

2020-07-15 OWASP Sendai  
Node.js の色々

---

OWASP Kansai board member  
はせがわようすけ



## 長谷川陽介 (はせがわようすけ)

---

(株)セキュアスカイ・テクノロジー 取締役CTO

[hasegawa@securesky-tech.com](mailto:hasegawa@securesky-tech.com)

<https://utf-8/jp/>

千葉大学 非常勤講師

OWASP Kansai ボードメンバー

OWASP Japan ボードメンバー

CODE BLUEカンファレンスレビューボードメンバー





# オワssp カンサイ

自分たちの直面するWebセキュリティの問題を  
自分たちの手で解決したい!

- ▶ 日本で2番目の OWASP Local Chapter
- ▶ Webセキュリティの悩み事を気楽に相談し情報共有できる場
  - ▶ スキル、役職、業種、国籍、性別、年齢に関係なし

vol.16 OWASP Kansai 森田 智彦 代表 | 地域のキーパーソンに聞く、経営課題としてのセキュリティ | 2020年サイバーセキュリティ月間企画 (近畿経済産業局)

<https://www.kansai.meti.go.jp/2-7it/k-cybersecurity-network/interview2020/kp16.html>

## OWASP API Security Top 10の翻訳

https://github.com/OWASP/www-chapter-kansai/blob/master/tab\_ja-kansai.md

### OWASP API Security Top 10(ja-kansai)

[原文\(en\)はこちら](#)

#### API1:2019 –潰れたオブジェクトレベルの認可

APIはオブジェクト識別子を処理するエンドポイントを開けっ広げにするクセがあつてな、広範囲な攻撃対象レベルのアクセス制御の問題を引き起こしよんねん。

オブジェクトレベルの認可チェックは、ユーザーからの入力を使ってデータソースにアクセスするすべての機能で考えとかなあきません。

#### API2:2019 –潰れた認証

たいがいの認証メカニズムは正しく実装されてへんので、こすい奴は認証トークンをパクったり、実装の欠陥を悪用して他のユーザーIDを一時的または恒久的に推測したりしよんねん。

クライアント/ユーザーを識別するシステムの能力を損なうと、APIのセキュリティ全体がわやになりまっせ。

#### API3:2019 –ハデなデータの露出

はやいこと実装しとて、いらちな実装者は個々の機密性を考慮せんとしてすべてのオブジェクトプロパティを公開するクセがあつてな、ユーザーに表示する前のデータフィルタリングをクライアントに丸投げすんねん。

クライアントの状態を制御せへんかったら、サーバーはますます多くのフィルターを受け取ってだな、これらのフィルターを悪用されて機密データにアクセスされんで。

#### API4:2019 -リソースの不足とレート制限

たいがいのAPIは、クライアント/ユーザーが要求できるリソースのサイズや数を制限しよらへん。

これはAPIサーバーのパフォーマンスに影響を与えるだけの一で、サービス拒否 (DoS) 攻撃につながり、ほんでもって、ブルートフォース攻撃などの認証の欠陥のドアを開いたままにしておくとちゅうこっちゃで。

たいがいの認証メカニズムは正しく実装されてへんので、こすい奴は認証トークンをパクったり、実装の欠陥を悪用して他のユーザーIDを一時的または恒久的に推測したりしよんねん。

# 今日のテーマ 「Node.jsなWebアプリのセキュリティ」



# Node.jsなWebアプリケーション

- ▶ V8 JavaScriptエンジンを採用したサーバーサイドのJS環境
  - ▶ JSは(スクリプト言語としては)速い<sup>[要出典]</sup>
- ▶ 非同期イベント駆動
  - ▶ ファイルやネットワークの入出力が非同期に行われる
- ▶ Webアプリとしては多数のクライアントを効率的に処理可能
  - ▶ リクエストごとにプロセスやスレッドが生成されない
  - ▶ 少し前に話題になったC10K問題へのひとつの解答

# Node.jsなWebアプリのセキュリティ

## ▶ OWASP Nodejs Security cheat sheet

[https://cheatsheetseries.owasp.org/cheatsheets/Nodejs\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html)

### NodeJS security cheat sheet

#### Introduction

This cheat sheet lists the things one can use when developing secure Node.js applications. Each item has a brief explanation and solution that is specific to Node.js environment.

#### Context

Node.js applications are increasing in number and they are no different from other frameworks and programming languages. Node.js applications are also prone to all kinds of web application vulnerabilities.

#### Objective

This cheat sheet aims to provide a list of best practices to follow during development of Node.js applications.

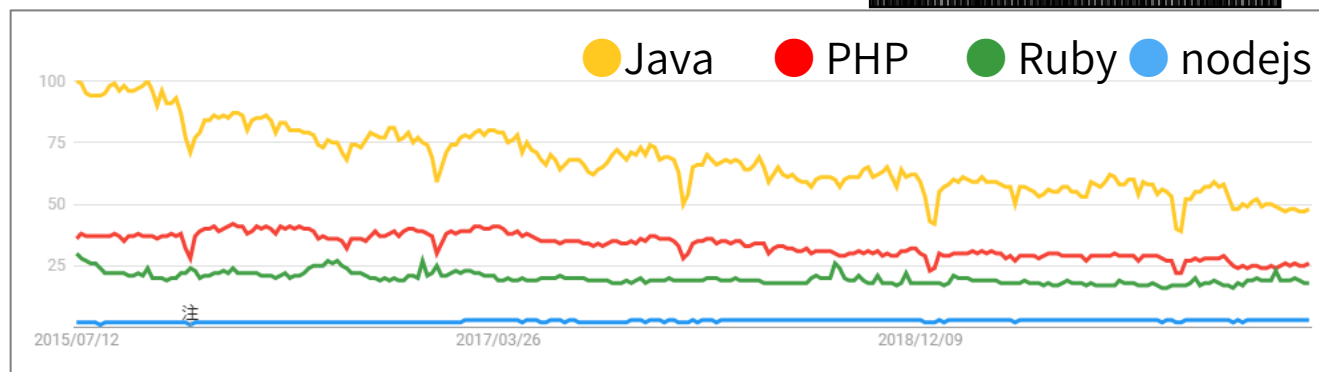
#### Recommendations

There are several different recommendations to enhance security of your Node.js applications. There are categorized as:

- **Application Security**
- **Error & Exception Handling**
- **Server Security**
- **Platform Security**

他の言語やフレームワーク同様にNode.js製アプリケーションも増加しています。**[要出典]**

誤差つ...!!  
圧倒的  
誤差つ...!!



from Google Trends, 5years, Worldwide

# OWASP Nodejs Security cheat sheet

- ▶ アプリケーションセキュリティ
  - ▶ 非同期処理はPromiseでフラットに書こう
  - ▶ リクエストのサイズを制限しよう
  - ▶ イベントループのブロックを避けよう
  - ▶ 入力の検証をきちんとしよう
  - ▶ 出力のエスケープをきちんとしよう
  - ▶ イベントループでネットワーク帯域も監視しよう
  - ▶ 総当たり攻撃に備えよう
  - ▶ CSRFトークンを使おう
  - ▶ 不要なURLルーティングを消しておこう
  - ▶ HTTPパラメータ汚染に備えよう
  - ▶ 必要な情報のみを返すようにしよう
  - ▶ Objectのプロパティ記述子を活用しよう
  - ▶ アクセスコントロールリストを使おう
- ▶ エラーと例外のハンドリング
  - ▶ uncaughtExceptionのハンドリング
  - ▶ EventEmitter使用時はエラーを確認する
  - ▶ 非同期呼び出しのエラーを捕捉する
- ▶ サーバーのセキュリティ
  - ▶ Cookieの属性を適切に
  - ▶ セキュリティ関連のHTTPヘッダーを適宜つけよう
- ▶ プラットフォームのセキュリティ
  - ▶ パッケージの更新を維持しよう
  - ▶ 危険な関数を使わないようにしよう
  - ▶ 危険な正規表現を避けよう
  - ▶ セキュリティ検査ツールを定期的にかけてよう
  - ▶ Strictモードを使おう
  - ▶ 一般的なセキュリティ原則を守ろう

[https://cheatsheetseries.owasp.org/cheatsheets/Nodejs\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html)



# OWASP Nodejs Security cheat sheet

- ▶ アプリケーションセキュリティ
  - ▶ 非同期処理はPromiseでフラットに書こう
  - ▶ リクエストのサイズを制限しよう
  - ▶ イベントループのブロックを避けよう
  - ▶ 入力の検証をきちんとしよう
  - ▶ 出力のエスケープをきちんとしよう
  - ▶ イベントループでネットワーク帯域も監視しよう
  - ▶ 総当たり攻撃に備えよう
  - ▶ CSRFトークンを使おう
  - ▶ 不要なURLルーティングを消しておこう
  - ▶ HTTPパラメータ汚染に備えよう
  - ▶ 必要な情報のみを返すようにしよう
  - ▶ Objectのプロパティ記述子を活用しよう
  - ▶ アクセスコントロールリストを使おう
- ▶ エラーと例外のハンドリング
  - ▶ uncaughtExceptionのハンドリング
  - ▶ EventEmitter使用時はエラーを確認する
  - ▶ 非同期呼び出しのエラーを捕捉する
- ▶ サーバーのセキュリティ
  - ▶ Cookieの属性を適切に
  - ▶ セキュリティ関連のHTTPヘッダーを適宜つけよう
- ▶ プラットフォームのセキュリティ
  - ▶ パッケージの更新を維持しよう
  - ▶ 危険な関数を使わないようにしよう
  - ▶ 危険な正規表現を避けよう
  - ▶ セキュリティ検査ツールを定期的にかけてよう
  - ▶ Strictモードを使おう
  - ▶ 一般的なセキュリティ原則を守ろう

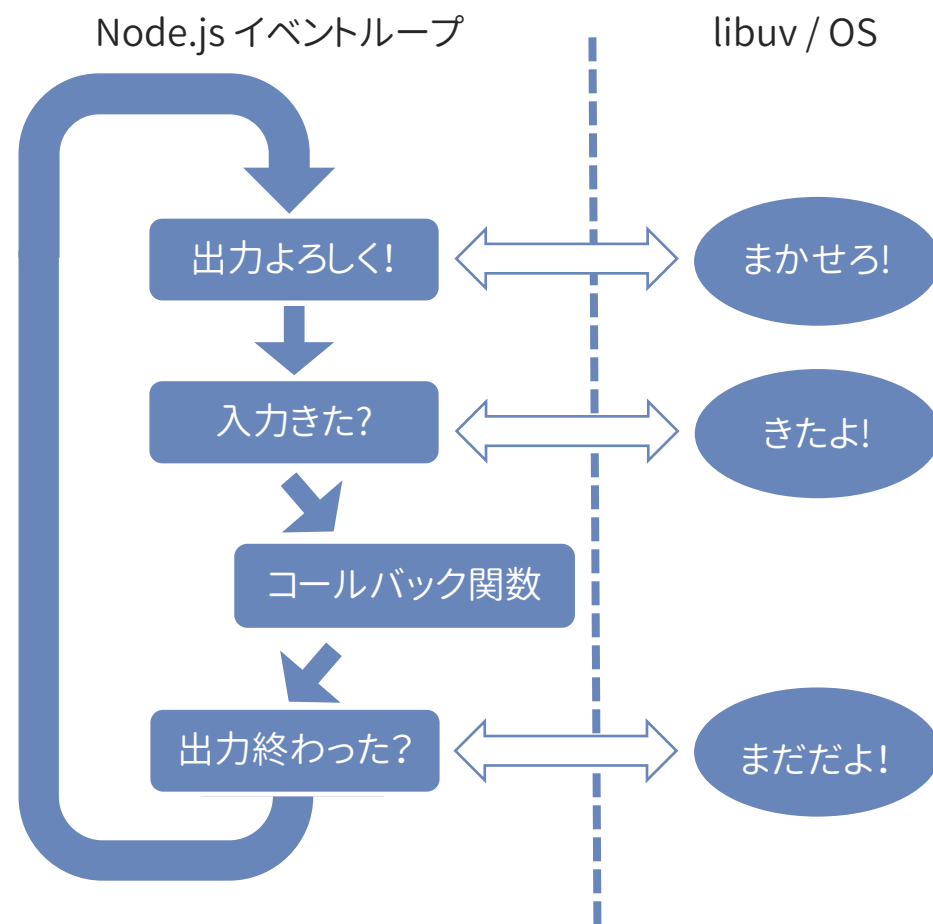
[https://cheatsheetseries.owasp.org/cheatsheets/Nodejs\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html)

# イベントループのブロックを避けよう



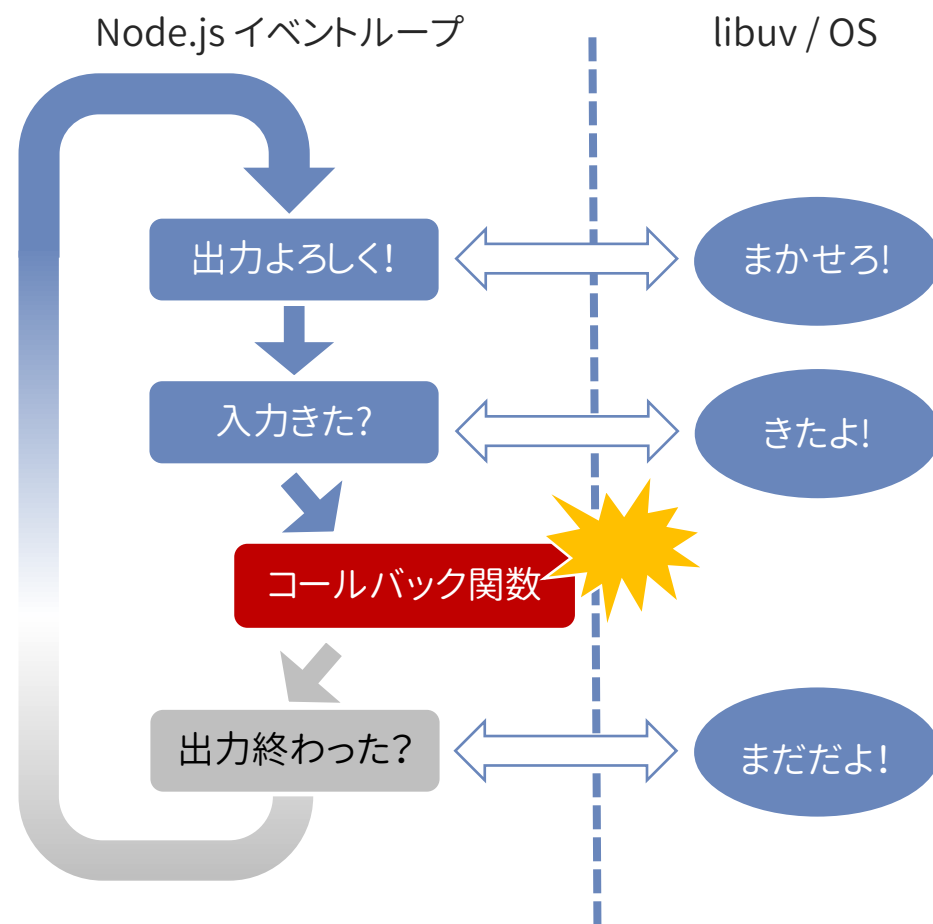
# イベントループのブロック

- ▶ Node.jsは非同期型のイベント駆動
  - ▶ イベントループにてイベントの発生を継続的に監視
  - ▶ I/Oなどは非同期に実行される
- ▶ イベントループが停止するとアプリケーション全体が止まる
  - ▶ イベントループを止めないよう各処理は速やかに処理を終えるか非同期に実行する必要がある



# イベントループのブロック

- ▶ イベントループが止まるとアプリケーション全体が止まる
  - ▶ CPUのみで行う処理などは同期的に実行され他のイベントが実行されない
  - ▶ 膨大な計算処理、時間のかかる正規表現、巨大な配列の処理、JSON.parseなど
- ▶ 攻撃者が意図的にそのような状況を作り出すことができるとDoSを発生させられる



# イベントループのブロック - ReDoS

- ▶ 正規表現によるDoS (ReDoS)
- ▶ 正規表現のパターンによっては内部的に膨大な照合処理が行われる

```
const re = /([a-z]+)+$/;  
re.test('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!'); //秒単位で処理時間がかかる
```

<https://gist.github.com/watson/7682ee718b9cf4aa2a5a605a3fb4a552> より

- ▶ 意図的にこの状態を攻撃者が作り出すことでパフォーマンスが低下する

# イベントループのブロック - ReDoS

## 対策

- ▶ 量指定子の回数を制限する  
「+」や「\*」ではなく「{0,20}」など
- ▶ safe-regexモジュールなどを用いて事前にパターンの検査を行う
- ▶ 正規表現を使わなくていい箇所では使わない  
String.prototype.includes や String.prototype.startsWith などを使用

# イベントループのブロック - JSON.parse

- ▶ JSON.parseは同期的に処理され、パース中は他のイベント処理は実行されない
- ▶ 巨大な文字列のJSON.parseはアプリケーション全体をブロックさせる可能性がある

```
const text = req.body.param
const obj = JSON.parse(text) // パースが完了するまで処理がブロック
```

# イベントループのブロック - JSON.parse

## 対策

- ▶ JSON.parseの前にテキストのサイズを確認

```
1 const text = req.body.param
2 // サイズが4k以下の時だけJSON.parseする
3 if (typeof text === 'string' && text.length < 4096) {
4   const obj = JSON.parse(text)
5 }
```

- ▶ asyncなJSONパーサーの導入
  - ▶ 使ったことがないので何とも言えず…
  - ▶ 他のライブラリで内部的にJSON.parseが呼び出されている場合には対応できない



# イベントループのブロック - JSON.stringify

- ▶ JSON.stringifyでも同じことはあり得る
  - ▶ 巨大な配列を含むJSONをシリアライズするときにブロック等  
参考:<https://techblog.yahoo.co.jp/advent-calendar-2018/goodbye-mym/#uopoo>
- ▶ 配列を含むJSONをstringifyするときは事前に配列の長さを確認

```
1 function foo (obj) {  
2   let text  
3   // 配列の長さが1k以下の時だけJSON.stringifyする  
4   if (obj.items instanceof Array && obj.items.length < 1024) {  
5     text = JSON.stringify(obj)  
6   }  
7   ...  
8 }
```

# OWASP Nodejs Security cheat sheet

- ▶ アプリケーションセキュリティ
  - ▶ 非同期処理はPromiseでフラットに書こう
  - ▶ リクエストのサイズを制限しよう
  - ▶ イベントループのブロックを避けよう
  - ▶ 入力の検証をきちんとしよう
  - ▶ 出力のエスケープをきちんとしよう
  - ▶ イベントループでネットワーク帯域も監視しよう
  - ▶ 総当たり攻撃に備えよう
  - ▶ CSRFトークンを使おう
  - ▶ 不要なURLルーティングを消しておこう
  - ▶ HTTPパラメータ汚染に備えよう
  - ▶ 必要な情報のみを返すようにしよう
  - ▶ Objectのプロパティ記述子を活用しよう
  - ▶ アクセスコントロールリストを使おう
- ▶ エラーと例外のハンドリング
  - ▶ uncaughtExceptionのハンドリング
  - ▶ EventEmitter使用時はエラーを確認する
  - ▶ 非同期呼び出しのエラーを捕捉する
- ▶ サーバーのセキュリティ
  - ▶ Cookieの属性を適切に
  - ▶ セキュリティ関連のHTTPヘッダーを適宜つけよう
- ▶ プラットフォームのセキュリティ
  - ▶ パッケージの更新を維持しよう
  - ▶ 危険な関数を使わないようにしよう
  - ▶ 危険な正規表現を避けよう
  - ▶ セキュリティ検査ツールを定期的にかけてよう
  - ▶ Strictモードを使おう
  - ▶ 一般的なセキュリティ原則を守ろう

[https://cheatsheetseries.owasp.org/cheatsheets/Nodejs\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html)

# Objectのプロパティ記述子を活用しよう



# Objectのプロパティ記述子

- ▶ 生成したオブジェクトのプロパティの書き込みや列挙を禁止できる
  - ▶ Object.defineProperty / Object.defineProperties
  - ▶ 意図しないオブジェクトの書き換えを阻止
  - ▶ strictモードと組み合わせることで早い段階でバグを検出できる

```
1 const user = {
2   name: 'hasegawa'
3 }
4 console.log(user.name) // 'hasegawa'
5 user.name = 'yosuke'
6 console.log(user.name) // 'yosuke'
```



```
1 'use strict'
2 const user = {}
3 Object.defineProperty(user, 'name', {
4   value: 'hasegawa',
5   writable: false
6 })
7 console.log(user.name) // 'hasegawa'
8 user.name = 'yosuke' // 例外発生
9 console.log(user.name)
```

# Objectのプロパティ記述子

- ▶ 既存オブジェクトのプロパティ変更も禁止できる
  - ▶ Object.preventExtensions / Object.seal / Object.freeze

```
1 'use strict'
2 const user = {
3   name: 'hasegawa'
4 }
5 Object.seal(user)
6 user.name = 'yosuke' // 既存のプロパティの値は変更可能
7 user.mail = 'hasegawa@utf-8.jp' // プロパティ追加で例外発生
```

```
1 'use strict'
2 const user = {
3   name: 'hasegawa'
4 }
5 Object.freeze(user)
6 user.name = 'yosuke' // 既存のプロパティの値変更で例外
```

# OWASP Nodejs Security cheat sheet

- ▶ アプリケーションセキュリティ
  - ▶ 非同期処理はPromiseでフラットに書こう
  - ▶ リクエストのサイズを制限しよう
  - ▶ イベントループのブロックを避けよう
  - ▶ 入力の検証をきちんとしよう
  - ▶ 出力のエスケープをきちんとしよう
  - ▶ イベントループでネットワーク帯域も監視しよう
  - ▶ 総当たり攻撃に備えよう
  - ▶ CSRFトークンを使おう
  - ▶ 不要なURLルーティングを消しておこう
  - ▶ HTTPパラメータ汚染に備えよう
  - ▶ 必要な情報のみを返すようにしよう
  - ▶ Objectのプロパティ記述子を活用しよう
  - ▶ アクセスコントロールリストを使おう
- ▶ エラーと例外のハンドリング
  - ▶ uncaughtExceptionのハンドリング
  - ▶ EventEmitter使用時はエラーを確認する
  - ▶ 非同期呼び出しのエラーを捕捉する
- ▶ サーバーのセキュリティ
  - ▶ Cookieの属性を適切に
  - ▶ セキュリティ関連のHTTPヘッダーを適宜つけよう
- ▶ プラットフォームのセキュリティ
  - ▶ パッケージの更新を維持しよう
  - ▶ 危険な関数を使わないようにしよう
  - ▶ 危険な正規表現を避けよう
  - ▶ セキュリティ検査ツールを定期的にかけてよう
  - ▶ Strictモードを使おう
  - ▶ 一般的なセキュリティ原則を守ろう

[https://cheatsheetseries.owasp.org/cheatsheets/Nodejs\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html)

# まとめ

- ▶ OWASPにはNode.jsアプリのチートシートがある
  - ▶ セキュリティだけでなく堅牢なコードにするための項目も多い
- ▶ イベントループのブロックがNode.jsの問題としては大きい
  - ▶ ReDoSはOSSの脆弱性としてもよく見つかっている
- ▶ チートシートに載っていない注意点もけっこうある
  - ▶ NoSQLインジェクションとか (Node.jsと組み合わせてよく使われる)
  - ▶ const / let、Strictモードでコードを堅牢にする
  - ▶ eslintでありがちなミスを減らす
  - ▶ TypeScriptで型を厳密にする等

One more thing ...





# prototype汚染



# prototype汚染

- ▶ `__proto__` 経由で `Object.prototype` が汚染されてしまう問題
  - ▶ `prototype.js` 時代の `prototype` 汚染とは異なる
  - ▶ あれは意図的に `Object.prototype` に便利メソッドを追加しているが迷惑だという問題
- ▶ 攻撃者が `__proto__` プロパティを宣言することでグローバルの `Object.prototype` を汚染する
  - ▶ 存在しないはずのプロパティが全オブジェクトに存在してしまう



# prototype汚染

① 外から来た文字列をJSON.parse

② そのオブジェクトを複製

③ 空のオブジェクトを生成

④ ここで表示されるのは?

```
1 function main (source) {
2   const x = JSON.parse(source) // 外からきた文字列
3   const y = clone(x)
4   const z = {}
5   console.log(z.polluted)
6 }
7
8 function clone(obj) {
9   return merge({}, obj)
10 }
11
12 function merge (a, b) {
13   for (const key in b) {
14     if (isObject(a[key]) && isObject(b[key])) {
15       merge(a[key], b[key])
16     } else {
17       a[key] = b[key]
18     }
19   }
20   return a
21 }
```

オブジェクトを複製する関数  
プロパティを列挙して再帰的にコピーするだけ

# prototype汚染

① 外から来た文字列をJSON.parse

```
1 function main (source) {
2   const x = JSON.parse(source) // 外からきた文字列
3   const y = clone(x)
4   const z = {}
5   console.log(z.polluted)
6 }
7
8 function clone(obj) {
9   return merge({}, obj)
10 }
11
12 function merge (a, b) {
13   for (const key in b) {
14     if (isObject(a[key]) && isObject(b[key])) {
15       merge(a[key], b[key])
16     } else {
17       a[key] = b[key]
18     }
19   }
20   return a
21 }
```

② そのオブジェクトを複製

③ 空のオブジェクトを生成

④ ここで表示されるのは?

source: '{"\_\_proto\_\_":{"polluted":"1"}}'  
1

オブジェクトを複製する関数  
プロパティを列挙して再帰的にコピーするだけ

# prototype汚染

```
{"__proto__": {"polluted": "1"}}
```

```
1 function main (source) {  
2   const x = JSON.parse(source) // 外からきた文字列  
3   const y = clone(x)  
4   const z = {}  
5   console.log(z.polluted)  
6 }  
7  
8 function clone(obj) {  
9   return merge({}, obj)  
10 }  
11  
12 function merge (a, b) {  
13   for (const key in b) {  
14     if (isObject(a[key]) && isObject(b[key])) {  
15       merge(a[key], b[key])  
16     } else {  
17       a[key] = b[key]  
18     }  
19   }  
20   return a  
21 }
```


\_\_proto\_\_の書き込み(複製)でグローバルなObject.prototypeが汚染される

prototype経由で存在しないプロパティが存在するかにようにふるまう  
valueOf()やtoString()なども書き替えられる可能性。最悪の場合はコード実行につながる

# prototype汚染 - 対策

- ▶ 外部からのJSONには`__proto__`のように動作に影響をあたえるキーが含まれている可能性がある
  - ▶ 外部からのJSONのキーを列挙してそのまま使用しない
  - ▶ キー名が想定されているものか確認する
  - ▶ 最低限、キーを列挙する際には `__proto__` を除外する

```
1 function merge (a, b) {  
2   for (const key in b) {  
3     if (key === '__proto__') continue  
4     if (isObject(a[key]) && isObject(b[key])) {  
5       merge(a[key], b[key])  
6     } else {  
7       a[key] = b[key]  
8     }  
9   }  
10  return a  
11 }
```



# prototype汚染 - 他の対策方法

- ▶ オブジェクトリテラル「`{}`」ではなく`Object.create(null)`を使う
  - ▶ prototypeがnullのオブジェクトが生成される
  - ▶ ただしprototypeチェーンがたどれなくなるので、`hasOwnProperty`などが直接は呼び出せなくなる(そもそも`Object.prototype.hasOwnProperty.call`するほうがいい)
- ▶ `Object.freeze`を用いてObject本体および`Object.prototype`を凍結する
  - ▶ 副作用で動かなくなるモジュールが発生する可能性がある
- ▶ JSON Schemaなどを用いて入力をより厳密に検証する
  - ▶ 入力されるJSONのキー名や型などを厳密に検証する
- ▶ ObjectではなくMapを使う
  - ▶ 外部からのデータはObjectではなくMapに格納して使用する
  - ▶ MapからObjectへコピーする際には`__proto__`をコピーしないよう注意が必要

参考: <https://techblog.securesky-tech.com/entry/2018/10/31/>



# 質問?



[hasegawa@securesky-tech.com](mailto:hasegawa@securesky-tech.com)



[@hasegawayosuke](https://twitter.com/hasegawayosuke)



<https://utf-8.jp/>