Webセキュリティのあるきかた

akiym

2024/10/5 YAPC::Hakodate 2024

自己紹介

- akiym
- 所属: Flatt Security
  - セキュリティエンジニア
  - 普段の業務は脆弱性診断など
- PAUSE ID: AKIYM
  - Smart::Args::TypeTiny, Crypt::OpenSSL::Guess
  - メンテナ: Amazon::S3::Thin, TheSchwartz

取り扱いについて

この資料ではセキュリティに関する内容を扱います。内容を正しく理解し、悪用を避けるために以下の点を厳守すること、また倫理観を持って取り扱うようにしてください。

- 資料の内容を不正に利用しないこと
- 不正アクセス、サービス妨害などの悪意ある行為を行わないこと
- 自身の管理するホストやインフラ以外に対して攻撃を行わないこと

はじめに

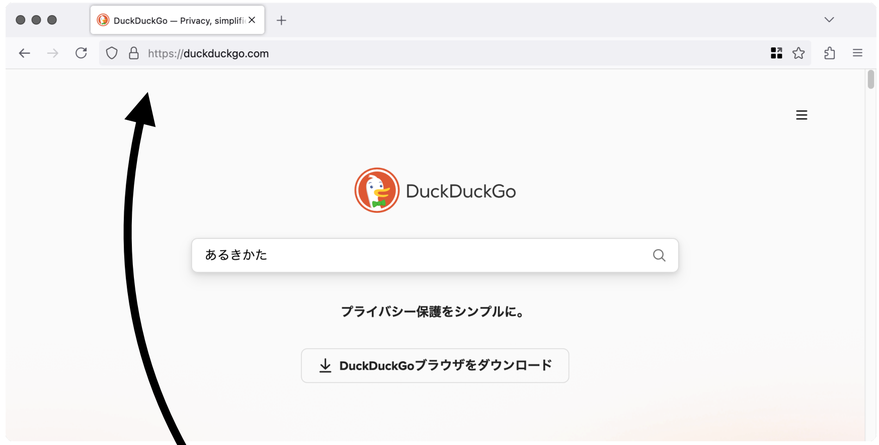
- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

Webセキュリティの「あるきかた」

- Webアプリケーション開発する上で気をつけるべきものは……？
  - 様々
  - フロントエンド
    - Cookie, CSRF, XSS
  - バックエンド
    - HTTPヘッダ、インジェクション、パストラバーサル
- できるだけ幅広く、かいつまんで紹介
  - 基礎をおさらいしつつ、ときには詳しく脇道に逸れて歩く

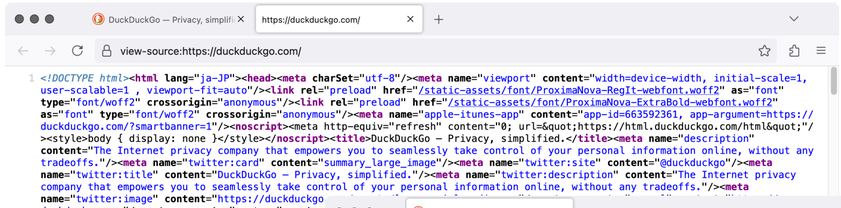
Webの構成要素

ブラウザであるURLを開いたときには……？

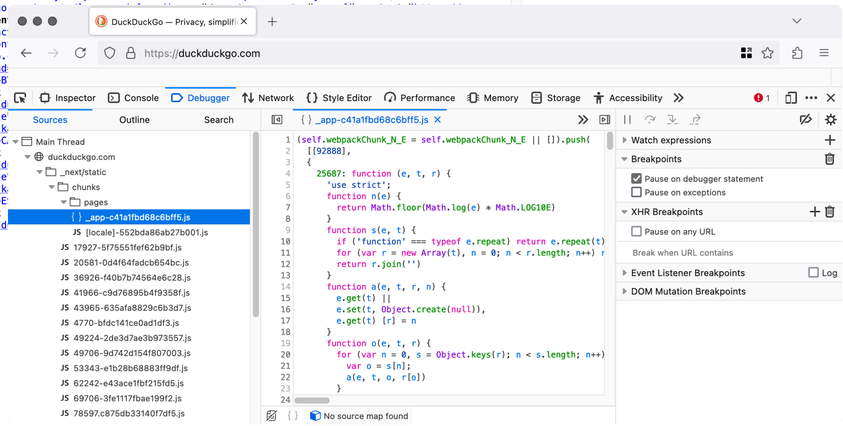


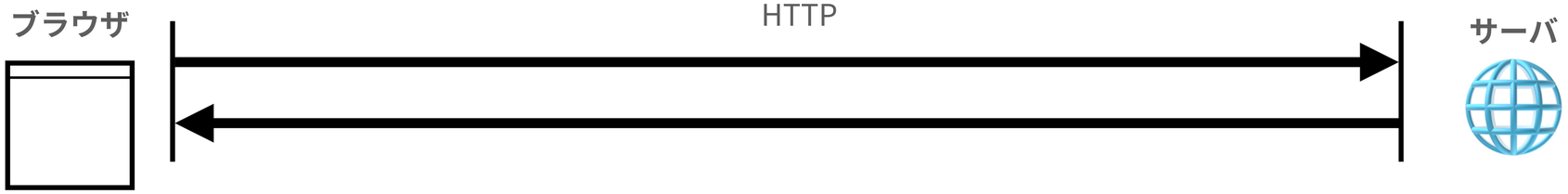
URL: https://duckduckgo.com/

HTML / CSS



JavaScript





URL

- scheme://user:pass@address:port/path?query#fragment
  - scheme: protocolとも
  - user:pass@ (userinfo): 認証情報を含めることができる
  - fragment: サーバには送られない
- userinfo部分にドメインと見間違えさせる文字列を含めてフィッシングに利用する攻撃も存在する
  - 例: https://website.test@attacker.test/

HTTP

- メソッド
- パス
- リクエスト/レスポンスヘッダ

```
GET / HTTP/1.1
Host: website.test
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1000

...
```

Cookie

- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

Cookie

- Webアプリケーションではユーザのログインなどの状態管理にCookieを用いることがある
  - HTTP単体は状態を持たないステートレス
- HTTPヘッダを経由して、Set-Cookieヘッダで設定し、Cookieヘッダで送信

Cookie

- 保存できるのはkey=valueの対
  - 有効期限を持たせることが可能
- Cookieはドメイン、パスの組み合わせによって同一のsiteと判定された場合にのみ送信される
  - portは考慮されない
  - Schemeful Same-Site
    - httpやhttpsなどのschemeを考慮した上で同一siteと判断するブラウザも存在する

Domain属性

- 指定されたドメインから派生するサブドメインより利用できるようになる
  - 指定しない場合はサブドメインでは利用できない
  - サブドメインからは親のドメインにCookieを書き込める
- 他のドメインには書き込めない(派生するサブドメインのみ)
- Domain=.website.testとしてもwebsite.testのCookieとして保存される
  - ただしDomain=website.testと指定したものとは別に保存される

```
Set-Cookie: test=...; Domain=website.test
```

Path属性

- 指定されたパスのサブディレクトリから利用できるようになる
  - Path=/
    - /, /foo, /foo/barのパスでCookieが送信される
  - Path=/foo
    - /foo, /foo/barのパスでCookieが送信される

```
Set-Cookie: test=...; Path=/test
```

Cookieの優先順位

- Domain, Pathが異なる場合には**同名のCookieを保存できる**
- RFC 6225 (5.4)ではブラウザは以下の並び順でCookieを送信する
  - Pathが長い順
  - 同一のPathの場合、Cookieの作成時間が早い順
    - (異なるDomainであったとしてもこの規則が適用される)

```
Cookie: test=1; test=2; test=3
```

test=1; Path=/foo
test=2; Path=/; Domain=.website.test（先に作成された）
test=3; Path=/; Domain=website.test（後に作成された）

Cookieの優先順位

- ブラウザの実装としても同様の規則
- Servo (Firefox)での例:

```rust
pub fn cookie_comparator(a: &ServoCookie, b: &ServoCookie) -> Ordering {
    let a_path_len : usize = a.cookie.path().as_ref().map_or( default: 0, f: |p : &&str| p.len());
    let b_path_len : usize = b.cookie.path().as_ref().map_or( default: 0, f: |p : &&str| p.len());
    match a_path_len.cmp(&b_path_len) {
        Ordering::Equal => a.creation_time.cmp(&b.creation_time),
        // Ensure that longer paths are sorted earlier than shorter paths
        Ordering::Greater => Ordering::Less,
        Ordering::Less => Ordering::Greater,
    }
}
```

https://github.com/servo/servo/blob/6f333a8e299d84c98a07a0b708fe32f40aeeeb72/components/net/cookie.rs

Cookieの優先順位

- 同名のCookieが送信された場合、どのCookieが優先されるかは
  アプリケーションの実装依存
- 最初の値を優先する:
  - Go (net/http), Java (Spring Boot), JavaScript (expressjs/cookie-parser), PHP ($_COOKIE), Perl (Cookie::Baker), Python (Flask), Ruby (Rails), Rust (actix-web)
- 最後の値を優先する:
  - JavaScript (Hono), Rust (rocket)
- 互換性を意識するような実装も→

```
# Take the first one like CGI.pm or rack do
$results{$key} = $value unless exists $results{$key};
```

https://metacpan.org/release/KAZEBURO/Cookie-Baker-0.12/source/lib/Cookie/Baker.pm

HttpOnly属性

- ブラウザ上のCookieはJavaScriptからdocument.cookieを介して参照可能
  - HttpOnly属性を指定することで、JavaScriptから参照できないようになる
- セッションを含むCookieはサーバへのリクエスト時に送信されればよいため、JavaScriptから参照される必要はない
  - XSS(後述)などにより、JavaScriptからセッションを奪取して攻撃の永続化ができなくなる

```
Set-Cookie: test=...; HttpOnly
```

Secure属性

- HTTPSの通信でのみサーバに送信されるようになる
- 中間者攻撃によってHTTPの通信を強制されたとしてもCookieが送信されない

```
Set-Cookie: test=...; Secure
```

__Secure- / __Host- prefix

- Cookie名の先頭に付けることで特別な意味を持つprefix
- __Secure-
  - Secureが必須
- __Host-
  - Secure必須、Path=/固定
  - Domain属性を設定できない

```
Set-Cookie: __Host-test=...; Path=/; Secure
Set-Cookie: __Secure-test=...; Secure
```

__Host- prefixの利点

- Cookieの特性を考えるとSecureに加えて、DomainとPathが固定できるのはかなりの利点
  - サブドメインからDomainの指定でCookieの作成がされない
    - 同名のCookieを設定されることを防ぐ
  - より長いPathを指定することで優先順位を変更し、Cookieの上書きするような挙動を避けられる

余談: __host-だとどう扱われるか？

- **__Host-**ではなく__host-, __HOST-, __HOst-といったprefixはどう扱われるか？
  - Chrome, Firefox
    - どれも__Host-と同じ扱い(大文字小文字を区別しない)
  - Safari
    - __Host-のみが特別扱い、**他は通常のCookieと同じ**
- __Secureでも同様の挙動
- この大文字小文字を区別しない仕様はRFC 6265bis (5.4)で言及されている

SameSite属性

- **異なるsite**からリクエストされた際でのCookieを制御する
- Strict: 同一siteのみで送信される(トップレベルナビゲーションによって送信されない)
- Lax: 加えてトップレベルナビゲーションや画像の読み込みなどに限定したcross-siteリクエストのときにも送信される
- None: cross-siteからのリクエストでも送信される(POSTリクエストなど)

```
Set-Cookie: test=...; SameSite=Strict
Set-Cookie: test=...; SameSite=Lax
Set-Cookie: test=...; SameSite=None; Secure
```

「同一のsiteである」とは？

- サブドメインは同一のsite
  - website.test (other-website.testのようなドメインであれば異なるsite)
  - **a.**website.test, **b.a.**website.test
- eTLD+1がドメイン名となる
  - TLD (top-level domain)
    - .com, .org, .net, .jpなど
  - eTLD (effective top-level domain)
    - ***.jp**
    - ***.co.jp**
    - ***.hokkaido.jp**
    - ***.hakodate.hokkaido.jp**

Public Suffix List

- https://github.com/publicsuffix/list で管理されているeTLDのリスト
- ユーザごとにサブドメインが割り当てされるケースでも利用される(同一siteとしない)
  - 例: Firebaseのweb.appのようなドメイン
    - Cookieの操作時に、web.appに対してもDomain指定はできない、他のサブドメインにも影響しない

```
1903    // jp : https://www.iana.org/domains/root/db/jp.html
1904    // http://jprs.co.jp/en/jpdomain.html
1905    // Submitted by registry <info@jprs.jp>
1906    jp
1907    // jp organizational type names
1908    ac.jp
1909    ad.jp
1910    co.jp
1911    ed.jp
1912    go.jp
1913    gr.jp
1914    lg.jp
1915    ne.jp
1916    or.jp
1917    // jp prefecture type names
1918    aichi.jp
```

https://github.com/publicsuffix/list/blob/master/public_suffix_list.dat

SameSiteを指定しない場合のデフォルトの挙動

- ChromeではLaxとして扱われる
  - Safari 18からはデフォルトでLaxになるように
    - https://developer.apple.com/documentation/safari-release-notes/safari-18-release-notes
      - > “ Fixed treating the lack of an explicit “SameSite” attribute as “SameSite=Lax”.
  - Firefoxは過去に上記のような挙動に変更したが、現在は戻されていてNoneとして扱われる

SameSite=NoneとSecure

- 多くのブラウザでSameSite=Noneの場合、Secure属性を必須とする
  - Firefoxでもついに131から必須となった
  - https://www.mozilla.org/en-US/firefox/131.0/releasenotes/
    - “ SameSite=None cookies will now be rejected when there is no Secure attribute included.

HttpOnly Cookieの上書き

- HttpOnlyのCookieはJavaScriptから上書きできる？
  - document.cookieからは**直接上書きできない**ようになっている
- 元のCookieより長いPathを指定することで優先順位を変更し、Cookieを上書きするような挙動は可能(ただし前述の通りアプリケーションの実装依存)
- Cookie eviction
  - Chrome、FirefoxではドメインあたりのCookie数の上限により、大量のCookieを作成することにより上書きできる
  - Safariでは(おそらく)その上限の制限がないため、以下では上書きできない

```
for (let i = 0; i < 180; i++) {
  document.cookie = `${i}=`;
}
document.cookie = 'test=overwritten';
```

Secure Cookieの上書き

- Secure CookieはHTTPでの通信から上書きできる？
  - 基本的にはできないようになっている
- Safariの挙動
  - 同一siteかつ異なるドメインであればSecure CookieでもCookieを上書きするような挙動は可能(ただし前述の通りアプリケーションの実装依存)
  - 例:
    - website.testのSecure Cookieに対して、サブドメインa.website.testからDomain=.website.testを指定して書き込む

Cookieのまとめ

- できるだけ安全にCookieを使うためには
  - JavaScriptから参照しない場合はHttpOnlyを指定する
  - __Host- prefixを使う
  - SameSite=StrictまたはLaxを指定する
- ブラウザによって現在でも細かな挙動が異なる
  - SameSiteを指定していないときのデフォルトの扱い

Cross-site request forgery (CSRF)

- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

Same-Origin Policy (SOP)

- same-origin(同一オリジン)以外からのリソースのアクセスを制限する
  - same-origin
    - オリジン(URLのscheme, host, port)が同一であること
  - same-site
    - Cookieの扱いはsame-site
  - cross-site
    - それ以外

Cross-Origin Resource Sharing (CORS)

- cross-origin(異なるオリジン)に対してのリソースのアクセスを個別に許可するための仕組み
- cross-originのリクエストが**単純リクエスト**ではない場合
  - ブラウザが送信するpreflight requestによって許可するオリジン、メソッド、ヘッダ、認証情報の有無などを指定する

単純リクエスト

- 単純リクエストであることの条件
  - GETまたはPOSTメソッド
  - Content-Typeが以下のいずれか
    - application/x-www-form-urlencoded
    - multipart/form-data
    - text/plain
- 単純リクエストではないリクエストにするには
  - カスタムヘッダを付与する
  - 上記以外のContent-Type(application/jsonなど)を付与する

Cross-site request forgery (CSRF)

- ユーザから意図しないリクエストを送信させて、認証済みのセッションを用いてアプリケーション上での操作をさせる攻撃
- 対策方法については時間の都合上省略
  - 例:
    - 令和時代の API 実装のベースプラクティスと CSRF 対策 https://blog.jxck.io/entries/2024-04-26/csrf.html

CSRFと単純リクエスト

- 単純リクエストではないことがCSRF対策になっていた場合の注意点
- JSONを受け取るが、Content-Typeがapplication/jsonであることを確認していない場合に注意
  - application/x-www-form-urlencodedの形式でJSONを送信できる
  - {"=":"","foo":"bar"}
  - {"foo":"bar"}=
    - JSONのパーサによっては受け付ける
- そもそもフレームワーク側でJSONもapplication/x-www-form-urlencodedもどちらも受け取れてしまう場合もある

Resource Isolation Policy

- Fetch Metadataを利用した対策
  - https://web.dev/articles/fetch-metadata?hl=ja
  - https://xsleaks.dev/docs/defenses/isolation-policies/resource-isolation/
- https://bughunters.google.com/blog/5896512897417216/a-recipe-for-scaling-security
  - Googleでは2023年には347サービスで導入しているとのこと
- Fetch Metadata
  - Sec-Fetch-Site, Sec-Fetch-Mode, Sec-Fetch-Destリクエストヘッダによってブラウザから送信されるメタデータ

Resource Isolation Policy

- 以下のように実装する
  1. Sec-Fetch-Siteが存在する
  2. 同一オリジン、同一siteからのリクエストのみを許可
     - Sec-Fetch-Siteの値がsame-origin, same-site, noneのいずれかである
     - (サブドメインからリクエストを無効にするならsame-siteは指定しない)
  3. リンクの遷移などのナビゲーション、iframeの埋め込みを許可
     - Sec-Fetch-ModeがnavigateでありHTTPメソッドがGET
     - Sec-Fetch-Destの値にobject, embedが含まれていないこと

CSRFとSameSite Cookie

- SameSite CookieはCSRF対策としては「ある程度」強力
  - デフォルトでSameSite=Laxと扱われた場合、対策忘れのケースであってもSameSite=Laxとして扱われることでmitigation(緩和)となる
  - SameSite Cookieのおかげで攻撃ができなかったというパターンはいくつもあるはず
- CSRFのmitigationとして導入された反面……
  - あくまで「same-site」であり、「same-origin」ではない
    - サブドメインからCSRFできてしまうパターンに注意
- **Originヘッダのチェックなどの**CSRF対策は依然として重要

Cross-site scripting (XSS)

- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

Cross-site scripting (XSS)

- 意図せずHTMLやJavaScriptを埋め込まれることで、悪意あるJavaScriptを実行させる
- XSSといってもいくつかの種類に分類できる
  - reflected (反射型) XSS
    - パラメータなどのユーザ入力がレスポンスにそのまま埋め込まれることにより発生
    - 特定のURLへの誘導などにより攻撃ができる
  - stored XSS
    - データベースなどに保存されたユーザ入力が特定のページ内で埋め込まれることにより発生
  - self-XSS
  - DOM-based XSS

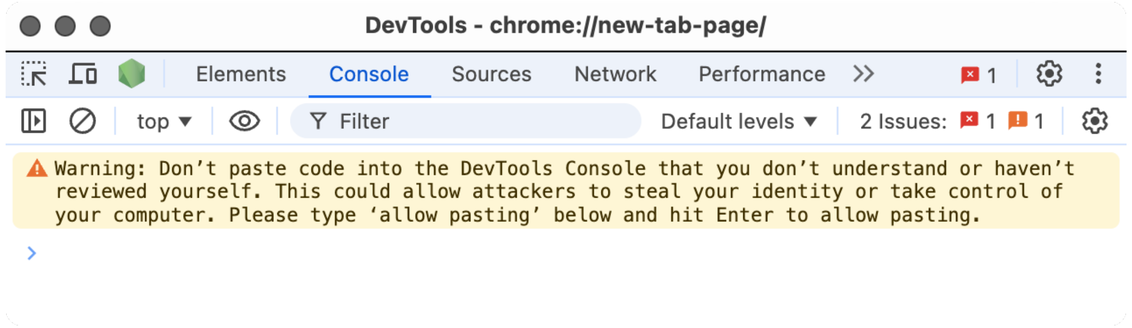
self-XSS

- 攻撃の起点として、**被害者の操作**により発生するXSS
  - URLの誘導などによって攻撃できないケース
  - 被害者自身がXSSペイロードを入力する必要がある
    - 攻撃者は入力させるように誘導させる

self-XSS

- DevToolsにコードを貼りつけてJavaScriptの実行を誘導するのも攻撃の一つ
  - 広くユーザが使うようなサービスではConsoleに警告を表示するものもある
- ブラウザによっては、DevToolsの初回起動時にクリップボードからの貼り付けが簡単にできないようにしているものもある (Chromeの例)

https://accounts.google.com/



DOM-based XSS

- JavaScript上でのDOM操作によって引き起こされるXSS
  - ユーザ入力(source)が影響箇所(sink)に入り込むことでXSSが発生する
- 例:
  - JavaScript自体の実行
    - ユーザ入力がそのままevalに含まれる
  - 不正なDOM操作
    - ユーザ入力がそのままinnerHTMLに含まれる

DOM-based XSS

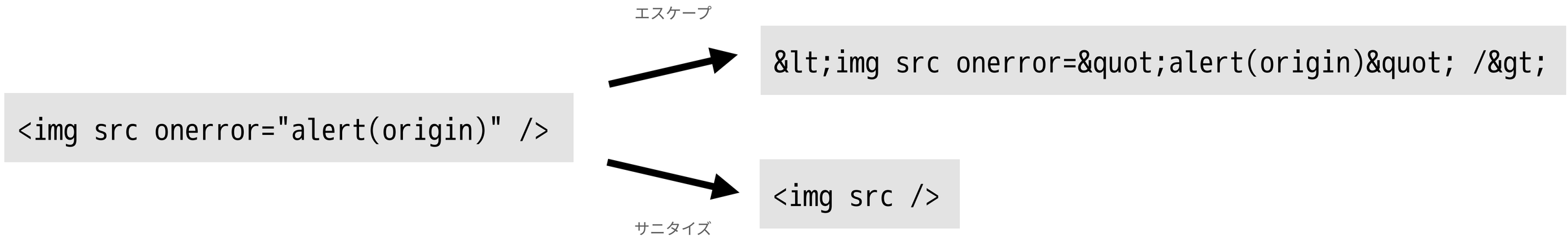
- sourceとsink
  - source: JavaScriptにおいて、ユーザ入力として何らかの値を受け付ける箇所
    - window.location (location.search, location.hash)
  - sink: 信頼できない値が入り込むことでJavaScriptの実行ができてしまう箇所
    - イベントハンドラ(onclick, onerror属性など)
    - <a>タグのhref属性 (javascript: URL)、<iframe>タグのsrc, srcdoc属性
    - location.href, window.open
    - innerHTML, outerHTMLなどのDOM操作ができるプロパティ
- safe sink: 信頼できない値が入り込んでも問題はない箇所
    - textContentプロパティなど

攻撃の目的

- ただJavaScriptが実行できるだけではなく、アプリケーションに対して操作を行うことが目的
- 攻撃可能であることのPoCとして、単にalert(1)やalert(document.cookie)だと目的としては不適切
  - httpOnly CookieであってもAPIへCookieを付与したリクエストは可能
  - アプリケーションのオリジンからリクエストできることでSOPの制限を回避する
- したがって以降の攻撃例ではalert(origin)と表記

エスケープとサニタイズ

- エスケープ
  - HTMLであれば、安全に表示できるようにエンコードするもの
  - コンテキストによってエスケープ方法が異なる
  - 各言語で提供されているライブラリを利用する
- サニタイズ(無害化)
  - HTMLであれば、表示してもXSSができないようにするもの



エスケープの例: HTML

- コンテキストに合わせてエンコードする必要がある
- HTMLであればHTML entity encode
  - & → &
  - < → <
  - > → >
  - " → "
  - ' → '

文字列を埋め込めるとする

```
<div>${username}</div>
```

```
<div><img src onerror="alert(origin)" />
</div>
```

正しいエスケープ

```
<div>&lt;img src
onerror=&quot;alert(origin)&quot; /&gt;
</div>
```

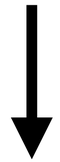
エスケープの例: JavaScript

- JavaScriptの場合
  - <script>内にJSONを埋め込んでしまうようなケースに注意
  - </script>によって開始タグを終了させられる
- データの受け渡しを<script>経由で行わない
- データの一部を埋め込む場合でも文字列自体を\xHHの形式でエンコードする

このJSON全体や一部文字列を埋め込む

```
<script>
 const data = {"username": "foo"}
 // ...
</script>
```

```
<script>
 const data = {"username": "</script><img
src onerror=alert(origin)>"}
 // ...;
</script>
```



ブラウザ上での解釈

```
<script>const data = {"username": "</script>
<img src="" onerror="alert(origin)"> event
"} // ...
```

サニタイズ

- ユーザが任意のHTMLを書けるようにしたい場合には、JavaScriptが実行されないようにサニタイズが必要
  - 利用してはいけないタグ、属性を取り除く必要があり、操作は非常に複雑
- DOMPurify
  - https://github.com/cure53/DOMPurify
  - 広く利用されているサニタイズを行うライブラリ
  - サイニタイズをバイパスできる脆弱性が見つかっていることから常に最新のバージョンを使う必要がある

sandbox domain

- ユーザによってアップロードされたHTMLを配信する場合には、サービスを提供するドメインとは異なるドメインである「sandbox domain」を使う
- Cookieに書き込めないことが重要
  - サービスを提供するドメインが website.test だったときにfile.website.testといったサブドメインではいけない
  - website.testとは異なるドメインを利用する
- 身近な例: GitHub
  - github.com
  - githubusercontent.com

sandbox domain

- Cookieが書き込めてしまうと何が困るのか
  - Cookieの上書きによって強制ログインが可能
  - 攻撃者のアカウントに気付かずにログインさせて、操作をしてしまう

現代におけるXSS

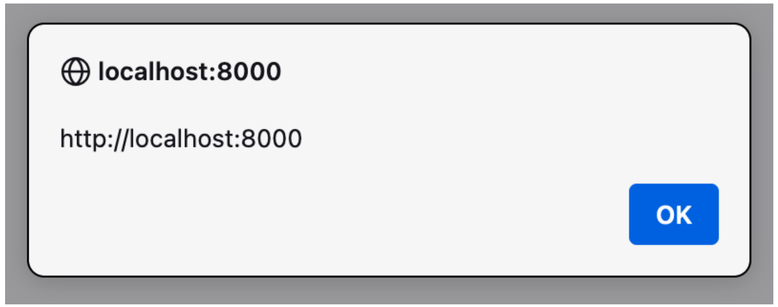
- ReactやVueなどのライブラリではデフォルトでエスケープされる
  - ただし、必要な際には**信頼された**文字列を直接HTMLとして出力できる
    - React: dangerouslySetInnerHTML
    - Vue: v-html
- 上記を使う必要があったら今一度考え直してみる
  - ユーザ入力が含まれないか？
  - HTMLを渡す必要はあるか？コンポーネントとして表現できないか？

javascript: URLに注意

- javascriptを実行するprotocolを指定したURL
- ユーザ入力が以下に渡されることでXSSが発生する
  - <a>タグのhref属性
  - <iframe>タグのsrc属性
  - location.href, window.open

```
<a href="javascript:alert(origin)">
  click
</a>
```

👉 click



javascript: URLに注意

- javascript: URLの判定を「ある文字列の先頭がjavascript: に一致する」と**してはいけない**

✅ javascript:alert(origin)

✅ javascript://a.test/%0Aalert(origin)

✅ JAVASCRIPT:alert(origin)

✅ java s c r i p t:alert(origin)

✅ java
script:alert(origin)

✅ 　javascript:alert(origin)
↑[\x00-\x1f]

javascript: URLに注意

- URLを扱う際の方針
  - URLとしてパースし、**protocol部分がhttpsであることを確認すること**が望ましい
  - JavaScriptでの例: `new URL(...).protocol`
- Reactでのjavascript: URLを判定する例(パースしない場合):

```
// A javascript: URL can contain leading C0 control or \u0020 SPACE,
// and any newline or tab are filtered out as if they're not part of the URL.
// https://url.spec.whatwg.org/#url-parsing
// Tab or newline are defined as \r\n\t:
// https://infra.spec.whatwg.org/#ascii-tab-or-newline
// A C0 control is a code point in the range \u0000 NULL to \u001F
// INFORMATION SEPARATOR ONE, inclusive:
// https://infra.spec.whatwg.org/#c0-control-or-space

const isJavaScriptProtocol =
  /^[\u0000-\u001F ]*j[\r\n\t]*a[\r\n\t]*v[\r\n\t]*a[\r\n\t]*s[\r\n\t]*c[\r\n\t]*r[\r\n\t]*i[\r\n\t]*p[\r\n\t]*t[\r\n\t]*\:/i;
```

https://github.com/facebook/react/blob/fc5ef50da8e975a569622d477f1fed54cb8b193d/packages/react-dom-bindings/src/shared/sanitizeURL.js

余談: React 19でのjavascript: URLの対応

- バージョン19以前からjavascript: URLが入力されると警告を出していた

```
Warning: A future version of React will block javascript: URLs as a security precaution. Use event handlers instead if you can. If you need to generate unsafe HTML try using dangerouslySetInnerHTML instead. React was passed "javascript:alert(document.domain)".
a
div
App
```

- React 19ではついにエラーになって使えなくなるようになった

```
Uncaught Error: React has blocked a javascript: URL as a security precaution.                    throw ne
    <anonymous> javascript:throw new Error('React has blocked a javascript: URL as a security precaution.'):1
```

- 実際にレンダリングされるDOMはこのようになる

```
<a href="javascript:throw new Error('React has blocked a javascript: URL as a security precaution.')">click</a>
<iframe src="javascript:throw new Error('React has blocked a javascript: URL as a security precaution.')">…</iframe> event
```

サードパーティのライブラリに注意

- ライブラリによってはXSSが発生するsinkが存在する場合がある
  - ライブラリにユーザ入力を渡す際は十分に注意する
- v-tooltip (v2)の例:
  - tooltipとして表示する文字列として、デフォルトでHTMLを受け取る
  - 多くのライブラリはこのような挙動を持たないが、とくに古くからあるライブラリに注意

⚠️ **HTML is enabled in the tooltip by default. If you plan on using content coming from the users, please disable HTML parsing with the `html` global option to prevent XSS attacks:**

**ユーザ入力が含まれる場合はXSSを防ぐために
オプションを無効にしなければいけない**

```
import VTooltip from 'v-tooltip'
Vue.use(VTooltip, {
  defaultHtml: false,
})
```

https://www.npmjs.com/package/v-tooltip

国際化の翻訳データに注意

- アプリケーションの国際化(i18n)のために、翻訳データ中でユーザ入力の表示や改行、文字装飾をしたい場合に注意する
- react-i18next (v9以前)の例:
  - dangerouslySetInnerHTMLをそのまま利用する形になっていた
  - 現在はTrans Componentによって部分的に翻訳データをコンポーネントに渡すことができるようになっている

翻訳データ

```
Hello,<br />Mx. <span class="name">{{name}}</span>!
```

```
こんにちは、<br /><span class="name">{{name}</span>さん！
```

利用時のコード

```
<div
  dangerouslySetInnerHTML={{
    __html: t('hello', { name })
  }}
/>
```

```
<img src onerror="alert(origin)" />
```

https://react.i18next.com/legacy-v9/interpolate#alternatives

Content Security Policy (CSP)

- XSSに対するセキュリティ機構
  - リソース(スクリプト、画像など)ごとにポリシーを設定できる
  - HTMLの埋め込みができたとしてもXSSができないよう影響を軽減する
    - 例: インラインスクリプトの実行を禁止する
- 強力ではあるが、完璧な既存のアプリケーションへの導入は困難
  - サードパーティのスクリプトのためにunsafe-inlineやunsafe-evalが常態化してしまうとCSPバイパスの可能性がある
  - https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass

Strict CSP

- Strict CSP
  - https://web.dev/articles/strict-csp
- strict-dynamic:
  - 許可されたスクリプトからさらに動的にスクリプトをロードする場合でも信頼されているとみなす

```
Content-Security-Policy:
  script-src 'nonce-{RANDOM}' 'strict-dynamic';
  object-src 'none';
  base-uri 'none';
```

https://web.dev/articles/strict-csp#structure

Tabnabbing

- target="_blank"を指定した<a>タグがクリックされた際にwindow.openerを参照することでオープン元のウィンドウの一部操作が可能
- window.opener
  - 異なるオリジンの場合、多くのプロパティはアクセスできないように制限されている
  - window.opener.locationの操作は可能であるため、オープン元のウィンドウを勝手に遷移させてフィッシングに利用される恐れがある
- **現代のブラウザではtarget="_blank"を指定した際のデフォルトの挙動がrel="noopener"を指定した状態になっている**
  - https://caniuse.com/mdn-html_elements_a_implicit_noopener
  - ただしwindow.open(url, '_blank')のようなケースには注意

CSS Injection

- style要素などに任意の入力を含められる場合、任意のCSSを指定できることになる
  - フィッシング
    - 透明な画面に覆い被されるようなリンクとなるようにスタイルを当てる
  - HTMLに含まれる情報のリーク
    - CSS selectorとbackground-imageを組み合わせ、部分的に情報を送信する
    - ほかにもいくつかのテクニックが考案されている
      - https://book.hacktricks.xyz/pentesting-web/xs-search/css-injection

Subresource Integrity

- cdnjsやjsDeliverなどのCDN上で配信されたライブラリを<script>や<link>タグから読み込み、利用することがある
- polyfil[.]ioの事例 (2024/6)
  - 広く利用されていたCDNではあったが、売却されたことにより不正なスクリプトが埋め込まれるように改竄された
  - CDNを利用する上でのメリットの反面、悪意のあるCDNの提供元によって任意のJavaScriptを実行できてしまっては困る
- 意図しないスクリプトが実行されないように、Subresource Integrityを用いることで改竄されていないことを検証できる

Subresource Integrity

- CDNから配信されているライブラリが正規のものであることを確認した上で、そのファイルに対応するハッシュを指定する
  - sha256, sha384, sha512が指定できるが、基本的にはsha384以上を推奨
  - 以下のコマンドで計算できる
    - `openssl dgst -sha384 -binary <ファイル名> | openssl base64 -A`

```
<script
  src="https://website.test/test.js"
  integrity="sha384-OLBgp1GsljhM2TJ+sbHjaiH9txEUvgdDTAzHv2P24donTt6/529l+9Ua0vFImLlb"
  crossorigin="anonymous"
></script>
```

近い未来
Trusted Types

- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

DOM-based XSSを未然に防ぐには？

- DOM-based XSSを防ぐ上で、sinkにユーザ入力が含まれるようなうっかりを避けるには……？
- 気をつけるべきsinkはいくつもある
  - innerHTML
  - outerHTML
  - insertAdjacentHTML
  - <iframe> srcdoc
  - document.write
  - document.writeln
  - DOMParser.parseFromString
  - javascript: URLの指定
  - eval

Trusted Types

- そもそもsinkに信頼できない値を注入できないようにする
  - 定義したポリシーによって作成されたTrusted Typesのオブジェクトしか受け入れないようになる
- Trusted Typesが導入されていなければsinkはコード全体にわたる
  - ユーザ入力のHTMLがサニタイズされずに使われないか？
  - サードパーティのライブラリ内でinnerHTMLは使われていない？
- 導入することでコードレビュー対象を狭めることができる
  - ポリシーが正しく実装されているか

Trusted Typesの使い方

- CSPのtrusted-typesに定義したポリシーを渡す必要がある
- サニタイズされず受け取った文字列をそのまま返すなどのポリシーの不備があってはいけない

```
Content-Security-Policy: trusted-types my-policy; require-trusted-types-for 'script'
```

```
const myPolicy = trustedTypes.createPolicy("my-policy", {
  createHTML: (s) => {
    return sanitize(s); // サニタイズを行った上で信頼された値として扱う
  },
});
div.innerHTML = myPolicy.createHTML("...");
```

Trusted Typesのポリシー

- trustedTypes.createPolicyによるポリシーの定義
  - createHTML
    - 例: HTMLをサニタイズする
  - createScriptURL
    - 例: 動的に追加するscript要素で、src属性に指定するURLのオリジンを限定する
  - createScript
    - 例: 動的に追加するscript要素で、スクリプトの内容を直接埋め込む場合
      - evalを使わないといけない場合(どちらも信頼された値に対して)

DOMPurifyとの組み合わせ

- RETURN_TRUSTED_TYPEオプションを指定することでTrusted TypesのTrustedHTMLオブジェクトを返してくれる
- デフォルトではdompurifyという名前のポリシーになっている

```
Content-Security-Policy: trusted-types dompurify; require-trusted-types-for 'script'
```

```
div.innerHTML = dompurify.sanitize(input, {
  RETURN_TRUSTED_TYPE: true,
});
```

Trusted Typesの対応状況

- 2024/10時点で対応しているのはChromiumベースのブラウザのみ
- FirefoxやSafariはまだ対応していない
  - https://github.com/w3c/trusted-types などのpolyfillは存在する
- DOM-based XSSを防ぐにはかなり強力な手段となる

HTTPヘッダ

- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

X-Frame-Options

- iframeを経由した埋め込みを制限する
  - DENY: 拒否、SAMEORIGIN: 同一オリジンの場合のみ許可
- Clickjacking
  - あるサイトでログイン状態のとき、そのサイトの透明なiframeを画面上に覆い被せることによって、意図しないクリック操作を行わせる
- CSPのframe-ancestorsを指定することでも対策可能

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
```

Strict-Transport-Security

- HTTPSで接続することを強制する
- 中間者攻撃によりサブドメインからCookieの書き込みをされないようにincludeSubdomainsの指定が重要
  - __Host- prefixを付けていないCookieであれば、サブドメインから同名のCookieを設定できてしまう

```
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
```

HSTS preload

- Strict-Transport-Securityヘッダを受け取ってからHTTPSを強制するということは初回はHTTPで受け入れてしまう可能性がある
- preloadディレクティブを指定した上で、HSTS preload listに登録しておくことで最初からHTTPSを強制できる
  - https://chromium.googlesource.com/chromium/src/net/+/refs/heads/main/http/transport_security_state_static.json
    - FirefoxのnsSTSPreloadList.incも上記から生成されている
      - https://github.com/mozilla/gecko-dev/blob/72d959d715d3f8323280576008l1c09ee5195ae1/taskcluster/docker/periodic-updates/scripts/getHSTSPreloadList.js#L20
- ちなみにgTLD単位でHSTS preloadすることを決められる
  - .devなどはHTTPS限定

Content-Type

- ブラウザが表示される際にHTMLとして解釈されるもの
- sandbox domain以外から任意のコンテンツを返す際に以下のContent-Typeを指定されるとXSSが発生する
  - text/html
  - application/xhtml+xml
  - application/xml
  - text/xml
  - image/svg+xml
  - text/xsl
  - text/xsl
  - application/vnd.wap.xhtml+xml
  - multipart/x-mixed-replace
  - text/rdf
  - application/rdf+xml
  - application/mathml+xml
  - text/vtt
  - text/cache-manifest

https://github.com/BlackFan/content-type-research/blob/4e437472545a3a1708fb5647929053c37fa49177/XSS.md

Content-Type

- image/svg+xml
  - ユーザのアップロードした画像のみを返す場合でもSVGには注意

```
<svg>
  <script type="text/javascript">alert(origin);</script>
  <image xlink:href onerror="alert(origin)" />
</svg>
```

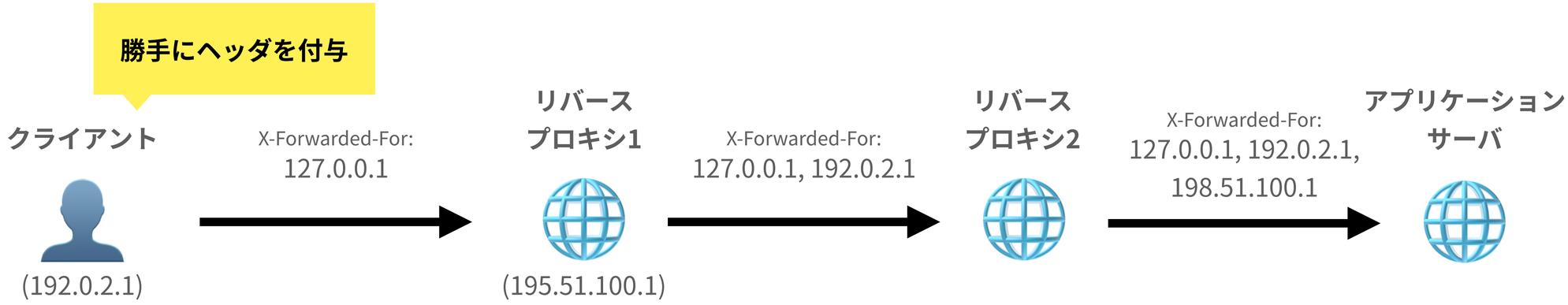
- SVGを表示したいユースケースではdata: URLが使える

```
<img src="data:image/svg+xml,&lt;svg&gt;&lt;script type=&quot;text/
javascript&quot;&gt;alert(origin);&lt;/script&gt;&lt;/svg&gt;" />
```

JavaScriptは動かない

X-Forwarded-For

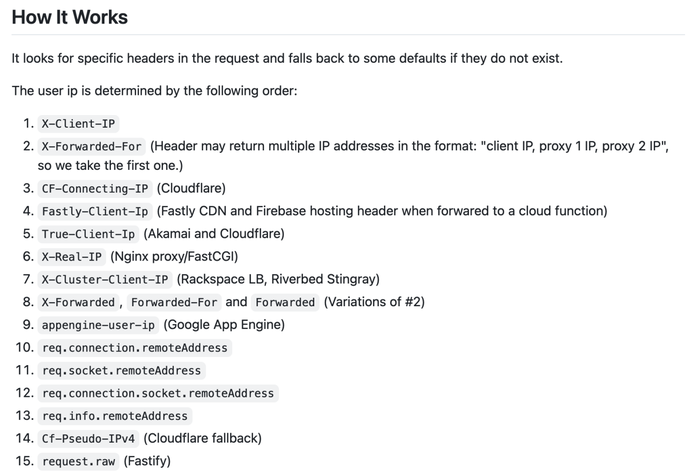
- リバースプロキシなどがクライアントとサーバの間にあるとき、接続元のIPアドレスを伝えるために「よく使われる」ヘッダ
- **クライアントのX-Forwarded-Forヘッダをそのまま受け付けてはいけない**
  - アプリケーションではプロキシのIPアドレスを排除した上で最も右に位置するアドレスを採用する
  - またはリバースプロキシで前段がクライアントと判定できるのであればヘッダを排除する




```
X-Forwarded-For: 127.0.0.1, 192.0.2.1, 198.51.100.1
```

X-Forwarded-For

- ライブラリやフレームワークによっては様々なヘッダを受け付ける場合がある
- 例: request-ip (Node)
  - https://github.com/pbojinov/request-ip
  - X-Forwarded-Forだけではない→



How It Works

It looks for specific headers in the request and falls back to some defaults if they do not exist.

The user ip is determined by the following order:

1. `X-Client-IP`
2. `X-Forwarded-For` (Header may return multiple IP addresses in the format: "client IP, proxy 1 IP, proxy 2 IP", so we take the first one.)
3. `CF-Connecting-IP` (Cloudflare)
4. `Fastly-Client-Ip` (Fastly CDN and Firebase hosting header when forwared to a cloud function)
5. `True-Client-Ip` (Akamai and Cloudflare)
6. `X-Real-IP` (Nginx proxy/FastCGI)
7. `X-Cluster-Client-IP` (Rackspace LB, Riverbed Stingray)
8. `X-Forwarded` , `Forwarded-For` and `Forwarded` (Variations of #2)
9. `appengine-user-ip` (Google App Engine)
10. `req.connection.remoteAddress`
11. `req.socket.remoteAddress`
12. `req.connection.socket.remoteAddress`
13. `req.info.remoteAddress`
14. `Cf-Pseudo-IPv4` (Cloudflare fallback)
15. `request.raw` (Fastify)

https://github.com/pbojinov/request-ip/blob/e1d0f4b89edf26c77cf62b5ef662ba1a0bd1c9fd/README.md

- 上記のような影響するヘッダが存在する場合、プロキシ/アプリケーション上で**信頼されたヘッダ以外を排除して**利用しなければいけないことに注意
  - 接続元のIPアドレスの詐称ができてしまう
  - X-Forwarded-Host, X-Forwarded-Portも同様

Forwarded

- X-Forwarded-Forが標準化されたヘッダ
  - for: X-Forwarded-Forと同等
  - host: X-Forwarded-Hostと同等
    - クライアントからのHostヘッダに相当する
  - proto: X-Forwarded-Protoと同等
    - クライアントからのプロトコルに相当する
- 同様にクライアントから信頼できない値を排除しないといけない

```
Forwarded: for=192.0.2.1;host=website.test;proto=https
```

インジェクション

- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

インジェクション

- 意図しないユーザ入力が入り込むことで発生する脆弱性
- 例:
  - SQLインジェクション
  - OSコマンドインジェクション
  - SSTI (Server Side Template Injection)
- ユーザ入力などの信頼できない値をアプリケーション上で扱う際には適切なエスケープが必要

orderに潜む罠

- ORMによってはORDER BYに相当するorderを渡す際にユーザ入力が渡されないことを前提していることに注意
  - SQL Injectionの可能性がある
- 例: gorm (Go)

```
313  func (db *DB) Order(value interface{}) (tx *DB) {
314          tx = db.getInstance()
315
316          switch v := value.(type) {
317          case clause.OrderBy:
318                  tx.Statement.AddClause(v)
319          case clause.OrderByColumn:
320                  tx.Statement.AddClause(clause.OrderBy{
321                          Columns: []clause.OrderByColumn{v},
322                  })
323          case string:
324                  if v != "" {
325                          tx.Statement.AddClause(clause.OrderBy{
326                                  Columns: []clause.OrderByColumn{{
327                                          Column: clause.Column{Name: v, Raw: true},
328                                  }},
329                          })
330                  }
331          }
332          return
333  }
```

ユーザ入力をそのまま渡してしまうとORDER BYに展開されてしまう

https://github.com/go-gorm/gorm/blob/68434b76eb567e00858a9f0eef72e79026e37b83/chainable_api.go

orderに潜む罠

- Ruby on Railsの場合
  - バージョン5以前ではorderにユーザ入力が渡されてはいけなかった
    - https://rails-sqli.org/rails5
  - 現在はSQL Injectionまでの危険性はないが、インデックスの効かないような意図しないorderを渡されないように注意

```
2039        def preprocess_order_args(order_args)
2040          model.disallow_raw_sql!(
2041            flattened_args(order_args),
2042            permit: model.adapter_class.column_name_with_order_matcher
2043          )
```

文字列が"name ASC"のような形式でなければ許可しない

https://github.com/rails/rails/blob/ba468db0bdc880c694df091b5800d114e963eff0/activerecord/lib/active_record/relation/query_methods.rb

ユーザ入力の混入

- どのようなときでも**バリデーションを忘れない**というのが大前提
- application/x-www-form-urlencodedで配列を受け取るパターンに注意
  - `a=1&a=2`や`a[]=1&a[]=2`のとき、aを`['1', '2']`と受け取れるフレームワークもある
  - Rack (Ruby)でのparse_nested_queryによる特殊なパターン
    - ハッシュとしても受け取れる
    - `a[][b]=1`のとき、aを`[{'b' => '1'}]`と受け取る

ユーザ入力の混入

- application/jsonでJSONを受け取るパターン
  - 配列やオブジェクトを渡せることになる
- 例: Prisma
  - $queryRawのようなSQLを直接書ける機能以外にもパラメータとしてユーザ入力が混入しないように注意する

Filter all `Post` records that contain `"prisma"`

```
// Run inside `async` function
const filteredPosts = await prisma.post.findMany({
  where: {
    OR: [
      { title: { contains: 'prisma' } },
      { content: { contains: 'prisma' } },
    ],
  },
})
```

例えば、ここに任意のオブジェクトが渡せてしまうとwhere条件を組み立てられる

https://www.prisma.io/docs/orm/overview/introduction/what-is-prisma

パストラバーサル

- はじめに
- Cookie
- CSRF
- XSS
- Trusted Types
- HTTPヘッダ
- インジェクション
- パストラバーサル

パストラバーサル

- 特定のディレクトリ以下にあるファイルを読み込める機能がある際に
  意図せず親ディレクトリ外のファイルを参照できてしまう脆弱性
  - ../などを用いて親ディレクトリ外を参照できる
- ユーザ入力とのファイルパスの連結時には、パスとしての連結後に正規化した上で、親ディレクトリで始まっていることを確認する

URLパスの正規化に注意

- URLにおいても、パス部分はファイルパスと同様に正規化できる仕様
- https://website.test/foo/bar と同じURLを表現するにしても……
  - `https://website.test/./foo/bar`
  - `https://website.test/../foo/bar`
  - `https://website.test/foo/baz/../bar`
  - `https://website.test//foo/bar`
  - `https://website.test/foo%2Fbar`
  - `https://website.test/foo\bar`

URLパスの正規化に注意

- 正規化するのは誰？
  - ロードバランサ: nginx, ALB, ...
  - Webアプリケーションフレームワーク
  - アプリケーションで扱う際
    - URLの結合などをユーザ入力に対して安易に利用しないように注意する
      - Python: urllib.parse.urljoin (正規化はもちろん、絶対URLとしても置き換えられてしまう)

Note: If *url* is an absolute URL (that is, it starts with `//` or `scheme://`), the *url*'s hostname and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

https://docs.python.org/3/library/urllib.parse.html

URLパスの正規化に注意

- 特定のURLパスを指定したアクセス制御などに注意
  - 正規化されている/いない前提で扱わない
  - フレームワークの挙動を知っておく
- 例題: Flatt Security Speedrun CTF - deny
  - /%2Fadmin/flagにアクセスできてしまう
  - WSGI (Python)におけるREQUEST_URIの扱い

```python
class ForbidAdminMiddleware:
    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        request_uri = environ['REQUEST_URI']
        if request_uri.startswith('/admin'):
            start_response('403 Forbidden', [])
            return [request_uri.encode()]
        return self.app(environ, start_response)


app = Flask(__name__)
app.wsgi_app = ForbidAdminMiddleware(app.wsgi_app)


@app.route("/")
def index():
    return 'Welcome to Speedrun CTF!'


@app.route("/admin/flag")
def flag():
    return os.environ.get('FLAG', '')
```

https://github.com/flatt-security/mini-ctf/blob/4617ed92e9bfe2c7600f3afb584e27dc0155d48c/2023-11-speedrun/001/server.py

URLの組み立て

- URLのパラメータを埋め込むケースにも注意
  - `https://website.test/foo?q=...&param=${param}`
    - `https://website.test/foo?q=...&param=1&bar=2&q=3`
  - `https://website.test/foo/${id}/create`
    - `https://website.test/foo/../bar?/create`
- 適切にURLのエンコードを行う
  - JavaScript: encodeURIComponent
  - クエリ部分は文字列として連結せず、URLとして正しく組み立てる

クラウドストレージにおけるパス

- AWSのS3やGoogle CloudのCloud Storageにおいてはディレクトリ構造ではなく、単なるパス
- パスに.や..を含めることができる
  - /foo/./../barはそのまま
    - 親ディレクトリ相当を参照しにいくわけではない
- だとすると問題ないと思いきや……？

クラウドストレージにおけるパス

- アプリケーションでパスの正規化をしないように注意する
  - ファイルパスとして結合してしまい、正規化されてはいけない
- 例: aws-sdk-go v1
  - デフォルトでパスを正規化してしまうが、互換性のためこの挙動は残されたまま
  - WithDisableRestProtocolURICleaningを指定する、もしくはv2への移行する必要がある
    - https://pkg.go.dev/github.com/aws/aws-sdk-go/aws#Config.WithDisableRestProtocolURICleaning

nginxでの落とし穴

- nginxのconfigを書く際には、やってはいけない落とし穴がある
- 例: alias traversal
  - 以下のような設定だと1つ上のディレクトリのファイルを参照できてしまう
  - https://github.com/yandex/gixy などのツールを使うことで検知可能

誤った例

/で終わっていない

```
location /img {
  alias /data/images/;
}
```

/img../secret.txtにアクセスすると
/data/images/../secret.txtを参照する

正しい例

```
location /img/ {
  alias /data/images/;
}
```

まとめ

まとめ

- Cookie: HttpOnly, __Host- prefix, SameSite
- XSS: javascript: URLに注意する、Trusted Typesの導入
- HTTPヘッダ: Strict-Transport-Security, X-Forwarded-For
- インジェクション: どのようなときでもバリデーションを忘れない
- パストラバーサル: パスの連結時の扱い、正規化する挙動

このほかにも紹介できなかった事例はたくさんありますが、Webセキュリティの世界は広く、「あるきかた」として皆さんの第一歩になれば幸いです。

リファレンス

- Michal Zalewski「めんどうくさいWebセキュリティ」翔泳社, 2012
- 米内貴志「Webブラウザセキュリティ ― Webアプリケーションの安全性を支える仕組みを整理する」ラムダノート, 2021

Cookie
- https://blog.flatt.tech/entry/samesite_csrf_hsts
- https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis/
- https://developer.mozilla.org/en-US/docs/Glossary/eTLD
- https://developer.mozilla.org/ja/docs/Glossary/TLD
- https://developer.mozilla.org/ja/docs/Web/API/URL
- https://developer.mozilla.org/ja/docs/Web/HTTP/Cookies
- https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Set-Cookie
- https://metacpan.org/release/KAZEBURO/Cookie-Baker-0.12/source/lib/Cookie/Baker.pm
- https://url.spec.whatwg.org/
- https://web.dev/articles/schemeful-samesite?hl=ja
- https://web.dev/articles/understanding-cookies?hl=ja
- https://www.rfc-editor.org/rfc/rfc6265.html

CSRF
- https://developer.mozilla.org/en-US/docs/Glossary/CSRF
- https://web.dev/articles/same-site-same-origin?hl=ja

XSS
- https://book.hacktricks.xyz/pentesting-web/xs-search/css-injection
- https://book.hacktricks.xyz/pentesting-web/xss-cross-site-scripting
- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- https://developer.chrome.com/blog/self-xss?hl=ja
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
- https://developer.mozilla.org/ja/docs/Web/Security/Subresource_Integrity
- https://developer.mozilla.org/ja/docs/Web/URI/Schemes/javascript
- https://github.com/facebook/react/blob/fc5ef50da8e975a569622d477f1fed54cb8b193d/packages/react-dom-bindings/src/shared/sanitizeURL.js
- https://portswigger.net/web-security/cross-site-scripting/dom-based
- https://portswigger.net/web-security/cross-site-scripting/reflected
- https://react.i18next.com/legacy-v9/interpolate#alternatives
- https://vuejs.org/guide/best-practices/security.html

リファレンス

Trusted Types

- https://docs.google.com/document/d/1m91JZWKAGOR3jQoicMVE9Ydcq79gM2BetcRIBemrex8/view#
- https://web.dev/articles/trusted-types?hl=ja

HTTPヘッダ

- https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/Forwarded
- https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/X-Forwarded-For
- https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/X-Forwarded-Host
- https://developer.mozilla.org/ja/docs/Web/HTTP/Headers/X-Forwarded-Proto
- https://github.com/BlackFan/content-type-research/blob/4e437472545a3a1708fb5647929053c37fa49177/XSS.md
- https://owasp.org/www-community/attacks/Clickjacking

インジェクション

- https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection
- https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html
- https://github.com/go-gorm/gorm/blob/68434b76eb567e00858a9f0eef72e79026e37b83/chainable_api.go
- https://github.com/rails/rails/blob/ba468db0bdc880c694df091b5800d114e963eff0/activerecord/lib/active_record/relation/query_methods.rb
- https://www.prisma.io/docs/orm/overview/introduction/what-is-prisma

パストラバーサル

- https://book.hacktricks.xyz/pentesting-web/file-inclusion
- https://docs.python.org/3/library/urllib.parse.html
- https://github.com/flatt-security/mini-ctf/blob/4617ed92e9bfe2c7600f3afb584e27dc0155d48c/2023-11-speedrun/001/server.py
- https://github.com/yandex/gixy
- https://owasp.org/www-community/attacks/Path_Traversal
- https://pkg.go.dev/github.com/aws/aws-sdk-go/aws#Config.WithDisableRestProtocolURICleaning