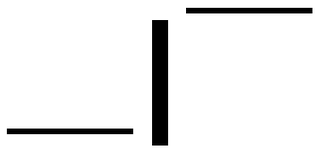


# Secure Web Coding

## ～実践から学ぶセキュアなWebの作り方～

Presented by yagihash from ISC

# yagihash #とは

- 環境情報学部4年
- ISC所属
- セキュリティ・キャンプ2013 Webセキュリティクラス
- SECCON CTF 2013 Final 7位(クソ雑魚)
- その他CTFもちょこちょこ(CSAWとか)
- 就活で攻撃力が落ちたのでリハビリ中

# お願い

- 僕自身こういう場を持つのは初めてなので、
  皆さんからたくさん勉強させてください
- #rgsecurecoding というヤギハッシュタグを
  用意しておきますので、随時思ったことなんかを
  ツイートしておいてくれれば、あとで僕が見直して
  涙を流しながら猛省するかもしれません
- あと間違ってるところとかあれば気軽に指摘してください
  (僕はまだまだ未熟者なので…)

# 対象者 and 非対象者

- 対象者
  - Webで戦うエンジニア達
  - Webで戦うエンジニア予備軍
  - その他の理由で純粋にWebセキュリティに興味がある人
- 非対象者
  - 次のスライドの「今日やること」、全部知ってるよっていう強い人
  - 危ない人

# 今日やること and やらないこと

- 今日やること
  - SQLiのお話
  - XSSのお話
  - CSRFのお話
  - その他の脆弱性のお話(ファイルアップロード関連、セッション関連とか)
- 今日やらないこと
  - ディレクトリトラバーサル, OSCi, RFIなどのお話

# なぜWebのセキュリティは重要か

- Webサービスの普及と浸透
  - Webを通じてやり取りされる情報が増えた
  - 量はもちろんのこと、その質もよりセンシティブに
  - クライアントサイドでの処理量が増加していたり
- サイバー攻撃のパラダイムシフト
  - 愉快犯ではなく金銭目的の犯行が増加
  - ビジネスなどにおいては致命的なダメージを被りかねない
  - 公共機関などが提供するWebアプリケーションの安全性も重要だったり

# なぜWebなのか

- 僕がWebしか知らないから

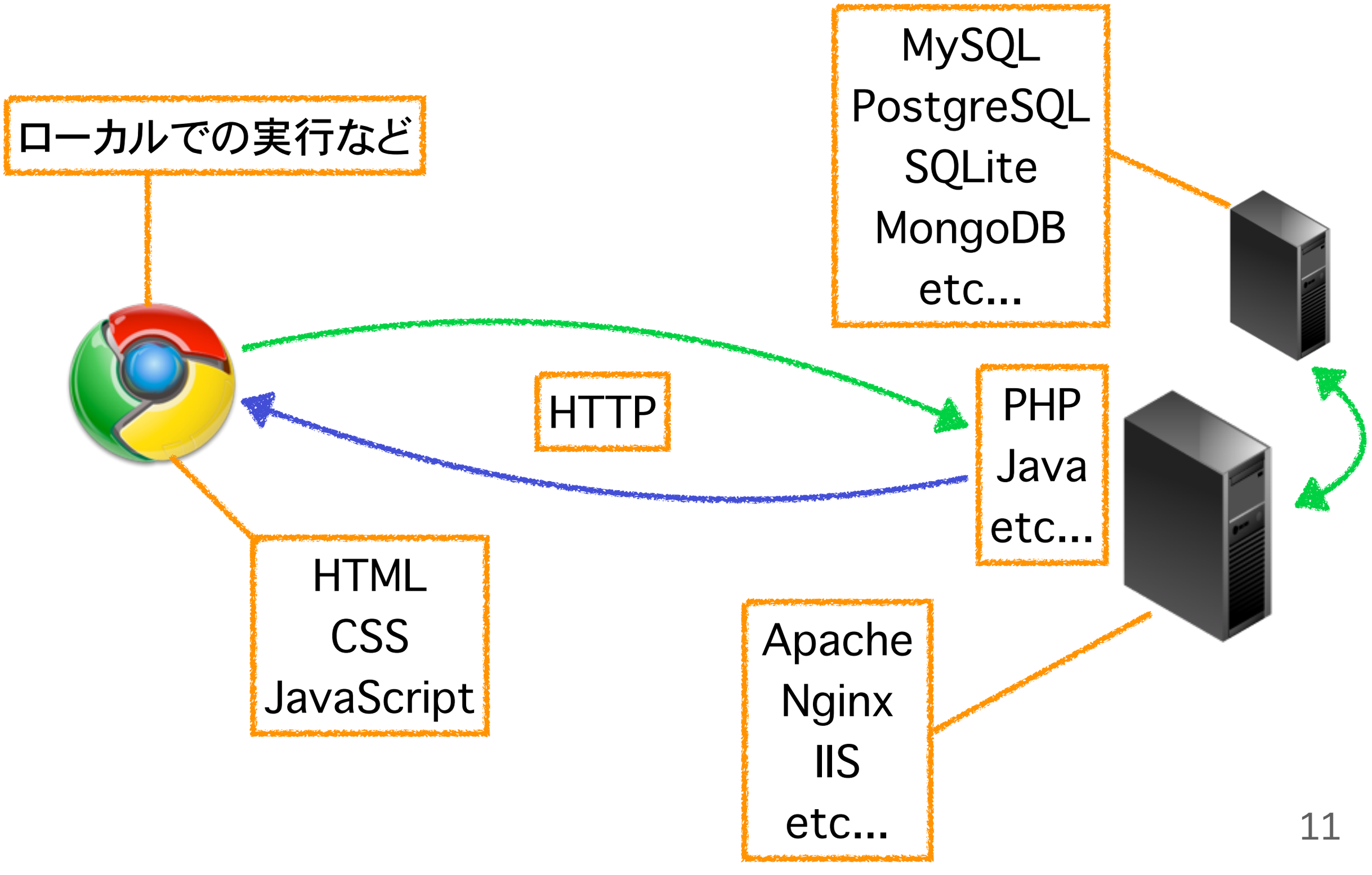
# Webじゃなければ何があるのか

- C/C++でのセキュアコーディング
  - バッファオーバーフローとか男の浪漫だよね
- Android/iPhoneでのセキュアコーディング
  - SSLの証明書の検証不備/HTMLパーサのバグとか
- OSのセキュリティ
  - DEP, ASLRでググると幸せになれる
- ネットワークのセキュリティ
  - これはもうなんといってもDNS Cache Poisoningの美学とか、スマートなDoSとか
- 組み込みのセキュリティ
  - ATMとか…?世の中のそれっぽいもの全て

# 改めて本日のメニュー

- SQLインジェクション
- XSS
- CSRF
- ファイルアップロード
- セッション関係

# ところでWebアプリって何だろう



# ところで脆弱性って何だろう

- Aさん「脆弱性は脆弱性だよ、XSSできたら危ないじゃん」
- Bさん「どうして危ないの?」
- Aさん「危ないから危ないんだよ!」
- Bさん「全てのXSSは危ないの?」
- Aさん「脆弱性は危ないから危ないんだよ!!!」
- Bさん「( ´・ω・`)」

# 攻撃者の視点を持つということ

- 脆弱性は「悪用できるバグ」
- 悪用できないバグもある
- 悪用できる"仕様"もある
- そのバグや仕様をどこまで悪用できるのか
- 攻撃側の視点を持つことで守るべきものが見えてくる
- 本気の攻撃者は良心なんて持ち合わせていない

# 本題

# 注意事項

- ここから先は、実際の攻撃手法を取り扱います
- 本資料は公開予定です
- 別段公開に際して問題のあるものは取り扱いません
- 今日ここで紹介する攻撃手法などを
  外部のサイトで試すなどの行為はおすすめしません
- 逮捕・起訴などの法的な手段によって
  相応の処分を受ける可能性があります

# 今日のサンプルなど

- https://github.com/yagihashoo/rg_secure_coding
  サンプルのソース↑
- http://web.sfc.wide.ad.jp/~yagihash/rg_secure_coding/
  実習環境URL↑
- PHP + SQLiteです(一部MySQL)
- RG/CNSのサーバなら(たぶん)動くはず
- 脆弱なコードなのでコピペ厳禁

# 受動的攻撃と能動的攻撃

- 受動的攻撃
  - 攻撃者：標的=ユーザという関係が成り立つもの
  - XSS, CSRFなどのユーザの操作を伴って攻撃が成り立つもの
- 能動的攻撃
  - 攻撃者：標的=アプリケーションという関係が成り立つもの
  - 攻撃者が直接的にサーバを叩いて攻撃を成り立たせるもの
  - SQLi, OSCiなどなど

# SQLインジェクション

- Webアプリケーションなどのバグを利用して
  バックエンドのDBに任意のクエリを発行する攻撃手法
- Webアプリケーションが発行するSQL文の構造を
  破壊されることで生じる
- DBに直接アクセスされるので、被害は大きくなりがち
- (たぶん)◯×株式会社の△□情報が流出みたいな
  世の中でよくニュースになるアレの原因はこれ

# SQLインジェクション

- よくありがちなのは、'(シングルクォーテーション)を
  入力するとなんかエラーが出て突然の死を迎えるやつ
- 今日はもう少し踏み込んだお話まで
- +αで実際に手を動かしてもらってイメージを掴む
- 対策手法は3種類ほど紹介
- これはほんとにヤバいので安易に試さないように

# パターン1: 突然の死

- /practice_sqli/sample1/
- とりあえずシングルクォーテーションで検索
- エラーが出て死にます
- 楽しい!!✌(´﹏`✌)三✌(´﹏`)✌三(✌´﹏`)✌
- エラーが出るだけじゃ全然面白くないのでもう少し遊ぶ
- よくわかんないけどadminのパスワードを抜こっか♡
- 暇な人は挙動とエラーメッセージから
  どんなSQL文なのか想像して待っててください

# パターン1: 突然の死

```
$query = "SELECT id, name FROM users WHERE name LIKE '%{$name}%' AND name != 'admin'";
```

- 15行目がダメ
  - 受け取った入力をそのままSQL文に文字列連結で挿入している
- SQLiはSQL文の構造を破壊する
  - つまりどういうこと?
- '%{$name}%'の部分で「構造が」破壊されてしまう
  - なぜ?
  - どうやって防げばいい?

# エスケープサニタイズ問題

| 元入力 | "><script>alert('xss');</script> |
|---|---|
| バリデーション | エラー：IDは英数字のみで入力してください |
| エスケープ(HTML) | "><script>alert('xss');</script> |
| エスケープ(MySQL) | "><script>alert(\'xss\');</script> |
| フィルタリング(1) | alert('xss'); |
| フィルタリング(2) | scriptalertxssscript |
| サニタイズ(1) | ___script_alert__xss_____script_ |
| サニタイズ(2) | "＞＜ｓｃｒｉｐｔ＞ａｌｅｒｔ（'ｘｓｓ'）；＜／ｓｃｒｉｐｔ＞ |

バリデーション、エスケープ、フィルタリング、サニタイズのイメージ(http://tumblr.tokumaru.org/post/69573778661)より引用

# パターン1を修正する

- /practice_sqli/sample1_modified/
- 14行目でSQLite3::escapeStringを使って
  ユーザの入力をエスケープ
- 'が''にエスケープされる
- 自分でエスケープ関数を作るなどはせず
  絶対に各DB専用のエスケープ関数を利用すること

# パターン2: 戦いは続く

- /practice_sqli/sample2/
- 今度はちゃんとエスケープしてあ…あれ?
- もう一度、adminのパスワードを抜いてみる
- 暇な人はXSSでもしててくださいw

# パターン2: 戦いは続く

- 原因は何でしょう?
  - 確かにエスケープはされている…けど。。。
  - 実際にSQLインジェクション可能な状態なのはなぜだろう
- 「構造」は場合によって異なる
  - パターン1ではシングルクォーテーションの外に自分の入力を反映させた
  - 今回はシングルクォーテーションはない
- 今回は数値をSQL文の中にベタ書きしているので…
  - 'をエスケープする意味がない

# パターン2を修正する

- 俺「数値しかこないならバリデーションでいいじゃんw」
- $id = preg_match("/^[0-9]+$/", $_GET["q"])...
- 「安全なパターン」を確実に定義できるのであれば
  ブラックリスト方式よりもホワイトリスト方式の方が安全
- ただし、上記例では特定の条件下では
  大きな落とし穴があるので要注意

# 正規表現の落とし穴

- 正規表現における^と$はそれぞれ何を指しているか
- 正規表現における\Aと\zはそれぞれ何を指しているか
- 実際のユースケースでどのような問題が想定されるか

# 正規表現の落とし穴

- 正規表現における^と$はそれぞれ何を指しているか
  - それぞれ行頭と行末を表現する
- 正規表現における\Aと\zはそれぞれ何を指しているか
  - それぞれ文字列の先頭と終末を表現する
- 実際のユースケースでどのような問題が想定されるか

# つまり……

- /^hogefuga$/は"evil string[0x0A]hogefuga"にマッチ
- ただしPHP,Perlなどではデフォルトでシングルラインモード
  - 全体を1行として扱う
  - "hogefuga\nhogefuga\n"みたいな感じ
- Rubyなどではデフォルトでマルチラインモード
  - .が\nにマッチしない
  - "hogefuga", "hogefuga"を順に見ていく感じ

# 完全マッチには\Aと\zを使う

- PHPでも/^[0-9]+$/は"100\n"にマッチしてしまう
- 改行まで含めたマッチなので、場合によっては困る
- なので、必ず/\A[0-9]+\z/で完全マッチを行う
- /practice_sqli/sample2_modified/
  は完全マッチで修正済
- 参考: http://blog.tokumaru.org/2014/03/z.html

# エスケープの限界

- いちいちエスケープのこと考えなきゃいけないの面倒臭い
- しかも一箇所忘れたらもう終わりだし…
- DB毎にエスケープ関数も違うし…
- DB毎にメタキャラクタも違うし…
- もっと簡単なのないの…( ´・ω・`)
- もっとこう、簡単で、安全で、イケてるやつ…

# あります( `・ω・´ )!!

- プレースホルダを使いましょう
- 簡単: SQL文のひな形を作って、変数を指定してあげるだけ
- 安全: 構造を先に決定し、後から変数を入れるので安全
- イケてる: これが現在の主流で、エスケープとか使わない

エスケープなんて最初からいらんかったんや！

# プレースホルダ

- プリペアドステートメント(静的プレースホルダ)
  - DBMS側で構文解析を済ませておき、後からパラメータを変更する
  - 原理上構文が変わることが有り得ないので、絶対に安全
- バインド機構(動的プレースホルダ)
  - パラメータを指定する「場所」を用意するという点では同じ
  - ただし原理上絶対に安全なプリペアドステートメントに比して、
    バインド機構そのものに脆弱性があればSQL文の構造が破壊される恐れも
  - とはいえメジャーなものはほぼほぼ安全と思ってよい

# プレースホルダ

- php-mysqlだとmysqli_stmt(静的)
- その他のDBMSだとPDOがバインド機構を提供
- MySQLしか使わないならセキュリティ面だけでなく
  パフォーマンス面でもmysqliが有利だったはず
- いろいろなデータベースを使う場合はPDOを使いましょう
- /practice_sqli/sample3
  PDO + SQLiteでのプレースホルダ使用サンプル

# SQLインジェクションまとめ

- 優先順位は
  (プリペアドステートメント = バインド機構) > エスケープ
- プレースホルダが使えない場合は
  仕方がないのでエスケープしましょう
- ただし、この手の論争の中では
  「エスケープが必要なコーディングそのものが既に間違っている」
  という意見もある(個人的には安全ならどっちでも)
- エスケープ関数は絶対に自作せず
  必ず各RDBMSに合わせて用意されている専用関数を使う
- エスケープするときは、データを受け取ったときではなく
  使う直前にエスケープするように(忘れづらいので)

# XSS(Cross Site Scripting)

- 標的のブラウザ上で攻撃者が任意のスクリプトを実行する攻撃手法、またそれを可能にする脆弱性
- その特性により大きく3種類に分けられる
- シンプルなのに奥が深い、堅揚げポテトみたいなやつ
- ぶっちゃけ全部取り上げてると時間足りないから今日は「とりあえずこれをやってくれ、頼む」案件だけ取り扱います

# XSSでできること、できないこと

- できること
  - JavaScript(VBScript)でできること、全て
  - httponly属性の付与されていないセッションIDの窃取
  - 認証済ユーザにしか閲覧権限のない情報の窃取
  - 任意のセッションIDの付与
  - キーロギング
- できないこと
  - JavaScript(VBScript)でできないこと、全て

# Reflected XSS

- ざっくり言うとGETのXSS
- http://example.com/?q=<script>alert(1)</script>などのURLに標的を誘導しなければならない
- 誘導が必要という点において攻撃のハードルが高め

# Stored XSS

- ざっくり言うとゴキブリホイホイ
- あるページ上で持続的に任意のスクリプトが動作し続ける
- 例えば掲示板の投稿など
- ユーザが存在していて、一定レベルのアクセスがあれば
  待ち伏せているだけでよい
  →ハードルは低め
- （ただ誰もアクセスしてこないようなところだと反射型と一緒）

# DOM Based XSS

- 最近アツいXSS
- JavaScriptでのDOM操作時に発生する
- 基本的な対策はその他のXSSと同じ
- 何点か性質が違うのでそこだけ覚えておく

# 基本原理

- ユーザ入力の各コンテキストへの出力における
  メタキャラクタのエスケープ漏れ
- つまるところ、ページへの出力時に
  メタキャラクタだけ適切にエスケープしとけば大丈夫
- はいはい、簡単簡単
- うん、簡単、うん
- 楽しい!!✌(´﹏`✌)三✌(´﹏`)✌三(✌´﹏`)✌

# 何がそんなに難しいの(´・ω・`)

- 簡単に「各コンテキスト」って言ったけど…
  - HTMLコンテキストでのエスケープ
  - JavaScriptコンテキストでのエスケープ
  - CSSコンテキストでのエスケープ

- 出力時のエスケープだけじゃ不十分な場合も!?
  - Content-Sniffingの闇
  - Content-Typeヘッダの正しい出力
  - X-Content-Type-Options: nosniff

# 各コンテキストでのエスケープ

- 各コンテキストでのメタキャラクタが違う
- CSSなんかは少し調べるとわかるけど解釈の仕方が
  マジで混沌としているのでCSS内出力はおすすめしない
- JavaScript内出力も禁じ手とされている
- 各コンテキストに合わせた出力を毎回忘れずに、
  間違えずに、ひとつも漏らさず確実に行えますか?
- あ、エスケープは出力の直前にやりましょう
  (忘れるしわかりにくいし、直前にやらないと文字列操作しにくいし)

# Always Keep It Simple

- 出力は全てHTMLコンテキストにおいて行う
- HTMLにおけるメタキャラクタは<, >, ", ', &のみ
- 上記5種のメタキャラクタをページ上に生成されなければこっちの勝ち
- それぞれ<>"'&などにリプレース
- 属性値は必ず"か'で囲う(後述)
- IEだと属性値を`(バッククォート)で囲うこともできるけどそんな書き方する人いないよね???

# Always Keep It Simple

- JSコンテキストにおいて動的コンテンツが必要な場合は?
  - HTMLコンテキストに出力→DOM操作で取得ってのがベストプラクティスっぽい
  - あとはXHRで取得とかでもよいのでは
  - とにかく<script><?php echo $_GET["q"]; ?></script>的なことはやめる
- HTMLタグを許可した実装にしたいけどXSSは防ぎたい
  - なんかいいライブラリがあるらしいっすよ
  - 後述のCSPを使えば防ぐこともできるかもしれないけど、おすすめはしないです

# エスケープ関数

- PHPならhtmlspecialchars()を使う
  - 第2引数にENT_QUOTESを指定
  - 第3引数に"UTF-8"など、そのページへの出力時のエンコーディングを指定
- JavaScriptには標準ではエスケープ関数はない
  - 基本的にはテキストノードとして出力すること
  - document.createTextNode()やjQueryの.text()など
  - どうしても自分でエスケープする場合は先のスライドのメタキャラクタをリプレース

# なぜ属性値を囲うのか

- /practice_xss/sample1/
- 関数h($s)でHTMLコンテキストでのエスケープ
- メタキャラクタは正しくエスケープされているが…
- 「a onfocus=alert(1)」などでアラートが出る
- 属性値は"か'で囲いましょう( `・ω・´ )

# Content-Sniffingの闇

- Content-Sniffingという便利機能があります
- コンテンツの中身をチェックして、勝手にそれらしいコンテンツとして処理してくれる素晴らしい機能です
- いくつかの素晴らしい脆弱性を我々に提供してくれました
- 「UTF-7 XSS」や「IE Content-Type XSS」なんかでググると幸せになれるかもしれません

# Content-Sniffingの闇

- とりあえずこれだけやってくれ案件
- 正しいContent-Typeヘッダの出力
  - Content-Type: text/html; charset=UTF-8
  - 赤字の部分が死ぬほど大事
    (てかこれってXSS対策以前の問題だと思うので普通にやっておこう)
  - metaタグは補助的なもので、Content-Typeヘッダが優先
- X-Content-Type-Options: nosniffの付加
  - Content-Sniffing無効化
  - トレードオフも特にないので全レスポンスにつけてくれ、頼む

# 僕「対策めんどくさ…もうええやろ…」

- Content Security Policyを使いましょう
- コンテンツの読み込み元を制限する
- 正しく使えばXSS撲滅可能(たぶん)
- CSPそのものの脆弱性を突かれない限りはおそらく大丈夫
- MDN(Mozilla Developer Network)なんかに詳細が
- これもかなり高機能なのでキリがなくなる前に切ります

# XSS Auditor、知ってる?

- ChromeのXSSフィルタ
- ちなみにIEにもXSSフィルタあり
- X-XSS-Protectionヘッダで操作可能
  - X-XSS-Protection: 0  => XSSフィルタ無効
  - X-XSS-Protection: 1  => XSSフィルタ有効
  - X-XSS-Protection: 1; mode=block  => XSSフィルタ有効(ブロックモード)

# XSSまとめ

- 大原則 → 出力時にエスケープ
- 大原則 → 出力先はHTMLコンテキスト
- 大原則 → HTTPレスポンスヘッダは正確に
- +α → CSP使える状況なら積極的に使う
- +α → X-XSS-Protectionヘッダも使ってみる
- ちなみにXSSだけで90分丸々使えるような気がするのでいろいろ自分でも調べてみてください
- HTTPレスポンスヘッダとかはまた後でまとめます

# XSSのついでに

- Cookieにはsecure属性やhttponly属性というのがある
- secure属性
  - HTTPSの通信でしか送信されなくなる
  - 盗聴のリスクの緩和
- httponly属性
  - HTTPレベルでの取り扱いのみ可能になる(≒JavaScriptからアクセスできない)
  - XSSでセッションIDそのものを抜いてくるような攻撃に対して有効な対策

# エスケープのサンプルコード

- 一応対策例をサンプルコードの中に入れておきます
- これらはコピペしてもおっけーです
- /practice_xss/sample_escaping/
- ?q=<s>a</s>#<s>a</s>とかしてみてください

# CSRF

- 罠ページを通して、標的に別サイトへの意図しない操作を行わせる攻撃手法(表現が難しいです、すいません)
- 例えば、標的が罠ページ(evil.net)を踏むと、ECサイト(ec.example.com)で美味しいお肉が100kg注文されてしまうとか
- ユーザの認証を行うサイトにおけるPOSTでは一律に全て対策するのが望ましい
- 特に認証などを行わないサイトでは、好きなように(犯罪予告事件みたいな不要なトラブルは避けられるかも)

# CSRF

- XHR Lv.2ではファイルの送信が可能になった
- CSRFでの攻撃の幅が広がっている
- Ex.) CSRFで不正ファイルアップロード→任意コード実行
- みたいな教科書的なお話もどっかにあるかもしれない
- ただし、別段対策手法が変わるわけではない

# CSRFの対策

- あるリクエストの正当性が検証できればそれでよい
- CSRF対策トークンを各リクエストに付与してあげましょう
- CSRF対策トークンは必ずしもワンタイムトークンでなくても構わないけど、最低でも1セッションに1トークン
- セッションIDをCSRF対策トークンとして用いるのは……
- トークンは予測不可能で十分な長さを持つものに

# 安全なトークン #とは

- 乱数には質のよいものとそうでないものがある
- いわゆる暗号論的擬似乱数生成器で良質な乱数を得る
- PHPならopenssl_random_pseudo_bytes()を使う
- トークンの長さはいろいろ見てると16進数32桁が多い
- bin2hex(openssl_random_pseudo_bytes(16))
  これでおっけー

# 実際の書き方(例)

送信側

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<?php
if (!isset($_SESSION["token"])) {
    $token = bin2hex(openssl_random_pseudo_bytes(16));
    $_SESSION["token"] = $token;
}
?>
<form method="POST" action="hoge.php">
    <input type="hidden" name="token" value="<?php echo $_SESSION["token"]; ?>" />
    <input type="hoge" name="hoge" />
    <input type="fuga" name="fuga" />
    <input type="submit" />
</form>
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

受信側

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<?php
if (!isset($_POST["token"])   or
    is_array($_POST["token"]) or
    $_POST["token"] !== $_SESSION["token"])
    die("token error");

$hoge = $_POST["hoge"];
$fuga = $_POST["fuga"];
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

だんだん雑になってきた(現在1:45)

# だいたい図も何もない時点でわかりにく(ry

# 残りのコンテンツ

- ファイルアップローダで気をつけること
- セッション管理関連
- 今日やったことに関係するHTTPレスポンスヘッダまとめ
- 質問タイム

- 「これだけやってくれ、頼む」案件方式でやっていきます

# ファイルアップローダ

- 例えばこういうやり方(サンプルコードにもあります)
  https://gist.github.com/yagihashoo/11223679
- 例えばphpファイルを実行されると、
  そこを通じて任意コード実行可能になる
- 例えば画像ファイルに偽装したスクリプトをContent-Sniffingを利用して(ry
- 意外と怖いファイルアップローダでした(雑

# アップロードされたファイルの扱い

- ファイル名はユーザに決めさせない方がよいかも
- "../"とか"..\"とかいろいろ気をつけるの面倒だし…
- X-Download-Options: noopenでIE8以降でブラウザ上でファイルを開く機能を制限できる
- Content-Typeヘッダは確実に想定しているMIMEタイプで
- X-Content-Type-Options: nosniffも合わせてどうぞ
- 理想を言えばアップロードしたファイルの閲覧はダウンロードが一番イケメンだと思う

# で、何すりゃいいのよ

- ファイルサイズとMIMEタイプをしっかり確認すること
- ファイル名は可能ならこちらで指定してしまうこと
- アップロードされたファイルの閲覧には気を使うこと
  - Ex.) PHPファイルを通してのみ閲覧可、強制的にPDFとして閲覧(よくないかも)
  - Ex.) 一旦ダウンロードさせて、手元で閲覧させるような実装
- 面倒臭がらずにきちんとやること
  - 任意コード実行が通ると一撃でかなりのダメージになる
  - カーネルレベルの脆弱性があったりするとrootまで取られかねない

# セッション周り

- ログイン後に
  session_regenerate_id(TRUE)してください♡
- 標的のブラウザに任意のセッションIDを植え付けられる
  →ログインさせる→植えつけたセッションID使ってアクセス
  →セッションハイジャック成功！
- ってのを防ぐために、ログイン後にセッションIDを振り直す
- ある人曰く
  「セッションIDなんて毎リクエストごとに変えればいいんだよ」
  らしいので僕はそうしてます

# HTTPレスポンスヘッダまとめ

- いろいろ紹介していきます
- どれも有用なものなので、TPOに合わせて使ってください

# まとめ1

- X-Content-Type-Options
  - nosniffとすることでContent-Sniffingを制限
- X-XSS-Protection
  - 各ブラウザに備わっているXSSフィルタを制御
  - 特に理由がなければX-XSS-Protection: 1; mode=blockでよい
- Content-Security-Policy
  - コンテンツの読み込み元を制限することでXSS対策を万全に
  - MDNのページが非常に詳細なのでそちらを参照してください

# まとめ2

- X-Frame-Options
  - クリックジャッキングという攻撃手法を防止するための機能
  - 特に理由がなければDENYかSAMEORIGINでよいのでは
- Strict-Transport-Security
  - ブラウザに安全にHTTPS使用を促す機能
  - HTTP→HTTPSにリダイレクトすると最初のHTTPでのリクエストの際に
    セッションIDなどが危険に晒される可能性がある
    HSTSを使うと最初からHTTPSに
- Set-Cookie
  - secure属性やhttponly属性を忘れないように(特にhttponly)

# 参考書籍

- 「体系的に学ぶ安全なWebアプリケーションの作り方」
  - http://www.amazon.co.jp/dp/4797361190
- 「めんどうくさいWebセキュリティ」
  - http://www.amazon.co.jp/dp/4798128090
- 「HTTPの教科書」
  - http://www.amazon.co.jp/dp/479812625X

# 参考サイト

- 「安全なウェブサイトの作り方」
  - http://www.ipa.go.jp/files/000017316.pdf
- 「葉っぱ日記」
  - http://d.hatena.ne.jp/hasegawayosuke/
- 「徳丸浩の日記」
  - http://blog.tokumaru.org/

# 布教

- もっとセキュリティについて知りたいと思った人、挙手
- 今年もセキュリティキャンプが開催されます！
- もうすぐ募集が始まると思うので、
  http://www.security-camp.org/
  とかをチェックしておいてください！
- 迷ってる暇があったら出してみた方が早いです

# スライドはここで終わっている……