

Code Reliability

Security.any #08 目指しているセキュリティLT

2026.01.27

RYO

Hi!! 

颅 SRE / Security Engineer / 説得係

某人材系企業にて転職支援・案件紹介サービスの開発に従事

書 主にAWS環境のセキュリティ品質の改善や雑務・雑用を担当

電 好きな技術は   

車 趣味はドライブ・ランニング・Webサイト構築・英語学習など...

娃 でも子供が生まれてからは自分の時間をあまり持ててないのが悩み...



👉 自分はどちらかと言えば好きな方

コードを書くことは好きですか？

👉 でも得意ではない（上手くない）



The era of humans writing code is over

Node.jsの作者である Ryan Dahl 氏も「人間がコードを書く時代は終わった」と明言している。
コードはAIに任せればいい時代...？

Ryan Dahl (@rough_sea) posted a tweet on January 20, 2026, at 1:02 AM. The tweet reads: "This has been said a thousand times before, but allow me to add my own voice: the era of humans writing code is over. Disturbing for those of us who identify as SWEs, but no less true. That's not to say SWEs don't have work to do, but writing syntax directly is not it." The post has received 949 replies, 3.9K retweets, 19K likes, 6.1K bookmarks, and 1.1K shares.

This has been said a thousand times before, but allow me to add my own voice: the era of humans writing code is over. Disturbing for those of us who identify as SWEs, but no less true. That's not to say SWEs don't have work to do, but writing syntax directly is not it.

1:02 AM · Jan 20, 2026 · 7M Views

949 3.9K 19K 6.1K 1.1K

コード、どれだけ理解していますか？ 

Quiz

これからTypeScriptで実装されたif文を含むコードを元にいくつかクイズを出します。

どうすれば条件式が `true` を返すか（AIに極力頼らずに）考えてみてください。

コードの理解があれば回答できるはずです。

例題です。次のコードで  `success!!` を出力させるには、`example()` に渡す最小の整数は何でしょう？

```
const example = (num: number) => {
  if (num === num + 1) {
    console.log(img alt="blue flower icon" data-bbox="228 428 248 448"/> success!!)
  } else {
    console.log("X failed!!")
  }
}

example(?)
```



X failed!!

Question 1

Question 1

次のコードで  success!! を出力させるには、`question()` に渡す文字列は何でしょう？
"A" 以外にも正解があります。

```
const question = (str: string) => {
  if ("A" === str) {
    console.log(img alt="blue flower icon" data-bbox="145 355 165 375"/> success!!")
  } else {
    console.log("✖ failed!!")
  }
}

question(?)
```

✖ failed!!

Answer

Answer

"\u0041" , "\u{41}" , "\x41" などが正解。

```
console.log("A" === "\u0041") // Unicodeエスケープ
console.log("A" === "\u{41}") // Unicodeコードポイント
console.log("A" === "\x41") // 16進数エスケープ
```

```
true
true
true
```

Unicodeは1つの文字を複数の異なる文字で表現したり、見えない文字（制御文字）を表現できる。
世の中にはUnicodeを悪用した攻撃がいくつか存在する。

Homograph Attacks

ホモグラフ攻撃 (Homograph Attacks) は視覚的に似ているUnicode文字を悪用して、偽のドメイン名を取得する手法です。ユーザを騙して偽のウェブサイトに誘導したりフィッシング詐欺に利用される。

次のURLはどちらが本物でしょうか？

- `https://www.google.com/`
- `https://www.google.com/`

上が本物、下が偽物です。

RLO Attacks

RLO攻撃（RLO Attacks）はUnicode特殊文字を悪用してファイル改ざんを行います。
RLO = Right-to-Left Override のことで、文字を逆転させる。

同じディレクトリ内に同じ名前のPDFがあります。どちらが本物のPDFですか？

名前	種類	サイズ	変更日
sample_exe.pdf	PDF書類	26 バイト	2026/01/09 12:41
sample_exe.pdf	PDF書類	26 バイト	2026/01/09 12:42

「詳細を見る」と本物がわかる。

The image shows two side-by-side file information windows for 'sample_exe.pdf' and 'sample_fdp.exe'. Both files are 26 bytes and were modified on January 9, 2026, at 12:41. The first file is a PDF document, while the second is an executable file.

Left Window (sample_exe.pdf):

- General Information:** Type: PDF Document, Size: 26 bytes (4 KB), Location: Macintosh HD • User → ryo.matsunami • Downloads, Created: 2026年1月9日 金曜日 12:41, Modified: 2026年1月9日 金曜日 12:41.
 - ひな形
 - ロック
- Detailed Information:** Last opened: 2026年1月9日 金曜日 12:51.
- Name and Extension:** Name and extension: sample_exe.pdf
 - Extension not shown
- Comments:** None.
- Preview:** Shows a thumbnail of a beach scene with a small bottle.

Right Window (sample_fdp.exe):

- General Information:** Type: PDF Document, Size: 26 bytes (4 KB), Location: Macintosh HD • User → ryo.matsunami • Downloads, Created: 2026年1月9日 金曜日 12:42, Modified: 2026年1月9日 金曜日 12:42.
 - ひな形
 - ロック
- Detailed Information:** Last opened: 2026年1月9日 金曜日 12:51.
- Name and Extension:** Name and extension: sample_fdp.exe
 - Extension not shown
- Comments:** None.
- Preview:** Shows a thumbnail of a beach scene with a small bottle.

Unicode Normalization Attacks

Unicode正規化攻撃（Unicode Normalization Attack）はUnicodeを巧妙に使い分けて、アプリケーションの文字列比較やフィルタリング処理をすり抜けて不正な入力を通す手法です。

タグの有無を判定するバリデーションチェックを例に、脆弱な実装と適切な実装を比べてみます。

```
1 // 改善後
2 const validation = (str: string) => {
3   // 正規化
4   const s = str.normalize("NFKC") // s = "<script>"
5   // < > を含む文字列はNGとするチェック
6   const pattern: RegExp = /[<>]/;
7   if (pattern.test(s)) {
8     throw new Error("✖ failed!!")
9   } else {
10     console.log("✅ success!!")
11   }
12
13   // ...
14 };
15
16 validation("\uFE64" + "script" + "\uFE65")
```

Question 2

Question 2

次のコードで 🎉 success!! を出力させるには、`/* add 1 code here */` の部分に何と実装すればいいでしょうか？

```
const question = () => {
  const target = []
  if (target.isAdmin) {
    console.log("🎉 success!!")
  } else {
    console.log("✖ Failed!!")
  }
}

/* add 1 code here */

question()
```

✖ Failed!!

Answer

Answer

```
Object.prototype.isAdmin = true、({}).__proto__.isAdmin = true、  
Object.defineProperty(Object.prototype, "isAdmin", { value: true }) などが正解。
```

```
const answer = () => {  
  | console.log({}.isUser)  
};  
  
answer() // 汚染前  
Object.defineProperty(Object.prototype, "isUser", { value: true })  
answer() // 汚染後
```

```
undefined  
true
```

オブジェクトはプロパティに目的のキーがない場合、親である `Object.prototype` を参照します。

全てのオブジェクトは親を継承しているため、たとえ空オブジェクトであっても親を汚染すれば子も影響を受けます。

Prototype Pollution

プロトタイプ汚染（Prototype Pollution）は、JavaScriptにおいて発生する有名な脆弱性の1つです。

攻撃者はPrototypeを改ざんしてXSS、リモートコード実行、権限昇格、パストラバーサルなどの攻撃を試みます。

Node.jsやlodashなどJavaScript製のツールで過去にプロトタイプ汚染による脆弱性が何度か見つかっています。

Vulnerability Details : [CVE-2022-21824](#)

Node.js console.table() Prototype Pollution via User-Controlled Properties Parameter

Due to the formatting logic of the "console.table()" function it was not safe to allow user controlled input to be passed to the "properties" parameter while simultaneously passing a plain object with at least one property as the first parameter, which could be "`__proto__`". The prototype pollution has very limited control, in that it only allows an empty string to be assigned to numerical keys of the object prototype. Node.js >= 12.22.9, >= 14.18.3, >= 16.13.2, and >= 17.3.1 use a null prototype for the object these properties are being assigned to.

Published 2022-02-24 19:15:10 Updated 2022-11-10 03:48:28 Source [HackerOne](#)

View at [NVD](#), [CVE.org](#), [EUVD](#)

出典 : [CVEdetails.com](#)

Prototype Pollution

URLのクエリパラメータを解析する際に、不正にブラケット記法が送られてきた場合、プロトタイプ汚染に繋がるリスクがある。

```
1 import { qs } from "qs";
2
3 const query = "?name=test3&__proto__[isAdmin]=true"; // 汚染されるケース
4 const parsed = qs.parse(query)
5 console.log(parsed) // { name: 'test3' }
6
7 console.log({}.isAdmin) // true
```

Monkey Patch

モンキーパッチ（Monkey Patch）は、プログラムをその時その場の実行範囲内で拡張または修正するというテクニックです。（これ自体は脆弱性ではない。）

ただ、使いすぎるとコードの予測が困難で保守性を低下させるリスクがあり、Node の セキュリティのベストプラクティス でも非推奨となっている。

```
Array.prototype.push = function (item) {
  // overriding the global [].push
};
```

プロトタイプ汚染を防ぐには、

- 🤖 オブジェクトをマージする際は、`__proto__`、`constructor`、`prototype` の利用を防止する
- 🔧 `Object.create(null)` で親を持たないオブジェクトを生成する
- 🔑 `Object.freeze(Object.prototype)` で `prototype` 自体をロックする（効果絶大だが破壊的副作用も大きい）

などの方法がある。

Question 3

Question 3

次のコードで  `success!!` を出力させるには、`question()` に渡す数値は何でしよう？

```
const question = (num: number) => {
  const value = 0.1 + 0.2
  if (value === num) {
    console.log(img alt="sparkling success icon" data-bbox="145 185 165 205"/> success!!)
  } else {
    console.log("✖ Failed!!")
  }
}

question(?)
```

✖ Failed!!

Answer

Answer

0.3000000000000004、0.30000000000000039、0.300000000000000444などが正解。

```
const sum = 0.1 + 0.2;  
console.log(sum === 0.3000000000000004)  
console.log(sum === 0.300000000000000444)
```



```
true  
true
```

算数の世界では $0.1 + 0.2$ は 0.3 ですが、ほとんどのプログラミング言語が採用している IEEE 754（浮動小数点数）の世界では話が変わります。

世の中には、浮動小数点数の仕組みを悪用した攻撃や脆弱性がある。

Subnormal Number

非正規化数（Subnormal Number）は、ゼロに極めて近いがゼロではない表現可能な最小の数よりも小さい数のことです。

2.225×10^{-308} 未満の値が該当する。

演算に使われると通常の数値に比べて処理時間が大幅に長くなる傾向があります。

非正規化数は浮動小数点の一部であり、これ自体は脆弱性ではないがパフォーマンス低下を招く可能性があります。

浮動小数点数による不具合を回避するには、

-  `decimal.js`、`bignumber.js` などのライブラリの活用
-  微細な誤差であれば、`Number.EPSILON`を使って浮動小数点数の等価性を検証する
 - `Math.abs(0.1 + 0.2 - 0.3) < Number.EPSILON // true`

などが有効。

Question 4

Question 4

次のコードで 🎉 success!! を出力させるには、`/* add 1 code here */` の部分に何と実装すればいいでしょうか？

```
const REG_CHECK = /test/g

const question = (str: string) => {
  if (str === "test" && !REG_CHECK.test(str)) {
    console.log("🎉 success!!")
  } else {
    console.log("✖ Failed!!")
  }
}

question("test")
/* add 1 code here */
```

✖ Failed!!

Answer

Answer

question("test") が正解。

```
const REG_CHECK = /test/g

const question = (str: string) => {
  if (str === "test" && !REG_CHECK.test(str)) {
    console.log("🎉 success!!")
  } else {
    console.log("✖ Failed!!")
  }
}

question("test")
question("test")
```

✖ Failed!!
🎉 success!!

これは実装されている正規表現に脆弱性があり、呼び出す度に結果が変わります。
正規表現は使いこなせば便利ですが、バグや脆弱性の原因にもなりやすい。

g

`const REG_CHECK = /test/g` のように「`g`」を付けると、ステートフルな正規表現となってしまう。
具体的には `lastIndex` という「照合を開始する位置」をこっそり保持することになります。

```
1 const REG_CHECK = /test/g
2 const str = "test for test and test"
3
4 REG_CHECK.test(str)
5 console.log(REG_CHECK.lastIndex) // 4
6
7 REG_CHECK.test(str)
8 console.log(REG_CHECK.lastIndex) // 13
9
10 REG_CHECK.test(str)
11 console.log(REG_CHECK.lastIndex) // 22
12
13 // 4回目：
14 REG_CHECK.test(str) // 22文字目から検索する -> ""
15 console.log(REG_CHECK.lastIndex) // 0 (ここでfalseとなる)
```

Regular expression Denial of Service (ReDoS)

ReDoS（レドス）は正規表現の処理の仕組みを悪用し、計算リソースを大量に消費させてサービス停止や遅延を引き起こす攻撃です。

主に正規表現にマッチしない入力をされた場合に発生し、再度ひとつ戻って文字を照合するというバックトラックの仕組みを悪用します。

```
const REG_CHECK = /^[([a-zA-Z0-9])++$/
```



```
// 計算時間を計測するサンプル
const example = (str: string) => {
  const start = performance.now()
  REG_CHECK.test(str)
  const end = performance.now()

  console.log(` ${(end - start).toFixed(2)} ms`)
}
```

```
example("abcdef123456@") // 文字を増やすと計算時間が増える
```

1.20 ms

Regular expression Denial of Service (ReDoS)

`/^(([a-zA-Z0-9])+)+$/` のように、パッと見ただけでは問題があるかどうかわかりにくい。

正規表現を悪用した攻撃を防ぐには、

-  `/(a+)+$/` や `/(a|b+)+$/` のような「繰り返しの中に繰り返し」がある構造を避ける
 - $O(2^n)$ の計算量を招く恐れがある
 -  入力可能な文字数を制限する
 -  node-re2などの正規表現用のライブラリを活用する

などの方法がある。

Summary

Summary

-  ここまで Unicode、Prototype、IEEE754（浮動小数点数）、正規表現 に関する Quiz を出しましたが、皆さんは正解できましたでしょうか？
-  コードを「書く」作業のほとんどを生成AIが担うかもしれません、それでも生成AIが書いたコードを監査できるのは開発者自身です
-  そう思うと現在においても開発者がプログラミングを学ぶ価値は失っていないと考えられます
-  脆弱性を見抜ける確かな眼を持つ開発者を目指したいですね

```
process.exit(0)
```