Webフロントエンドの脆弱性つまみ食い 2024年版

自己紹介

山崎啓太郎 / Twitter: @tyage

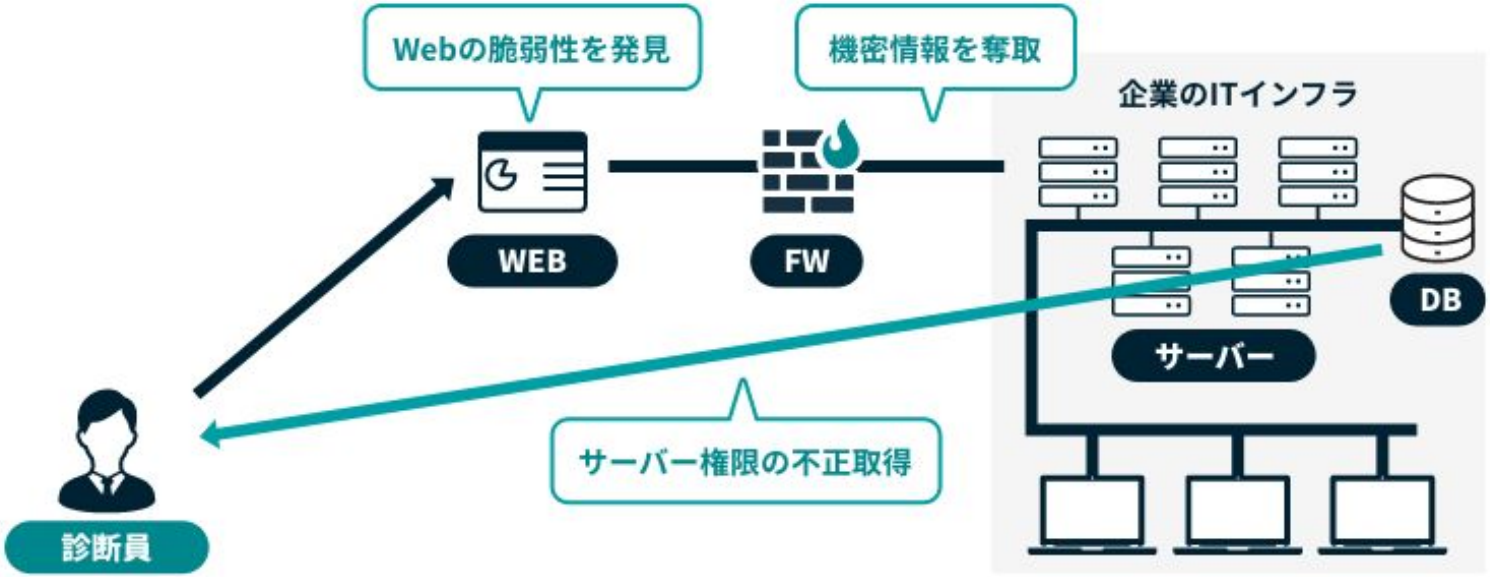
GMOサイバーセキュリティ byイエラエ アプリケーションセキュリティ課

- 元々はWebサービスの開発をしていてセキュリティの職種は6年目。JavaScriptが好き。授業中に電子辞書でjQueryのソースを読み耽っていた。
- その他
  - セキュリティ・キャンプ全国大会 2022, 2023 講師
  - Bug Bounty Rewarded: Google GitHub, Twitter, etc
  - [ブログ] サーバサイドレンダリングの導入から生じるSSRF
  - [ブログ] 危険なCookieのキャッシュとRailsの脆弱性CVE-2024-26144

[内容が公表されていれば追加予定のページ]

【広告】Webペネトレーションテスト

WebアプリケーションやWebAPIのセキュリティ対策状況を攻撃者視点で評価



https://gmo-cybersecurity.com/service/assessment/pentest/

</AD>

Webフロントエンドの脆弱性

色々な会社様のサービス・ソースコードを見てきました

機密情報が保管されるバックエンドやAPIサーバが注目されがちですが、

Webフロントエンドも重要です！

最近もまだ見るフロントエンドの脆弱性を10分で紹介

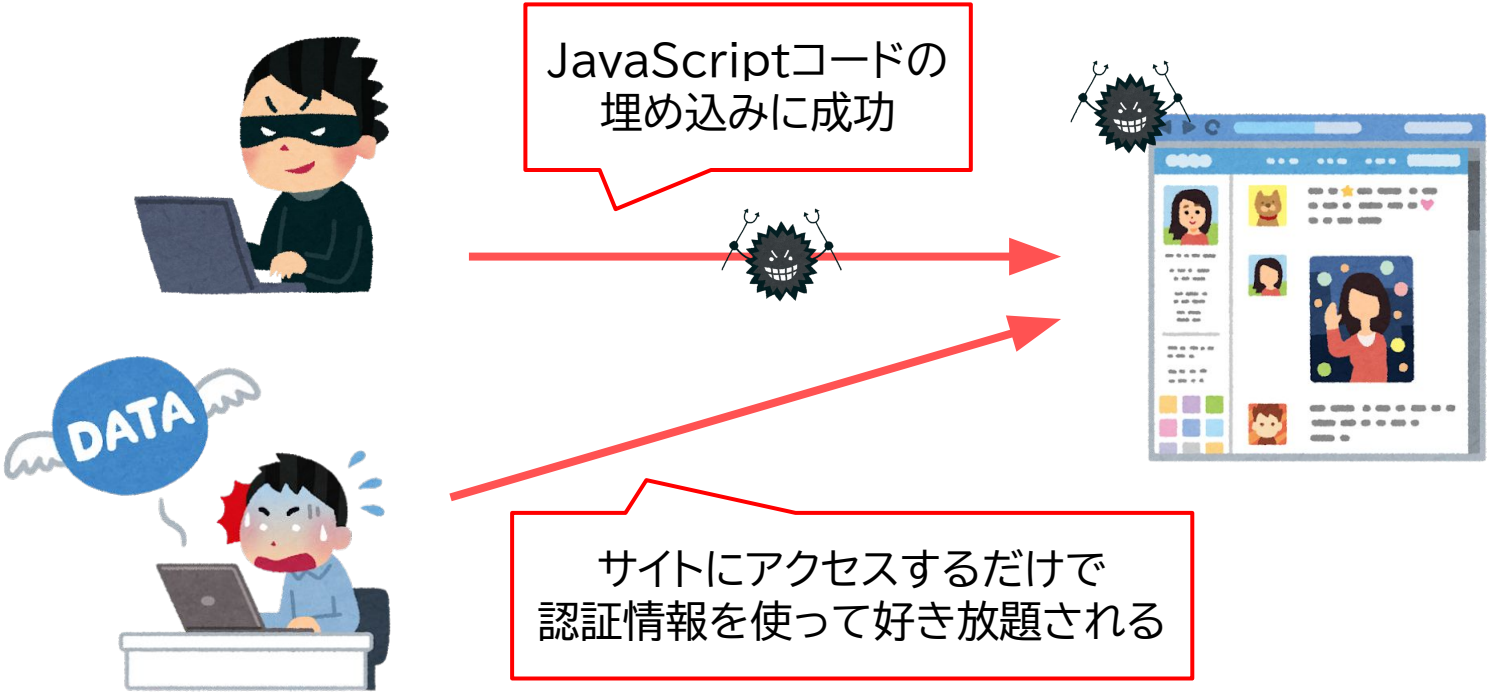
（個人的）フロントエンドの脆弱性TOP10

1. 🥇 XSS
2. 🥈 XSS
3. 🥉 XSS
4. CSRF
5. Information Leak
6. Open Redirect
7. APIのPath Traversal
8. DoS
9. XS-Leaks
10. Prototype Pollution

XSS

XSS（広義）＝
Webサイトに訪問中の被害者のブラウザ上で
JavaScriptを実行する攻撃

XSS：クロスサイトスクリプティングの一例



Webフロントエンドでの要注意ポイント

- DOMElement#innerHTML
- React dangerouslySetInnerHTML
- window.location = …
- <a href={...}>
- <iframe src={...}>
- <FOO {...props}>
- eval(...), setTimeout(...)
- などなど

Webフロントエンドでの要注意ポイント

- DOMElement#innerHTML
- dangerouslySetInnerHTML
- window.location = …
- <a href={…}>
- <iframe src={…}>
- <FOO {…props}>
- eval(…), setTimeout(…)
- などなど

```
// case1. Markdownによる入力をサポート
<div dangerouslySetInnerHTML={{
  _html: md2html(markdown)
}} />
```

Webフロントエンドでの要注意ポイント

- DOMElement#innerHTML
- dangerouslySetInnerHTML
- window.location = …
- <a href={…}>
- <iframe src={…}>
- <FOO {…props}>
- eval(…), setTimeout(…)
- などなど

```
// case1. Markdownによる入力をサポート
<div dangerouslySetInnerHTML={{
    _html: md2html(markdown)
}} />
```

_html: にJavaScriptを実行するHTMLが入り込むとアウト

_html: '<img src=0 onerror="fetch(…)">'

Webフロントエンドでの要注意ポイント

- DOMElement#innerHTML
- dangerouslySetInnerHTML
- window.location = …
- <a href={…}>
- <iframe src={…}>
- <FOO {…props}>
- eval(…), setTimeout(…)
- などなど

```
// case2. 多言語対応テキストでHTMLを使用
// e.g. <a href={}>利用規約</a>に同意する
<div dangerouslySetInnerHTML={{
    _html: i18n(error.type, error.args)
}} />
```

Webフロントエンドでの要注意ポイント

- DOMElement#innerHTML
- dangerouslySetInnerHTML
- window.location = …
- <a href={…}>
- <iframe src={…}>
- <FOO {…props}>
- eval(…), setTimeout(…)
- などなど

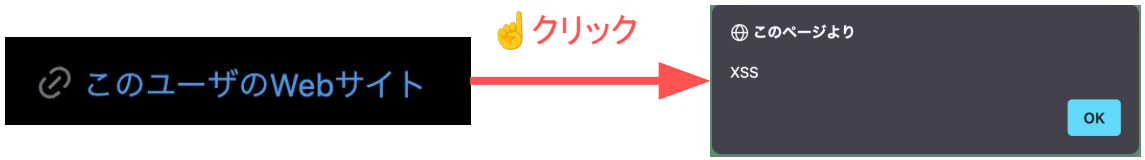
```
// case3. JSON-LDを埋め込みたいので「”」がHTMLエスケープされないように
<script type=”application/ld+json”
   dangerouslySetInnerHTML={{
      _html: JSON.stringify(jsonLd)
   }}
/>
```

ref: https://nextjs.org/docs/app/building-your-application/optimizing/metadata#json-ld

Webフロントエンドでの要注意ポイント

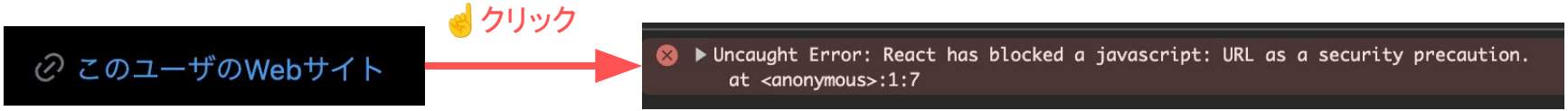
- DOMElement#innerHTML
- dangerouslySetInnerHTML
- window.location = …
- <a href={...}>
- <iframe src={...}>
- <FOO {...props}>
- eval(...), setTimeout(...)
- などなど

```
<a href="javascript:alert('XSS')">
  このユーザのWebサイト
</a>
```



【朗報】React 19から安全に

<a href={...}>等でjavascript: URLが使えなくなった！



ref: Generate safe javascript url instead of throwing with disableJavaScriptURLs is on #26507 https://github.com/facebook/react/pull/26507

(再掲)Webフロントエンドでの要注意ポイント

- DOMElement#innerHTML
- React dangerouslySetInnerHTML
- window.location = …
- <a href={...}>
- <iframe src={...}>
- <FOO {...props}>
- eval(...), setTimeout(...)
- などなど

こういった箇所はDOM Based XSSのsinkと呼ばれる

WebフロントエンドでのXSS対策

危険な関数、プロパティは使わない

WebフロントエンドでのXSS対策

- **危険な関数、プロパティはどれ → Lint、SASTツールの警告も参考に**
  - マイナーライブラリまではカバーできないが、ある程度はカバーできるはず

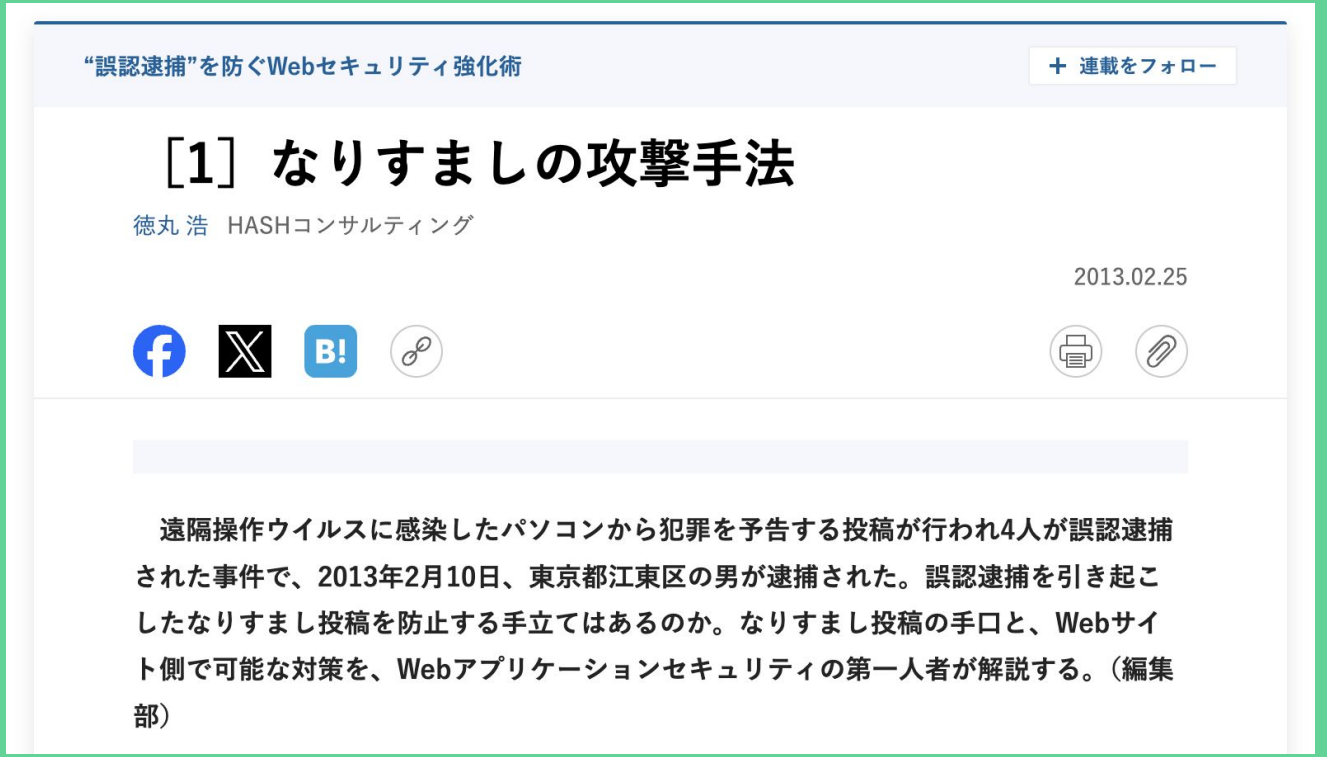
- **dangerouslySetInnerHTMLが必要 → サニタイズはDOMPurifyがオススメ**
  - DOMPurify https://github.com/cure53/DOMPurify
  - XSSに詳しい人々がこぞって改善しまくっているサニタイジングライブラリ

- **Content-Security-Policyで十分？ → ないよりマシだが回避可能なことが多い**
  - 例: Google Tag Manager www.googletagmanager.com を許可するとなんでも実行できる
  - 厳格なCSPは導入コストが高い

もちろんサーバ側もXSSの対策は必要

- HTML出力時のエスケープ
  - <script>, <a href>, <span onclick>等コンテキストに合わせたエスケープ
- ファイルアップロード先はサンドボックスドメインに
- 正確なContent-Typeヘッダの設定
- Content-Security-Policyの設定
- SSTIの対策
- キャッシュ汚染の対策
- パストラバーサルの対策
- などなど

CSRF



“誤認逮捕”を防ぐWebセキュリティ強化術

＋ 連載をフォロー

［1］なりすましの攻撃手法

徳丸 浩　HASHコンサルティング

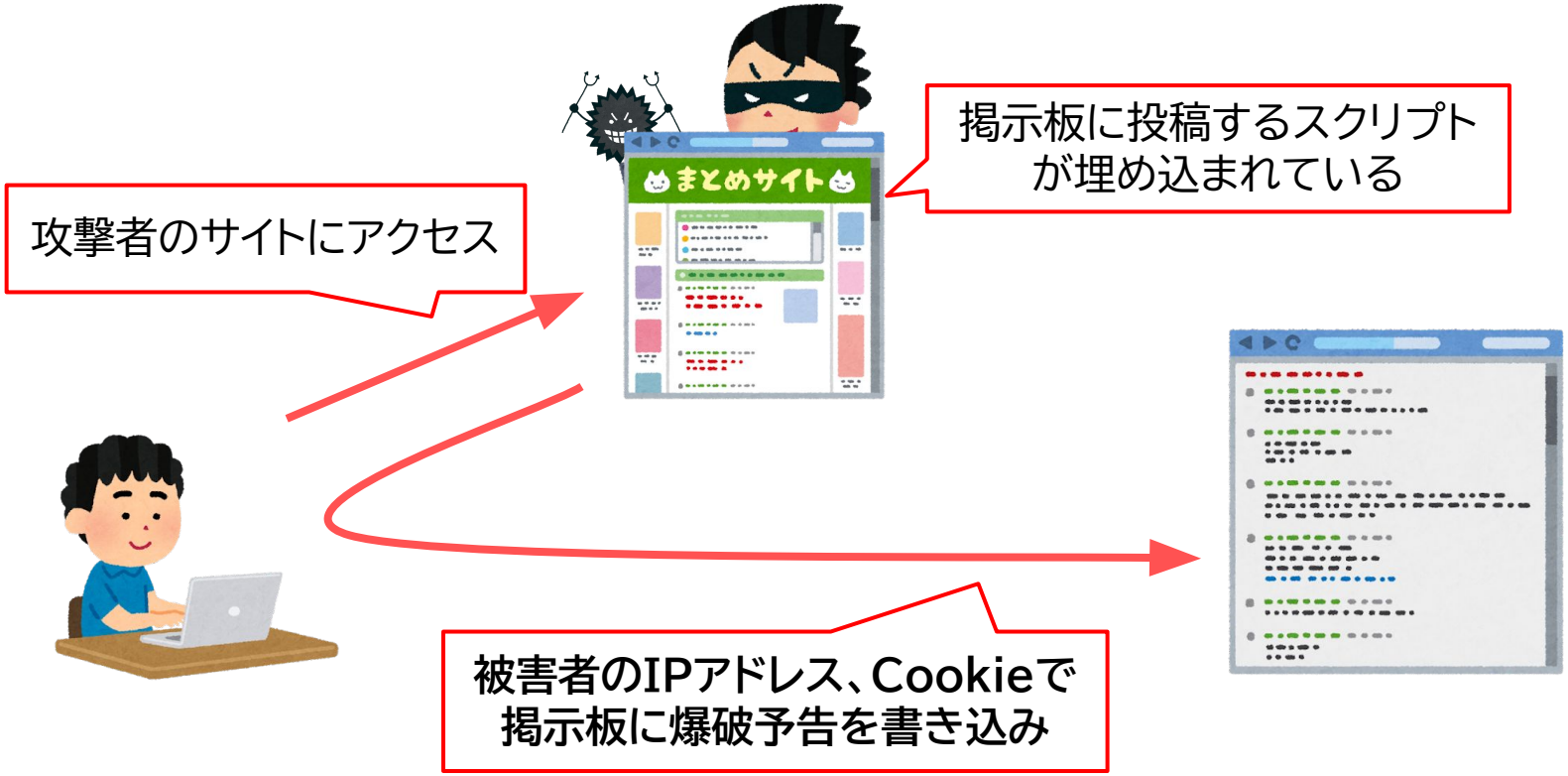
2013.02.25

遠隔操作ウイルスに感染したパソコンから犯罪を予告する投稿が行われ4人が誤認逮捕された事件で、2013年2月10日、東京都江東区の男が逮捕された。誤認逮捕を引き起こしたなりすまし投稿を防止する手立てはあるのか。なりすまし投稿の手口と、Webサイト側で可能な対策を、Webアプリケーションセキュリティの第一人者が解説する。（編集部）

https://xtech.nikkei.com/it/article/COLUMN/20130218/456763/

CSRF =
被害者のブラウザから意図しない
リクエストを送信し、サーバに処理させる攻撃

CSRF：クロスサイトリクエストフォージェリの一例



CSRFの実例

- パソコン遠隔操作事件（CSRFで掲示板に殺人予告）
- はまちちゃん事件（mixiに勝手に日記が投稿される）

上記は掲示板やブログへの意図しない書込み処理がCSRFの対象だが、

CSRFによって**意図しない認証連携やパスワード変更ができれば**

アカウントの乗っ取りに繋がる場合も

CSRFはバックエンドだけの問題？

- ユーザが意図したリクエストかどうか、**バックエンドで判断**できれば防げる
  - CSRFトークンやHTTPヘッダを検証するのがメジャーな対策
- 認証情報がlocalStorageにあればクロスオリジンリクエストでもサーバに認証情報が飛ばない
- 現代ではCookieのSameSite=laxデフォルト化によりCSRF緩和済み（※）

→ フロントエンドでCSRFは気にしなくていい？

→ Cookie使ってなければ問題ない？

※ ChromeではSameSite属性なしのCookie発行後2分間はCSRFでCookieが飛ぶ

NO

Case1. URLにアクセスすると処理が発生するもの

別サイトから遷移後、フロントエンドで自動的に処理(もしくはフロントエンドからバックエンドへリクエストを送信)するものに要注意

- ID連携　http://example.com/callback?token=eyJ….
- メールの購読停止　http://example.com/unsubscribe
- 決済　http://example.com/purchase?token=abcabc
- スマホアプリ連携　http://example.com/app?deviceId=blahblah

もしこれらのURLに**意図せずアクセスさせられてしまうと…？**

バックエンド側では意図したリクエストか攻撃か判断できないかも

Case2. リクエスト先の改ざんに要注意

```
// バックエンドからユーザ情報を取ってきて表示する
// ※ params.id がURLエスケープされていない
fetch(`/users/${params.id}`, { method: 'POST', headers: {...} })
```

example.com/users/tyage →fetch("/users/tyage")

example.com/users/tyage%2Ffollow→fetch("/users/tyage/follow")

URLにアクセスさせられると、**意図せずフォローしてしまう**

WebフロントエンドでのCSRFの対策

- そのURLにユーザが意図せずアクセスしてきても大丈夫か考えてみる
  - 処理する前に「本当に◯◯しますか？」とユーザの意図を確認する必要があるかも

- URLパスの構築時には適切にURLエンコードするかバリデーションする

まとめ

- ブラウザやライブラリの防御機構によりデフォルト安全になってはいる
  - 特にSameSite Cookieは近年でも大きな変更
- しかし対策をしなくてよくなったわけでもない
- XSSはまだまだ健在なのでお気を付けて！

- オススメ資料: Webセキュリティのあるきかた YAPC::Hakodate 2024
  https://speakerdeck.com/akiym/websekiyuriteinoarukikata

おまけ: 今回含められなかった内容

- XS-Leaks, BF-Cache等、新しい罠も
  - 新しいAPIは攻撃者にとって有用になりがち
    - Performance API, Service Worker, postMessage
- 近年のブラウザのセキュリティ対策
  - CSP, COOP, CORP等のヘッダ, Trusted Typesを適切に使えばいい感じになる。適切に設定できれば...
  - window.open等にはUser Interactionが必要になった
  - Private Network Accessの制限
  - 逆にブラウザ独自のXSS保護機構はなくなった
- ブラウザのセキュリティを回避するための”ハック”は攻撃者にとっても美味しい
  - Third party cookie対応のために”穴”を作っていませんか？
  - よく考えずに SameSite=None してませんか？
  - よく考えずに Access-Control-Allow-Origin したことはありませんか？
    - これはSameSite=Lax下では影響が小さくなった
- 旧来からずっとある脆弱性
  - DoS
    - 場合によってはサービスが止まりうる
  - Open redirect
    - あまり重要視されないし、実際そうだと思う
    - ただし、別の脆弱性と組み合わせてよくないことが起きることも