exactly 4**. The `startsWith("KEY")` branch is dead code and doesn't affect the outcome.

To make the challenge a bit more interesting, I built a small *Python* keygen that mirrors the validator and can generate valid 10-character keys on demand.

---

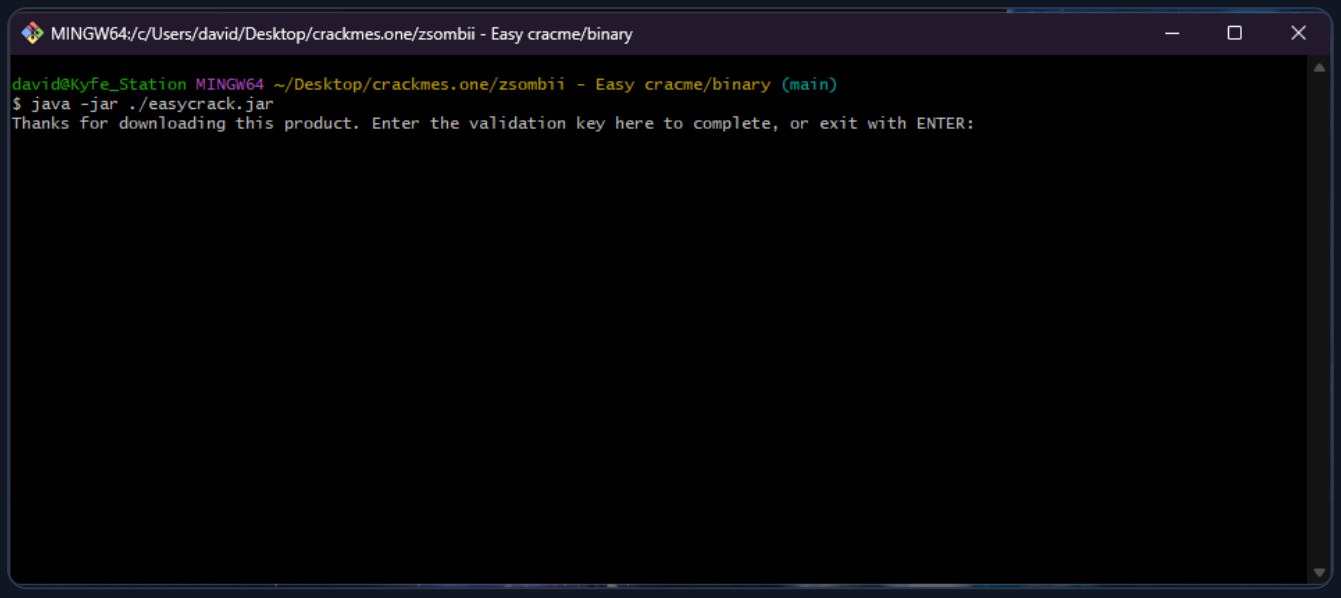
2. Target Overview

2.1 UI / Behaviour

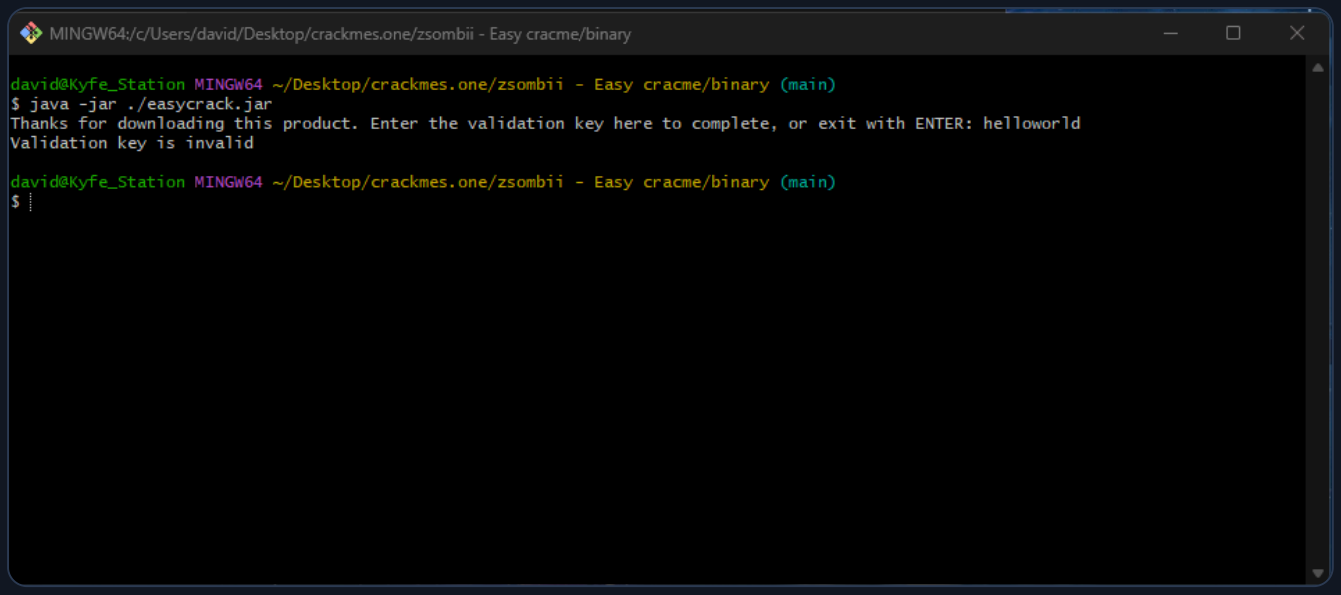
- Inputs: *A validation key.*
- Outputs: *Thanks for downloading this product. Enter the validation key here to complete, or exit with ENTER:*
    - Invalid key: *Validation key is invalid*
    - Key length not 10 characters: *Length requirement not met*
    - No Input: *Exiting...*

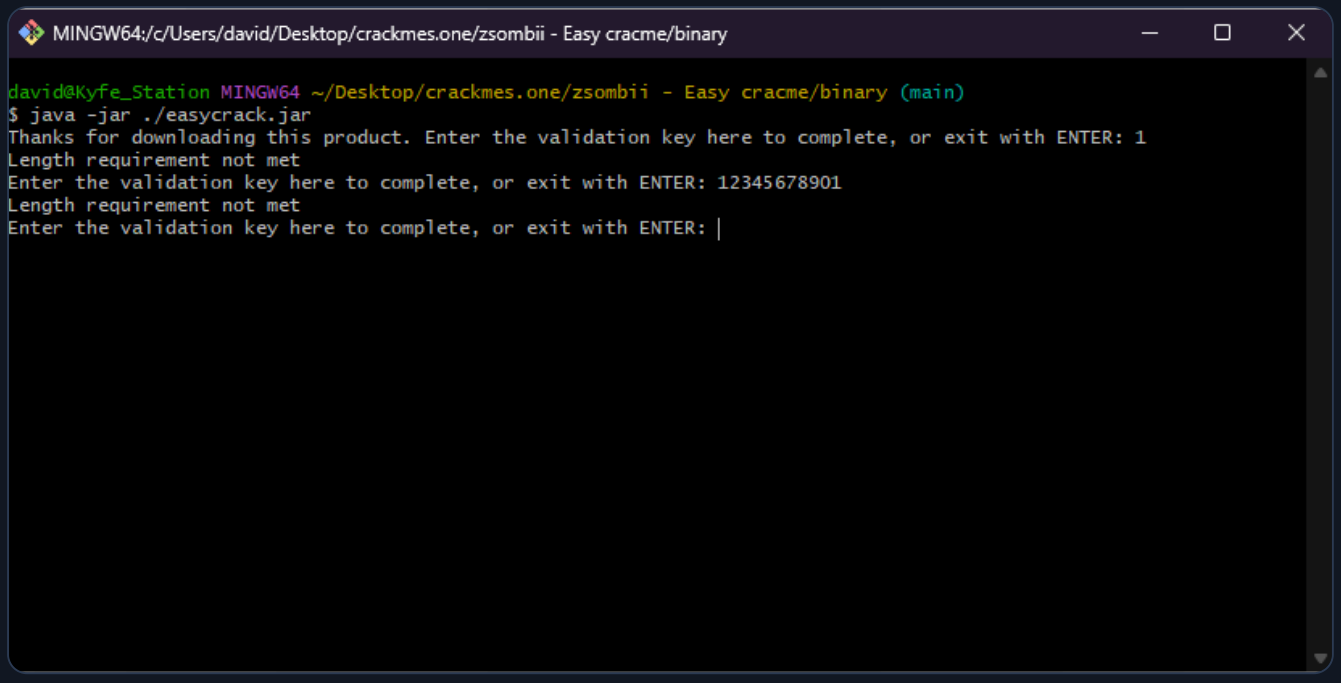
2.2 Screens

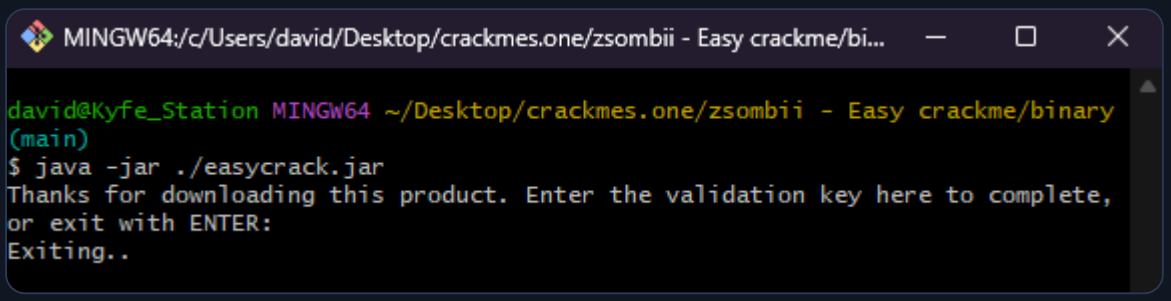
Start-up



Failure case







3. Tooling & Environment

- OS: *Windows 11*
- Static tools: *JADX*

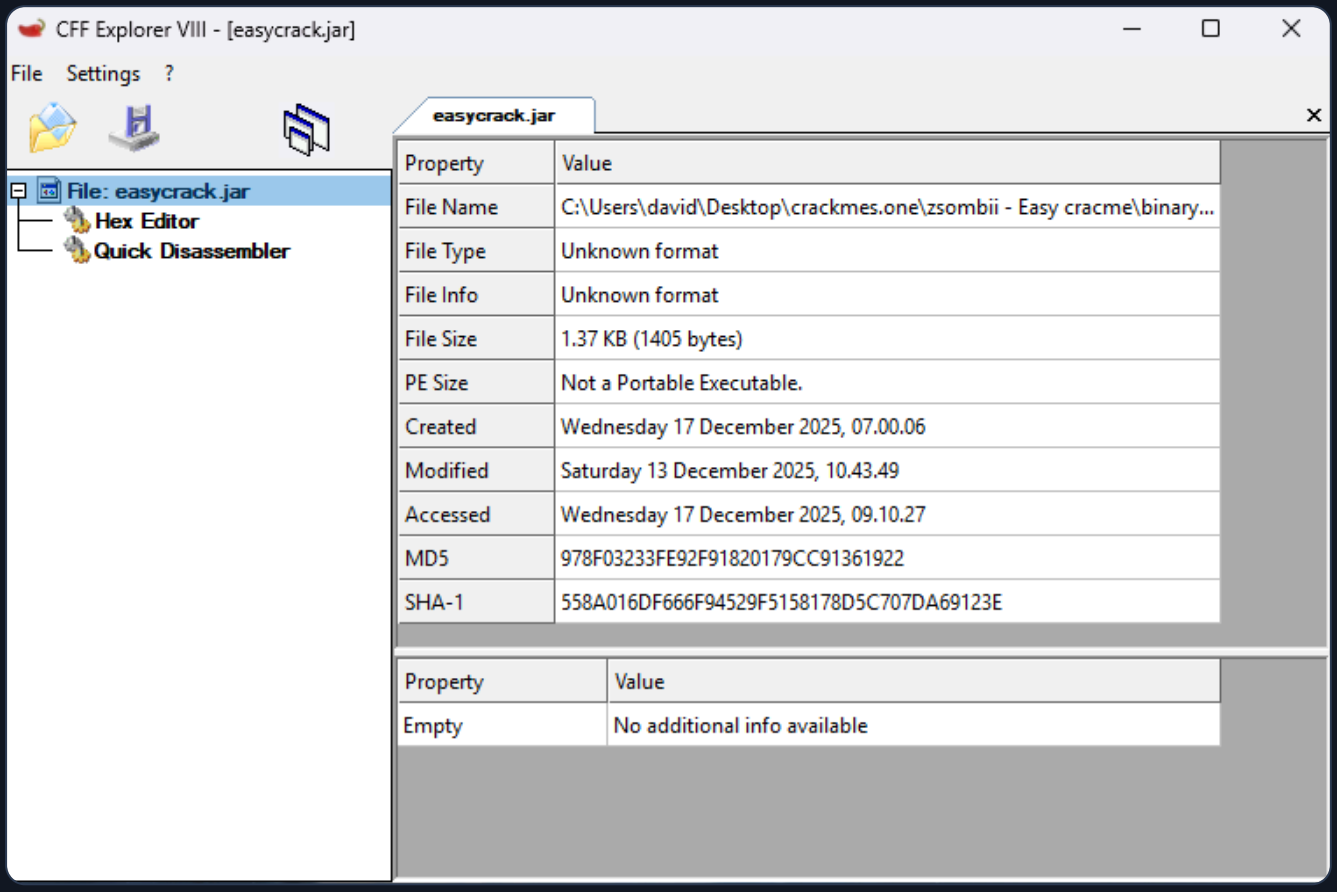
During initial recon of this crackme, it became clear to me that my current kit (*x64dbg, CFF Explorer*) will not be sufficient enough to handle this task. I start looking into Java reverse engineering tools that I can add to my toolbox.

I come across: *JADX* for triaging and searching, *Recaf* for patching and repacking the `.jar`, and *IntelliJ* for debugging and confirming logic. Although, during this crackme I only had to utilize *JADX*.

---

4. Static Recon

Opening the binary within *CFF Explorer* doesn't reveal much information.



Opening the `.jar` file within *JADX* provided me with the following source code.

```
package defpackage;

import java.util.Scanner;

/* loaded from: easycrack.jar:Main.class */
public class Main {
    public static void main(String[] args) {
        String key;
        Scanner scanner = new Scanner(System.in);
        System.out.print("Thanks for downloading this product. ");
        do {
            System.out.print("Enter the validation key here to complete, or exit
with ENTER: ");
            key = scanner.nextLine();
            if (key.trim().isEmpty()) {
                System.out.println("Exiting..");
                System.exit(0);
            }
            if (key.length() != 10) {
                System.out.println("Length requirement not met");
            }
        } while (key.length() != 10);
        if (!checkValidity(key)) {
            System.out.println("Validation key is invalid");
            System.exit(0);
        } else {
            System.out.println("Validation key accepted");
        }
    }

    public static boolean checkValidity(String key) {
        int keyPoints = 0;
        if (key.startsWith("KEY")) {
            keyPoints = 0 + 0;
        }
        for (int i = 0; i < key.length(); i++) {
            keyPoints += (key.charAt(i) & 3) == 0 ? 1 : 0;
        }
        return keyPoints == 4;
    }
}
```

It seems that the program loops indefinitely until the user enters an input that is exactly *10* characters long. Otherwise, "*Length requirement not met*" is printed and key input is re-prompted. Once the user enters in a key that is 10 characters long it proceeds to check the validity of that key. If the key is valid, it prints "*Validation key accepted*". Otherwise, on an invalid key, it will print "*Validation key is invalid*". If no key is entered, it will print "*Exiting...*" and terminate execution.

---

5. Dynamic Analysis

x64dbg does not support opening `.jar` files. Furthermore, since I was able to extract the source code from the `.jar` file using *JADX*, there was no need for dynamic analysis. Although, it is important to keep in mind that decompilers can lie under obfuscation and runtime debugging (*IntelliJ / JDWP*) is how to confirm what actually executes.

---

6. Validation Path

The obvious validity checking function `checkValidity` stood out immediately.

```
public static boolean checkValidity(String key) {
    int keyPoints = 0;
    if (key.startsWith("KEY")) {
        keyPoints = 0 + 0;
    }
    for (int i = 0; i < key.length(); i++) {
        keyPoints += (key.charAt(i) & 3) == 0 ? 1 : 0;
    }
    return keyPoints == 4;
}
```

`keyPoints` is effectively a *score counter*. During the loop it increments by *1* each time a character meets the bitmask condition, and the key is considered valid only if the total score is *exactly 4* once all characters have been processed.

The first conditional is completely useless and can be removed as it does nothing of value and importance. The interesting part is in the `for` loop. The amount of times the loop runs is dependent on the length of the user input. *BUT*, since the user input restricts us to only allow a *10* character input this loop will ***ALWAYS*** loop 10 times.

`key.charAt(i)` gives a `char`, a 16-bit integer under the hood.

`& 3` is a **bitmask**; `3` in binary is `0b11`. So `char & 3` keeps **only the lowest 2 bits** of the character's numeric value.

So essentially what `(key.charAt(i) & 3)` is doing is: *taking the character code and look at it modulo 4*. This is because masking with `0b11` is equivalent to `value % 4` for non-negative integers.

Because `4 = 2²`, the remainder mod 4 is stored in the **lowest 2 bits**, so:

$$n \& 3 \equiv n \bmod 4$$

The comparison after to `== 0` is checking if that character code is divisible by *4*. If so, it increments the `keyPoints` counter by one.

A character counts if its ASCII value ends in binary with `..00` (last two bits are 0). So in plain English **every 4th ASCII value** counts.

Any non-negative integer `n` can be written as:

$$n = 4q + r \quad \text{where } r \in \{0, 1, 2, 3\}$$

If we take the character `$`, it's *ASCII* decimal value is *36*. Which in binary is `0010 0100`. Since the last two bits are *0* we know that this character will increment the points counter.
Lets also take another character; `!` which has an *ASCII* decimal value of 33, which in binary is `0010 0001`. The last bit is a *1* which means that this character will ***NOT*** increment the points counter.
Let's put this to the test by taking 4 `$` and combining it with 6 `!` characters - order doesn't matter: `$$$$!!!!!!` and plugging it into the *Portable Executable* (*PE*).

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/zsombii - Easy crackme/binary
(main)
$ java -jar ./easycrack.jar
Thanks for downloading this product. Enter the validation key here to complete,
or exit with ENTER: $$$$!!!!!!
Validation key accepted

david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/zsombii - Easy crackme/binary
(main)
$ java -jar ./easycrack.jar
Thanks for downloading this product. Enter the validation key here to complete,
or exit with ENTER: $!$!$!$!!!
Validation key accepted

david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/zsombii - Easy crackme/binary
(main)
$ java -jar ./easycrack.jar
Thanks for downloading this product. Enter the validation key here to complete,
or exit with ENTER: !!!!!!$$$$
Validation key accepted
```

As proven, any combination of the aforementioned would result in a valid key.

---

7. Standard Printable ASCII Set

This is the standard printable *ASCII* set (`32` to `126`). Basically letters, digits, and the common keyboard symbols on a US layout.

Char	ASCII (dec)	ASCII (hex)	Binary (8-bit)
SP	32	0x20	00100000
!	33	0x21	00100001
"	34	0x22	00100010
#	35	0x23	00100011
$	36	0x24	00100100
%	37	0x25	00100101
&	38	0x26	00100110
'	39	0x27	00100111
(	40	0x28	00101000

Char	ASCII (dec)	ASCII (hex)	Binary (8-bit)
)	41	0x29	00101001
*	42	0x2A	00101010
+	43	0x2B	00101011
,	44	0x2C	00101100
-	45	0x2D	00101101
.	46	0x2E	00101110
/	47	0x2F	00101111
0	48	0x30	00110000
1	49	0x31	00110001
2	50	0x32	00110010
3	51	0x33	00110011
4	52	0x34	00110100
5	53	0x35	00110101
6	54	0x36	00110110
7	55	0x37	00110111
8	56	0x38	00111000
9	57	0x39	00111001
:	58	0x3A	00111010
;	59	0x3B	00111011
<	60	0x3C	00111100
=	61	0x3D	00111101
>	62	0x3E	00111110

Char	ASCII (dec)	ASCII (hex)	Binary (8-bit)
?	63	0x3F	00111111
@	64	0x40	01000000
A	65	0x41	01000001
B	66	0x42	01000010
C	67	0x43	01000011
D	68	0x44	01000100
E	69	0x45	01000101
F	70	0x46	01000110
G	71	0x47	01000111
H	72	0x48	01001000
I	73	0x49	01001001
J	74	0x4A	01001010
K	75	0x4B	01001011
L	76	0x4C	01001100
M	77	0x4D	01001101
N	78	0x4E	01001110
O	79	0x4F	01001111
P	80	0x50	01010000
Q	81	0x51	01010001
R	82	0x52	01010010
S	83	0x53	01010011
T	84	0x54	01010100

Char	ASCII (dec)	ASCII (hex)	Binary (8-bit)
`U`	85	0x55	01010101
`V`	86	0x56	01010110
`W`	87	0x57	01010111
`X`	88	0x58	01011000
`Y`	89	0x59	01011001
`Z`	90	0x5A	01011010
`[`	91	0x5B	01011011
`\\`	92	0x5C	01011100
`]`	93	0x5D	01011101
`^`	94	0x5E	01011110
`_`	95	0x5F	01011111
``	96	0x60	01100000
`a`	97	0x61	01100001
`b`	98	0x62	01100010
`c`	99	0x63	01100011
`d`	100	0x64	01100100
`e`	101	0x65	01100101
`f`	102	0x66	01100110
`g`	103	0x67	01100111
`h`	104	0x68	01101000
`i`	105	0x69	01101001
`j`	106	0x6A	01101010

Char	ASCII (dec)	ASCII (hex)	Binary (8-bit)
k	107	0x6B	01101011
l	108	0x6C	01101100
m	109	0x6D	01101101
n	110	0x6E	01101110
o	111	0x6F	01101111
p	112	0x70	01110000
q	113	0x71	01110001
r	114	0x72	01110010
s	115	0x73	01110011
t	116	0x74	01110100
u	117	0x75	01110101
v	118	0x76	01110110
w	119	0x77	01110111
x	120	0x78	01111000
y	121	0x79	01111001
z	122	0x7A	01111010
{	123	0x7B	01111011
|	124	0x7C	01111100
}	125	0x7D	01111101
~	126	0x7E	01111110

8. Challenging Myself

Since this crackme essentially exposed it's source code, reverse engineering it wasn't too difficult of a challenge. To increase the difficulty, I thought it would be neat to create a *Python* script that would generate a valid keys - a *keygen*.

Rewriting the validation function in *Python*:

```
def is_valid(key: str) -> bool:
    points = 0
    for chr in key:
        points += 1 if (ord(chr) & 3) == 0 else 0
    return points == 4
```

I define the character set that I will be using for my key generator:

```
printable_ascii = "".join(chr(c) for c in range(32, 127))
```

This will loop over the numbers *32* to *127* (exclusive), convert it to a character, and append it to `printable_ascii`. We don't include *127* as `del` is not a printable character.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

I next create two lists from this `printable_ascii` string. One with *good characters - characters that will increment the counter by *one*. And one with *bad characters - characters that will ***NOT*** increment the counter.
This is done by looping over each character within `printable_ascii` and performing the validation check on it (could also perform a modulo 4). If the character passes the check, it gets added to the good character list and vice versa.

```
def generate_good_and_bad_character_lists(char_range):
    good_characters, bad_characters = [], []
    for ch in char_range:
        if (ord(ch) & 3) == 0:
            good_characters.append(ch)
            continue
        bad_characters.append(ch)

    return good_characters, bad_characters
```

With that done, generating the key is just choosing 4 characters from the `good_characters` list and 6 characters from the `bad_characters` list. Utilizing the `random` Python library makes this a breeze.

```
def generate_key(len, good_characters, bad_characters) -> str:
    rng = random.Random()
    key = rng.sample(good_characters, 4) + rng.sample(bad_characters, 6)
    random.shuffle(key)

    return ''.join(key)
```

Alright, let's put it all together and test it out!

```
import random

KEY_TOTAL_VALUE = 4


def is_valid(key: str) -> bool:
    points = 0
    for chr in key:
        points += 1 if (ord(chr) & 3) == 0 else 0
    return points == 4


def generate_good_and_bad_character_lists(char_range):
    good_characters, bad_characters = [], []
    for chr in char_range:
        if (ord(chr) & 3) == 0:
            good_characters.append(chr)
            continue
        bad_characters.append(chr)

    return good_characters, bad_characters


def generate_key(len, good_characters, bad_characters) -> str:
    rng = random.Random()
    key = rng.sample(good_characters, 4) + rng.sample(bad_characters, 6)
    random.shuffle(key)

    return ''.join(key)
```

```
def main():
    printable_ascii = "".join(chr(c) for c in range(32, 127))
    good_characters, bad_characters =
generate_good_and_bad_character_lists(printable_ascii)

    key = generate_key(len, good_characters, bad_characters)
    print(f"[+] Key Wrapped in Quotes: \"{key}\"")


if __name__ == '__main__':
    main()
```

```
$ py ./keygen.py
[+] Key Wrapped in Quotes: "x4{C,+h)e6"
```

```
david@Kyfe_Station MINGW64 ~/Desktop/crackmes.one/zsombii - Easy cracme/binary (main)
$ java -jar ./easycrack.jar
Thanks for downloading this product. Enter the validation key here to complete, or exit with ENTER: x4{C,+h)e6
Validation key accepted
```

Success!
I then proceeded to update the script to accept command line arguements as well as have the ability to generate multiple keys at once.

8.1 Afterthoughts

Currently the *keygen* implementation generates the character samples with `rng.sample(...)` which picks *unique* characters - no repeats. But, the crackme logic allows repeats. There's no all characters must be unique rule or validation logic. A better choice here would be to use `random.choices()` so repeats are allowed. This doesn't change correctness, but it makes the *keygen* logically match the validator.

---

9. Conclusion

This was a nice and simple *keygen* exercise that added new tools to my kit as well as give me experience with one, *JADX*. Due to the nature of the validation function, the possibilities for valid keys is quite enormous. For normal keyboard printable *ASCII* there are ***95*** characters (ASCII *32–126*).

- g = *24 are divisible by 4*
- b = *71 are NOT divisible by 4*

$$\binom{10}{4} \cdot 24^4 \cdot 71^6 = 8,925,125,957,616,476,160$$

≈ **8.93 × 10[18]** valid keys. Which equates to `8,925,125,957,616,476,160` total number of possible valid keys.

This crackme was a great exercise in recognizing when my usual native *RE* workflow doesn't apply and switching toolchains appropriately. Once *JADX* produced readable *Java*, the entire challenge collapsed into a single scoring rule: for a *10*-character key, exactly *4* characters must have an *ASCII* value divisible by *4* (IE: the last two bits are `00`). Everything else in the function is either boilerplate or deliberate noise (the `startsWith("KEY")` branch is dead code and has no effect on the result).

Even though this one didn't require runtime debugging, it reinforced an important habit: treat decompiler output as a *hypothesis* until proven, and keep tools like *IntelliJ/JDWP* in the back pocket for cases where obfuscation or control-flow tricks make the decompile unreliable. To push the challenge further, I implemented a *Python* key generator that mirrors the validator and can produce valid keys on demand, which also helped validate my understanding of the bitmask/modulo relationship.