ちょっと話題の記事

node.JsにおけるCSRF対策

node.js

梶原裕

2014.07.31

はじめに

現在参画中の案件ではNode.js + Expressを用いた開発を行っています。

開発を行っているのはWebアプリのため、当然セキュリティ対策も必要になってきます。

今回は、CSRF(クロスサイトリクエストフォージェリ）対策として、 ミドルウェアであるcsurfを検証しました。

> Webサイトにスクリプトや自動転送(HTTPリダイレクト)を仕込むことによって、閲覧者に意図せず別のWebサイト上で何らかの操作(掲示板への書き込みなど)を行わせる攻撃手法。

CSRFとは 〔 クロスサイトリクエストフォージェリ 〕 【 XSRF 】 - 意味/解説/説明/定義 : IT用語辞典

この攻撃の特徴としては、利用者が攻撃者が用意したリンクやスクリプトにアクセスすることで、 本来フローとは異なるフローでアクセスを行うといった点でです。

対策としては登録などの行うフローにおいて、その前段階(例：入力等)において、Tokenを発行し、 登録時にTokenが正しいかを確認するといった方法があげられます。

csurfとは

Node.jsのCSRF対策のミドルウェアになります。

CSRF対策で必要とされるTokenの発行・その検証を行ってくれます。

expressjs/csurf · GitHub

導入方法

Expressの雛形に対して、CsurfとSession管理の仕組みを導入します。

```
$ npm install --save connect-redis
$ npm install
```

Session保存先としてRedisを使っています。MacのHomebrew導入済みの環境ならば 以下で導入が可能です。

```
$ brew install redis
$ redis-server
```

Linuxの場合は以下になります。ディストリビューションとしてはAmazon Linux利用しています。

```
$ sudo yum install gcc-c++ make openssl-devel
$ sudo yum install git
$ wget http://download.redis.io/redis-stable.tar.gz
$ tar xvzf redis-stable.tar.gz
$ cd redis-stable
$ make
$ sudo make install
$ redis-server
```

App.js

```
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var session = require('express-session');
var RedisStore = require('connect-redis')(session);
var csurf = require('csurf');

var routes = require('./routes/index');

```

```
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

```
app.use(csurf());

app.use('/', routes);
app.use('/users', users);
```

app.use(csurf());を呼ぶ箇所が肝になります。

csurf自体がCookieやSessionを使う仕組みになるため、 呼ぶのはCookieやSessionの設定が終わってから呼ぶ必要があります。

javascript - Error: misconfigured csrf - Express JS 4 - Stack Overflow

実装例

csurfは挙動として、Get Head Option以外のの要求に対して。Tokenのチェックを行います。 そのため、今回の作例ではフォームとAjaxでPost通信を行い、Tokenを用いた認証ができることを確認します。

/routes/index.js

```
var express = require('express');
var router = express.Router();
router.get('/', function(req, res) {
  res.render('index', { title: 'Express',reqCsrf:req.csrfToken()});
});

router.post('/regist',function(req,res){
        res.send('OK')
});

router.post('/registXhr',function(req,res){
    if(req.xhr){
        res.send('xhr Access');
```

```
module.exports = router
```

/view/index.ejs/

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
    <script src="/javascripts/xhrAccess.js"></script>
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
    <form action="/regist" method="post">
        <input type="submit" value="想情(Token沫)">
    </form>
    <form action="/regist" method="post">
        <input id="token" type="hidden" name="_csrf" value="<%= reqCsrf %>">
        <input type="submit" value="想情(Token弥)">
    </form>
    <form action="/registXhr" method="post">
        <input id="token" type="hidden" name="_csrf" value="<%= reqCsrf %>">
        <input type="submit" value="想情(Token弥 Xhr沫)">
    </form>
    <input id="xhrSubmit" type="submit" value="Ajax想情(Token弥)">
  </body>
</html>
```

/public/javascripts/xhrAccess.js

```
'use strict';
$(function() {
```

```
            if (token) {
                return jqXHR.setRequestHeader('X-XSRF-Token', token);
            }
        }
    });
    function cheer() {
        var cheerPost = $.post('/registXhr', '');
        cheerPost.done(function(result) {
            alert(result);
        });
    }
    $('#xhrSubmit').click(function() {
        cheer();
    });
});
```

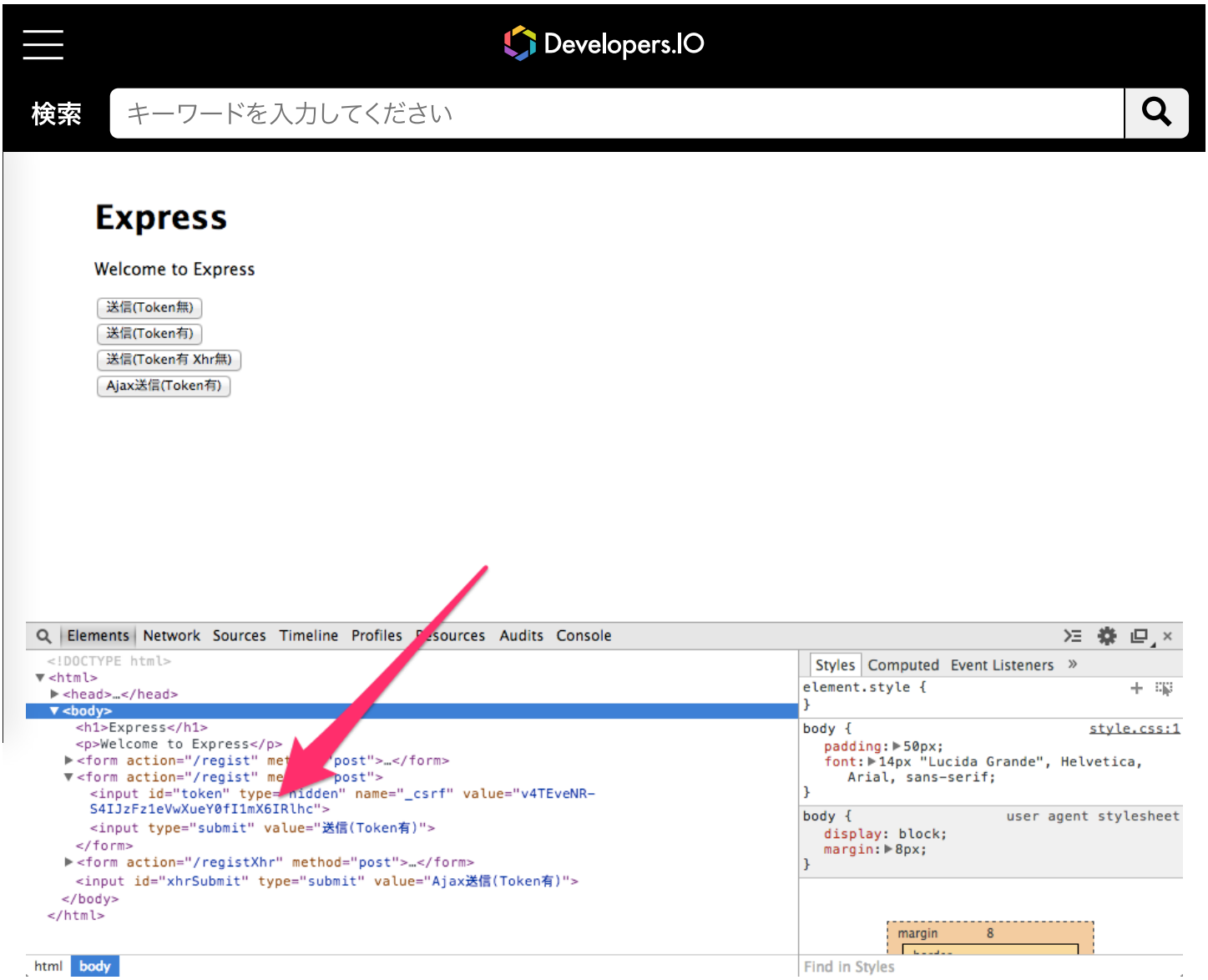
動作解説・検証(Form)

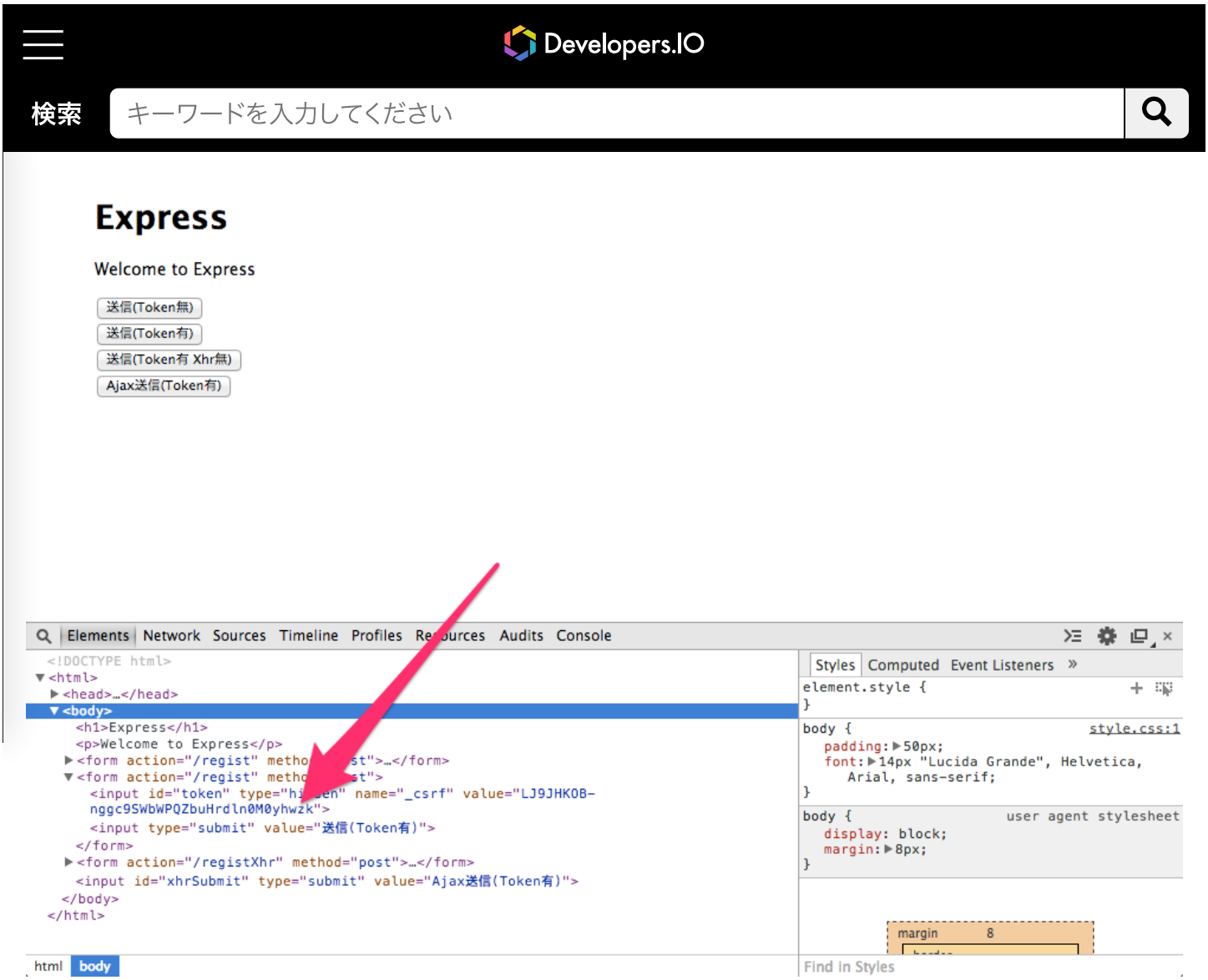
上記の実装を完了したアプリを起動します。

```
npm start
```

http://localhost:3000にアクセスし、ソースコードをみてみます。

Hiddenフィールドにtokenが埋め込まれていることが確認できるかと思います。 また、ページを再読み込みすることでtokenが更新されていることもわかるかと思います。


Developers.IO
検索
キーワードを入力してください
Express
Welcome to Express
送信(Token無)
送信(Token有)
送信(Token有 Xhr無)
Ajax送信(Token有)
Elements Network Sources Timeline Profiles Resources Audits Console
<!DOCTYPE html>
<html>
<head>…</head>
<body>
<h1>Express</h1>
Welcome to Express
<form action="/regist" method="post">…</form>
<form action="/regist" method="post">
<input id="token" type="hidden" name="_csrf" value="v4TEveNR-S4IJzFz1eVwXueY0fI1mX6IRlhc">
<input type="submit" value="送信(Token有)">
</form>
<form action="/registXhr" method="post">…</form>
<input id="xhrSubmit" type="submit" value="Ajax送信(Token有)">
</body>
</html>
Styles Computed Event Listeners
element.style {
}
body { style.css:1
padding: 50px;
font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}
body { user agent stylesheet
display: block;
margin: 8px;
}
margin 8
html body
Find in Styles



送信(TOken無)を押下すると、invalid csrf tokenといったエラーになります。 Tokenがない状態でPOSTしたためcsurf側でチェックしてエラーになったことがわかります。

また、送信(Token有)を押下すると、Tokenチェックが行われ、POSTの処理が行われたことがわかります。

_csrfTokenの内容って実は適当で良いんじゃないの？

ページ読み込むたびにTokenが変わっているってことはTokenの値って適当でもよいのでは？ と思い、 適当でよかったら困るので検証します。

上記の送信(Token有)の通信内容をChromeの開発者ツールのネットワークからcURL形式で取得し、 正しく通信できることを確認後、 Tokenを一文字書き換えて通信できるか確認します。

```
curl 'http://localhost:3000/regist' -H 'Cookie: connect.sid=s%3AbFQmT585doqOI
OK%
```

Token一文字書き換え

```
curl 'http://localhost:3000/regist' -H 'Cookie: connect.sid=s%3AbFQmT585doqOI
<h1>invalid csrf token</h1>
(握応小落)
```

tokenを一文字書き換えただけで、Tokenエラーとなっていることが確認できます。

tokenの生成や確認ルーチンについては別途記事を書きたいと思います。

踏査確認・検証(Ajax)

FormではHiddenフィールドにTokenを格納して検証を行わせていましたがAjaxを用いた場合は、リクエストヘッダーにX-CSRF-Token/X-XSRF-TokenとしてTokenをつけてAjax通信を行うことで、 CsurfはFormで通信したときと同様に、Tokenの有効/無効を検証してくれます。

jQueryを用いた場合のリクエストヘッダーのセット例は以下になります。

/public/javascripts/xhrAccess.js

```
'use strict';
$(function() {
    $.ajaxPrefilter(function(options, originalOptions, jqXHR) {
```

```
                return jqXHR.setRequestHeader('X-XSRF-Token', token);
            }
        }
    });
    function cheer() {
        var cheerPost = $.post('/registXhr', '');
        cheerPost.done(function(result) {
            alert(result);
        });
    }
    $('#xhrSubmit').click(function() {
        cheer();
    });
});
```

csurfの話からはそれるのですが、Expressのチェック機能として、Ajaxか非Ajaxかを見分けることが可能です。 実装例では、/registXhrに対するPOST通信がAjaxか非Ajaxかで分岐して出力内容を切り替えています。

よって、送信(Token有 Xhr無)を押下すると、Tokenの検証を行いXHR通信ではない処理が実行されます。 また、送信(Token有 Xhr有)を押下すると、Tokenの検証を行い、XHR通信として処理されています。

注意点

csurfはXSRF対策のtokenを払い出し、その検証を行ってくれますが、Tokenはワンタイムではありません。

よって、以下の点に注意が必要です。

- 二重投稿の抑制はできません。
- Tokenが盗まれた場合、成りすまされる危険性があります。

二重投稿はボタンの二度押し等で発生するので、正しいフォームからの投稿になります。 付与されているtokenは正しいTokenのため、 発行されたリクエストはcsurf内では正しいアクセスとし

必要があります。

それらが盗まれている状態というのは、xsrf対策とかそういう次元ではなく、 アプリケーション全体のセキュリティとして問題が発生しているという状態になります。 この点に関しては、Tokenがワンタイムの方が良いのか否かといった枠で考えるのではなく、 インフラも含め、アプリケーション全体のセキュリティが問題ないかを設計・実装段階で検討する必要があります。

まとめ

Expressを用いたWebアプリ開発のCSRF対策を探していた際に、当該のミドルウェアを探しました。

ただ、GitHubにも実装例は少なく、テストコードとReadMe読むが主な資料だったので、 実装例を含めて紹介しました。

Expressを用いる開発において、当該ミドルウェアをCSRF対策として採用することで、 かなり楽ができるかと思われます。

参考URL

Node.js のセキュリティの話 - SSSSLIDE

ソースコード

CM-Kajiwara/csurfSample