

Unicode as low-level attack primitive



Alexandre ZANNI (@noraj)

11/09/2025

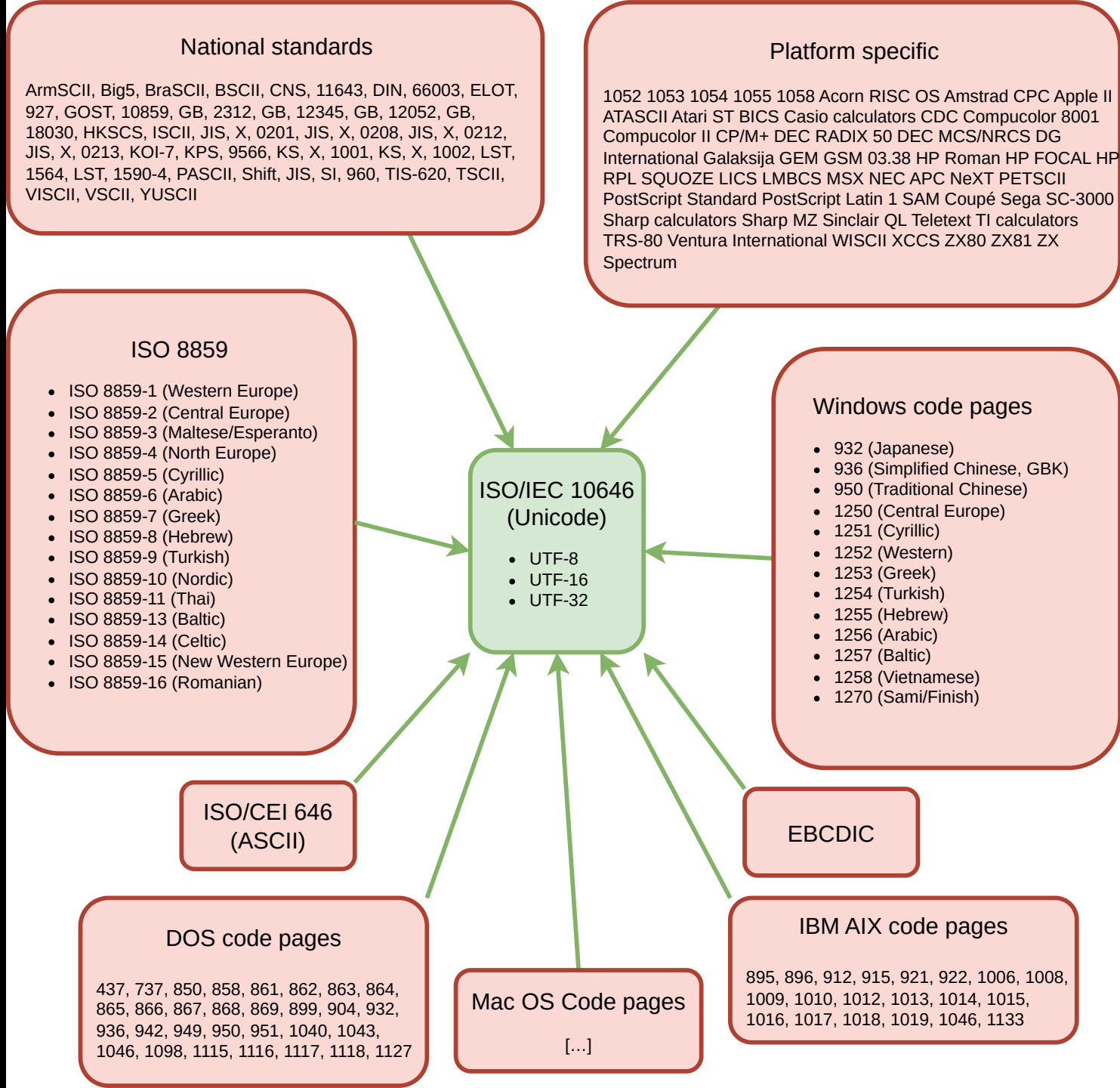
Whoami

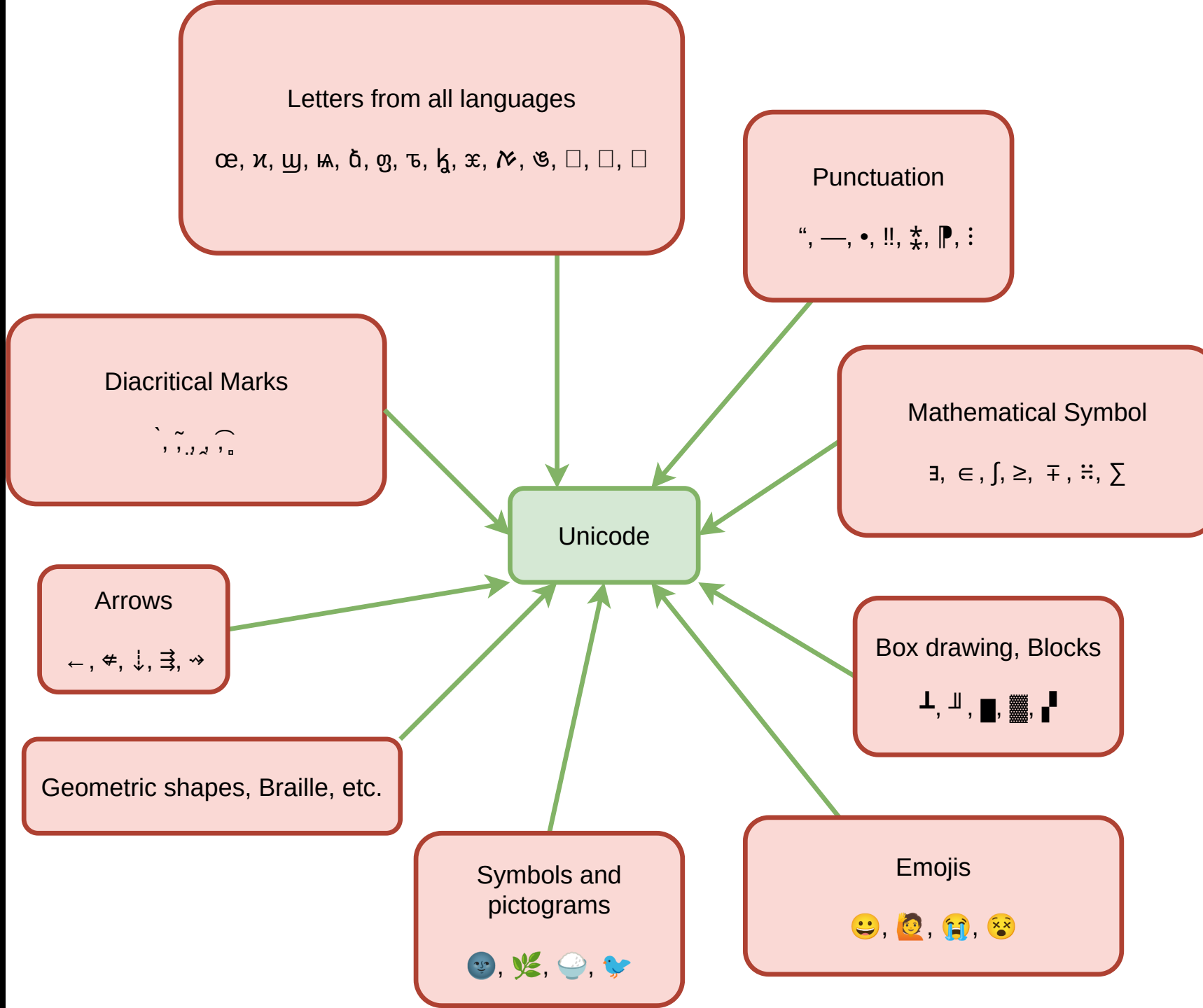
Alexandre ZANNI a.k.a **noraj**

Pentester @**Synacktiv**

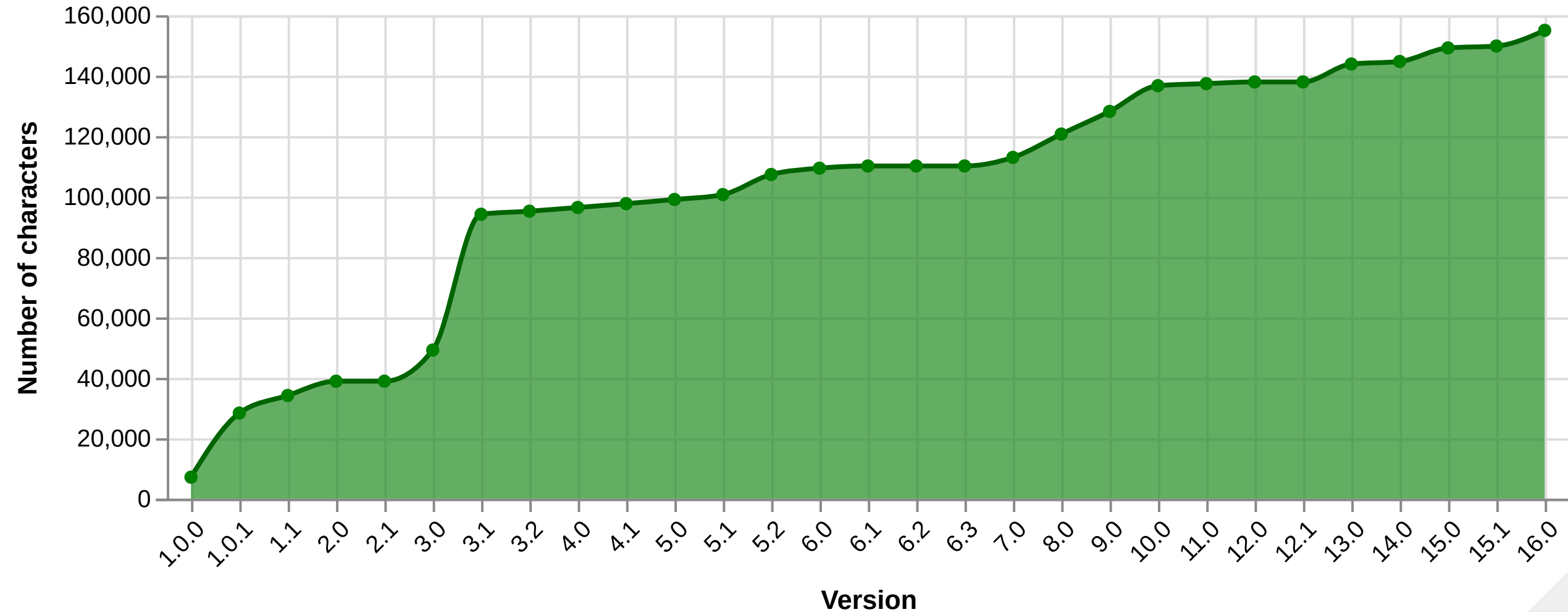
Maintainer of **BlackArch** Linux

Introduction To Unicode





Number of Unicode characters per version.



- **Code points:** numerical values from 0 (0x0) to 1114111 (0x10FFFF), ~ 1.1M values.
- This range is named the **codespace**.
- Those values are denoted as `U+<hex_code_point>`
- Code points are the unique **identifier** or **reference** number of a Unicode character.

Format	Bits	Encoding	Mapping
Byte	8	UTF-8	variable (1 \Rightarrow 1 to 4)
Word	16	UTF-16	variable (1 \Rightarrow 1 to 2)
Double-Word	32	UTF-32	fixed (1 \Rightarrow 1)




Composing




precomposed char vs composite char




```
"\u00E9" # => "é"  
"e\u0301" # => "é"  
"\u00E9" == "e\u0301" # => false
```

Joining

- modifying character shape with **variation selectors** or **modifiers**
- joining characters with **joiners**

 +  =  (`\u{1F44C}\u{1F3FB}`)

 +  =  (`\u{2764}\uFE0F\u200D\u{1F525}`)

 +  =  (`\u{1F3F4}\u200D\u2620\uFE0F`)

Low-Level Attack Primitive

Where do issues come from?

- Implementation errors
- Complexity of Unicode mechanisms
- Lack of awareness

Homoglyphs

Glyphs that look like each other but which are visually different by a tiny bit while there are totally different characters logically.

Depends on the font used.



Latin Small Letter A
Unicode U+0061



Mathematical Sans-Serif Small A
Unicode U+1D5BA

Both set in Helvetica Neue



U+0061 LATIN SMALL LETTER A



U+0430 CYRILLIC SMALL LETTER A

Both glyphs set in Helvetica LT Std Roman at identical weight, size, and baseline.

Visual attacks based on homoglyph / confusion?

- Internationalized Domain Names (IDN)
- Font specific attacks
 - Inadequate Rendering Support
 - Missing Glyphs
- Source code backdooring

Example of source code backdooring from Certitude

```
const [ ENV_PROD, ENV_DEV ] = [ 'PRODUCTION', 'DEVELOPMENT' ];  
/* ... */  
const environment = 'PRODUCTION';  
/* ... */  
function isUserAdmin(user) {  
    if(environment !== ENV_PROD){  
        // bypass authZ checks in DEV  
        return true;  
    }  
  
    /* ... */  
    return false;  
}
```

Do you see an issue?

Sorry, in the condition (`!=`)

- it is **NOT** the negation operator `!` (EXCLAMATION MARK, U+0021)
- but actually `!` (LATIN LETTER RETROFLEX CLICK, U+01C3)

It has nothing special but can be used in identifiers.

```
-environment != ENV_PROD  
+environment! = ENV_PROD
```

Invisible Characters

They are not meant to be seen by humans such as spacing, control, filler characters.

about:newtab

no

OK

Console



Filtrer



Erreurs

Avertissements

Informations

Jo

```
>> var = alert;
```

```
← undefined
```

```
>> var yay = 'yes';
```

```
← undefined
```

```
>> if (yay === `no`) {}
```

Hangul Filler, U+3164

```
var <U+3164> = alert;
```

```
var yay = 'yes'
```

```
if (yay === <U+3164> `no`) {}  
// => alert
```

Visual attacks based on invisible characters?

- Source code backdooring
 - Reordering
 - Early Returns
 - Commenting-Out
 - Stretched Strings
 - Isolate Shuffling
 - Whitespaces

Size

What the size of 🧑🧑?

1? 6? 8? 16? 20?

The size of what?



code points: U+1F469 U+200D U+2764 U+FE0F U+200D
U+1F468

- Graphemes size: 1
- String size / code unit count: 6
- Byte size:
 - UTF-8: **20 bytes**
 - Hexadecimal: f0 9f 91 a9 e2 80 8d e2 9d a4 ef b8 8f e2 80 8d f0 9f 91 a8
 - UTF-16 (without BOM): **16 bytes**
 - Big Endian: d83d dc69 200d 2764 fe0f 200d d83d dc68
 - Little Endian: 3dd8 69dc 0d20 6427 0ffe 0d20 3dd8 68dc

- Byte size:
 - UTF-32 (without BOM): **24 bytes**
 - Big Endian: 0001f469 0000200d 00002764
0000fe0f 0000200d 0001f468
 - Little Endian: 69f40100 0d200000 64270000
0ffe0000 0d200000 68f40100


```
// nodejs - javascript - UTF-16  
a.length // 8, maybe because of surrogates  
  
[...a].length // 6  
  
// Return 20 instead of 16  
// nodejs  
Buffer.byteLength(a, 'utf16le') // 16  
Buffer.byteLength(a, 'utf16be') // 20  
// javascript (UTF-8)  
(new TextEncoder()).encode(a).length // 20  
  
[...new Intl.Segmenter().segment(a)].length // 1
```

```
a = '👩'  
a.size # 6 (number of code points)  
a.bytesize # 20 (number of bytes in UTF-8)  
a.grapheme_clusters.size # 1 (number of graphemes)
```

Impacts?

- Buffer overflow
- String truncation
- Wrong index access
- Sanitizing misplaced

Case Transformation

- ~~bijection: $1 \Rightarrow 1$~~
- $1 \Rightarrow n$
- $n \Rightarrow 1$
- contextual (depending on what is placed around it)
- local-sensitive (regional settings)

Can result in:

- Changing the string length
- Creating collisions
- Entries mismatch
- Security bypasses

Case transformation collisions (uppercase)

```
a = 'ß'  
len(a) # 1  
a.upper() # SS  
len(a.upper()) # 2
```

ß (LATIN SMALL LETTER SHARP S, U+00DF)



2 × S (LATIN CAPITAL LETTER S, U+0053).

Case transformation collisions (lowercase)

```
kelvin = 'K'  
k = 'K'  
kelvin == k # False  
kelvin.lower() == k.lower() # True
```

K (KELVIN SIGN, U+212A)



k (LATIN SMALL LETTER K, U+006B) when converted to lowercase.

Contextual transformation (context-sensitive case mapping)

Σ (GREEK CAPITAL LETTER SIGMA, U+03A3)

Σ	⇒	σ
Σa	⇒	σa
aΣ	⇒	aζ
ΣΣ	⇒	σζ

- σ (GREEK SMALL LETTER SIGMA, U+03C3) if the letter is followed by a cased letter (or alone)
 - ζ (GREEK SMALL LETTER FINAL SIGMA, U+03C2)
- else

Language-sensitive transformation (language-sensitive case mappings)

```
"I".toLocaleLowerCase("fr-FR")  
// 'i'  
"I".toLocaleLowerCase("az-AZ")  
// 'ı'  
"i".toLocaleUpperCase("fr-FR")  
// 'I'  
"i".toLocaleUpperCase("tr-TR")  
// 'İ'
```

Example of account takeover via password reset using case transformation collisions (lowercase)

```

app.post('/api/password/reset', function(req, res) {
  var email = req.body.email;
  db.get('SELECT id, email, FROM users WHERE email = ?',
    [email.toLowerCase()],
    (err, user) => {
      if (err) {
        console.error(err.message);
        res.status(400).send();
      } else {
        generateTemporaryPassword((tempPassword) => {
          accountRepository.resetPassword(user.id, tempPassword, () => {
            messenger.sendPasswordResetEmail(email, tempPassword);
            res.status(204).send();
          });
        });
      }
    });
});
});

```



```
email.toLowerCase()
```


```
Admin@synacktiv.com ⇒ admin@synacktiv.com
```

```
> a = "K";  
'K'  
> b = "K";  
'K'  
> a == b  
false  
> a.toLowerCase() == b.toLowerCase()  
true
```



```
email.toLowerCase()
```

```
admin@synacktiv.com ⇒ admin@synacktiv.com
```

- Select `email.toLowerCase()` in DB
 - Transformed user input
 - The legit user email is selected in DB (and so it's password reset token)
- Send the email to `email` 
 - Unmodified user input



Implicit Unicode behaviors in database string functions

Article (p. 73) in Paged Out!
magazine n°6

Another example for MariaDB /
MySQL

Normalization

Canonical vs Compatible

Form	Description
Normalization Form D (NFD)	Canonical Decomposition
Normalization Form C (NFC)	Canonical Decomposition, followed by Canonical Composition
Normalization Form KD (NFKD)	Compatibility Decomposition
Normalization Form KC (NFKC)	Compatibility Decomposition, followed by Canonical Composition

Source		NFD		NFC		NFKD		NFKC
fi	:	fi		fi		f i		f i
FB01		FB01		FB01		0066 0069		0066 0069
2 ⁵	:	2 5		2 5		2 5		2 5
0032 2075		0032 2075		0032 2075		0032 0035		0032 0035
fi	:	f ̇ ̇		fi ̇		s ̇ ̇		š
1E9B 0323		017F 0323 0307		1E9B 0323		0073 0323 0307		1E69

2 examples of attacks using normalization

- Host splitting
- XSS filter bypass

Host splitting

Abusing compatible normalization modes to bypass
URL whitelist

Technique presented by *Jonathan Birch* at *Black Hat
USA 2019* (*Host/Split
Exploitable Antipatterns in Unicode Normalization*)

With NFKD and NFKC
Addressed To the Subject

$\text{a/s (U+2101)} \Rightarrow \text{a (U+0061) + / (U+002F) + s (U+0073)}$

$\text{a/s} \Rightarrow \text{a/s}$

```
url = 'https://synacktiv.c%u00e5support.target.com'  
url.unicode_normalize('nfkd')  
# => "https://synacktiv.ca/support.target.com"  
url.unicode_normalize('nfkc')  
# => "https://synacktiv.ca/support.target.com"
```

Will bypass *.target.com and lead to
synacktiv.ca

The attacker just need to register a domain with a TLD that ends by `a` in that case: `.ca` , `.media` , `.ninja` , `.pizza` , `.mba` , `.moda`

Similar splitters:

- $\%_c$ (U+2100)
- $\%$ (U+2105)
- $\%_u$ (U+2106)

More characters to play with URLs:

- `?` (U+2048) \Rightarrow `?` , GET parameters
- `□` (U+FF0F) \Rightarrow `/` , like `%` but no constraint 🤪
- `□` (U+FF03) \Rightarrow `#` , neutralize stuff as anchor
- `□` (U+FF20) \Rightarrow `@` , username
- `(user@cdn.target.com)`

- `:` (U+FF1A) \Rightarrow `:`, credentials
- `(user:pass@secret.target.com)`
- `1.` (U+2488) \Rightarrow `1.` crafting IP addresses

Use cases:

- URL parsers
- Validation with regexp
- Web browser or server `Location` HTTP header & redirecting to normalized input

XSS filter bypass

HTML escaping functions will escape HTML special characters like `<`, `>`, `"`, `&`, etc.

With NFKD and NFKC

Bypass $<$ (U+003C) with:

- \square (U+FE64)
- \square (U+FF1C)

Bypass $>$ with:

- \square (U+FE65)
- \square (U+FF1E)

Bypass " (U+0022) with:

- □ (U+FF02)

Bypass & (U+0026) with:

- □ (U+FE60)
- □ (U+FF06)

```
test = '◻' # U+FE64
test.unicode_normalize(:nfkc) # => "<" (U+003C)
test.unicode_normalize(:nfkd) # => "<" (U+003C)
test = '◻' # U+FF1E
test.unicode_normalize(:nfkc) # => ">" (U+003E)
test.unicode_normalize(:nfkd) # => ">" (U+003E)
```



```
▯script▯alert(document.cookie)▯/script▯
```

↓ (NFKC, NFKD) + HTML escape

```
<script>alert(document.cookie)</script>
```

instead of

```
&lt;script&gt;alert(document.cookie)&lt;/scrip  
t&gt;
```

Use cases:

- Incorrect ordering of filtering steps by the developer.
- Normalisation performed implicitly by the DBMS framework

Example: In all application that are doing

```
normalizeNFKC(escapeHTML(user_param))
```

instead of

```
escapeHTML(normalizeNFKC(user_param))
```

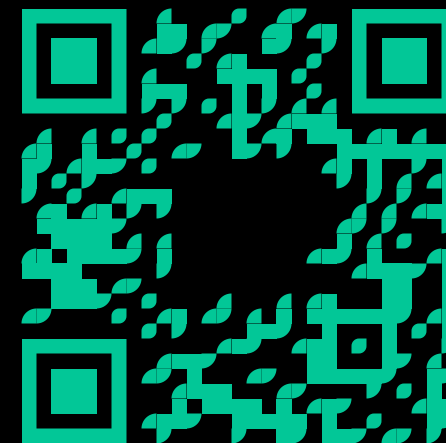
With NFC

> followed by U+0338 will be recomposed into >/

⇒ cancel the end of a HTML tag

```
<textarea id=desc>INJECTION_HERE</textarea>  
<!-- ↓ injection -->  
<textarea id=desc>U+0338 autofocus onfocus=alert(document.cookie) </textarea>  
<!-- ↓ normalized -->  
<textarea id=desc>/ autofocus onfocus=alert(document.cookie) </textarea>
```

Thank you for your attention



 **SYNACKTIV**