Carsten Eilers / www.ceilers-it.de

Sicherheit von Anfang an

Vorstellung

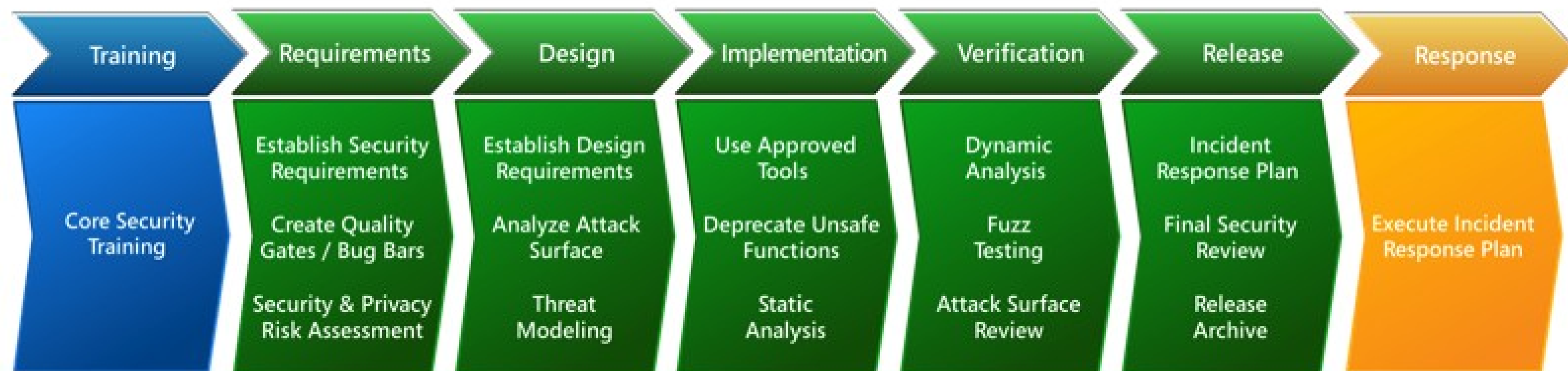
- Berater für IT-Sicherheit
  - Web Security (Pentests, Beratung, ...)
  - ...
- Autor
  - PHP Magazin, Entwickler Magazin
  - Blog: www.ceilers-news.de
  - ...
- Schwachstellen- und Exploitsammlung

Agenda

- **Microsofts SDL**
- Sicherheit von Anfang an
- Ab in die Praxis
- Zurück zur Theorie

Microsofts SDL (1)

Security Development Lifecycle



Microsofts SDL (2)

- Sieht kompliziert und aufwändig aus, ist aber sehr erfolgreich
- Erstes nach SDL entwickeltes System: Vista
  - Nach einem Jahr 45% weniger Schwachstellen als in XP

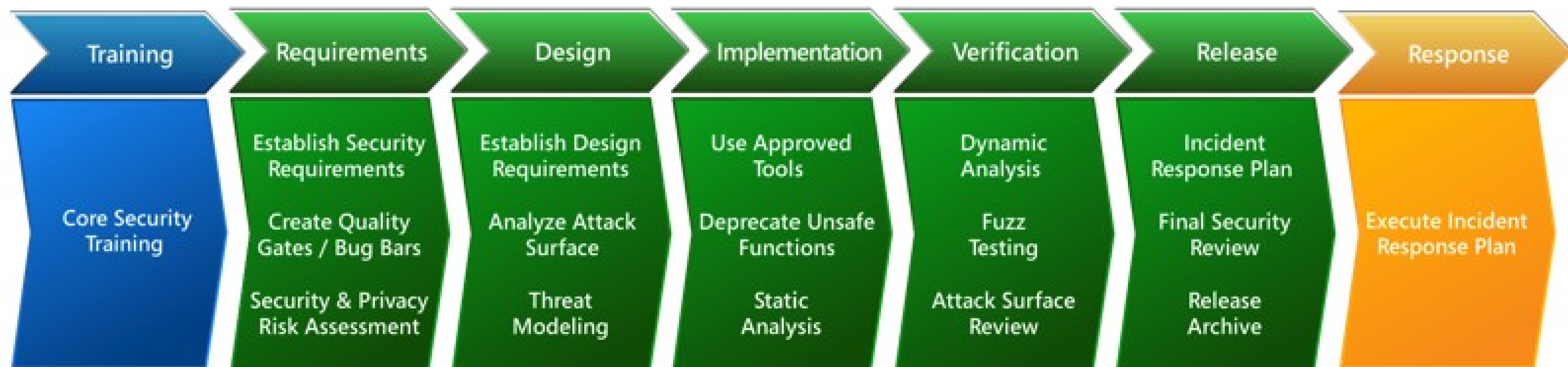
Microsofts SDL (3)

7 Phasen:

1. Training
2. Requirements
3. Design
4. Implementation
5. Verification
6. Release
7. Response

Microsofts SDL (4)

Phase 2 bis 6 je 3 Schritte



Microsofts SDL (5)

- 4 „Best Practices“ als Grundlage
  1. Secure by Design
     Sicherheit als Entwurfsziel
  2. Secure by Default
     Sichere Default-Konfiguration
  3. Secure by Deployment
     Sichere Default-Installation
  4. Communications
     Reden Sie über Schwachstellen!

Agenda

- Microsofts SDL
- **Sicherheit von Anfang an**
- Ab in die Praxis
- Zurück zur Theorie

Sicherheit von Anfang an (1)

Schritt 1:

Wissen, über was man redet

Informieren Sie sich über Schwachstellen und mögliche Angriffe.

Einmal relativ ausführlich als Grundlage,

danach gilt *„Bleiben Sie auf dem Laufenden“*

Sicherheit von Anfang an (2)

Schritt 2:

Analysieren Sie die Bedrohungen für Ihre Neuentwicklung – und zwar bevor Sie sie implementieren!

„Thread Modeling“
„Bedrohungsmodellierung“

„Threat Modelling“

1. Vision
   Wo und wie wird das Programm eingesetzt?
2. Model
   Diagramm der Anwendung
3. Identifiy Threats
   Welche möglichen Bedrohungen gibt es?
4. Mitigate
   Minimieren Sie die Bedrohungen
5. Validate
   Überprüfen Sie das Ergebnis

„Identifiy Threats“

STRIDE

- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation Disclosure
- **D**enial of Service
- **E**levation of Privilege

Sicherheit von Anfang an

Wer hat Lust auf weitere Kategorien, Listen, Phasen...?

Agenda

- Microsofts SDL
- Sicherheit von Anfang an
- **Ab in die Praxis**
- Zurück zur Theorie

Ab in die Praxis

Wir entwickeln ein Gästebuch:

Eintragen:

Ihr Name:	Ihr Name	
Ihre E-Mail:	Ihr E-Mail	(wird nicht angezeigt)
Ihre Homepage:	Ihre Homepage	(optional)
Titel:	Der Titel	
Ihre Nachricht:	Ihr Eintrag	
	Reset Eintragen	

Das Gästebuch (1)

- Alle Benutzer dürfen Einträge schreiben und Teile der Daten lesen
- Admins dürfen alle Daten lesen
- Nur für Admins sichtbar:
  E-Mail-Adresse
  IP-Adresse des Eintragenden

Das Gästebuch (2)

Parameter:

- Name `$name`
- URL der Homepage `$link`
- Titel `$titel`
- Gästebucheintrag `$eintrag`
- Nur für Admins zugänglich:
  - E-Mail-Adresse `$mail`
  - IP-Adresse `$_SERVER['REMOTE_ADDR']`

Das Gästebuch (3)

Funktionen:

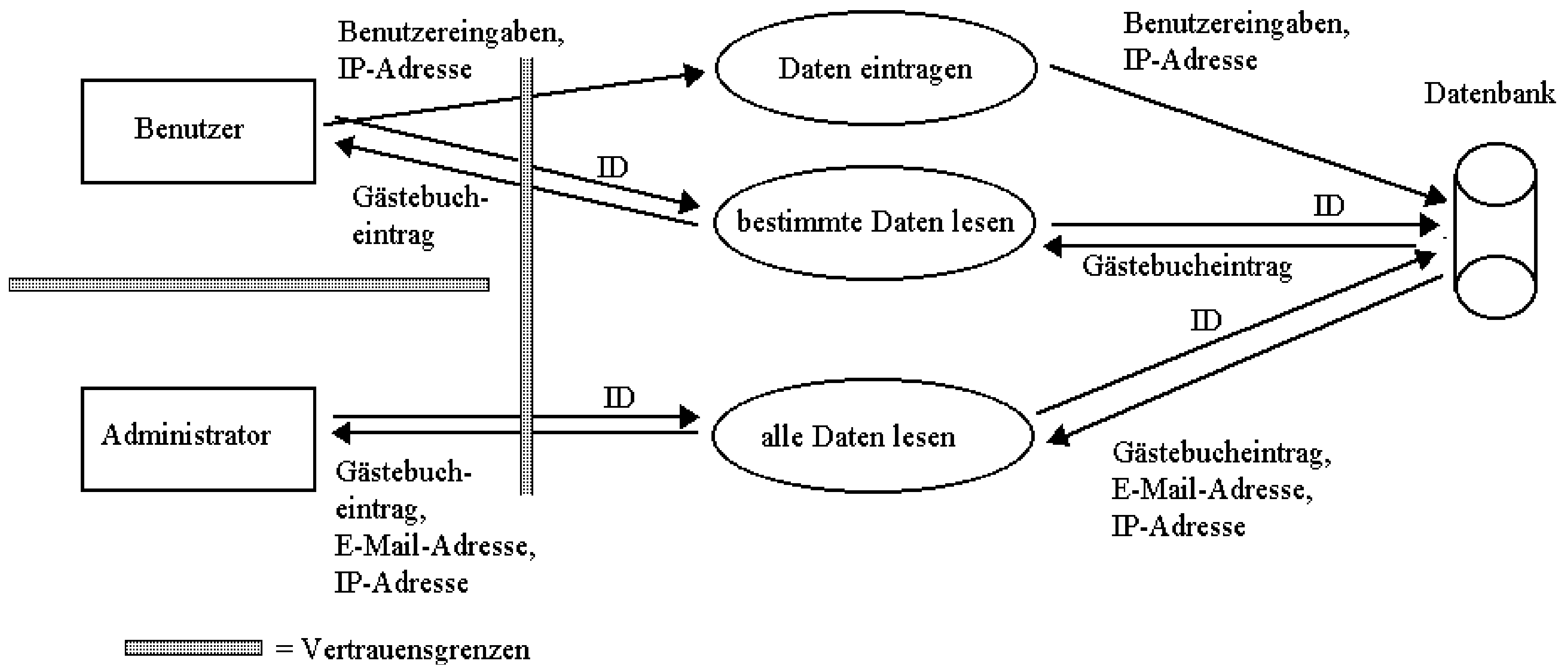
- `eintragen($titel, $eintrag, $name, $link, $mail, $ip)`
- `anzeigen($id)`
- `admin($id)`
- `auflisten()`

Bedrohungsmodellierung (1)

Diagramm:

- Kreise für Code
- Kästen für externe Dinge wie Personen, Server usw.
- Zylinder für Datenbanken etc.
- Pfeile für Datenflüsse

Bedrohungsmodellierung (2)



Bedrohungsmodellierung (3)

Vertrauensgrenzen (offiziell):

Mehr als eine Person oder Prozess greifen auf ein Objekt zu

Bedrohungsmodellierung (4)

Für jeden Datenfluss, der eine Vertrauensgrenze überschreitet, alle Bedrohungen ermitteln

=> STRIDE

STRIDE `eintragen()` (1)

- Spoofing:
  - Für Einträge irrelevant,
  - IP-Adresse zur Tarnung
- Tampering:
  - SQL-Injection,
  - XSS,
  - `$email` SMTP-Injection
- Repudiation:
  - irrelevant

STRIDE `eintragen()` (2)

- Information Disclosure:
  - Da nur eingetragen wird, nicht möglich
- Denial of Service:
  - Übermäßiger Ressourcenverbrauch durch präparierte Eingaben
- Elevation of Privilege:
  - Mangels Privilegien nicht möglich

STRIDE `anzeigen()` (1)

- Spoofing:
  - nicht möglich
- Tampering:
  - SQL-Injection über `$id`,
  - XSS über `$id` in SQL-Fehlermeldung
- Repudiation:
  - nicht möglich

STRIDE `anzeigen()` (2)

- Information Disclosure:
  - Aufruf mit beliebiger ID, nur bei Moderation relevant
- Denial of Service:
  - Durch große Anzahl von Abfragen
- Elevation of Privilege:
  - Mangels Privilegien nicht möglich

STRIDE `admin()` (1)

- Spoofing:
  - Angreifer kann sich als Admin ausgeben
- Tampering:
  - SQL-Injection über `$id`,
  - XSS über `$id` in SQL-Fehlermeldung
- Repudiation:
  - nicht möglich

STRIDE `admin()` (2)

- Information Disclosure:
  - Nicht möglich, da Admin alles sehen darf
- Denial of Service:
  - Durch große Anzahl von Abfragen
- Elevation of Privilege:
  - Benutzer kann Admin-Rechte erlangen

STRIDE & Datenschutz

Eins fehlt in STRIDE:
Der Datenschutz

Soll hier nicht weiter interessieren, aber:
Brauchen Sie die IP-Adresse?

In diesem Fall: Eigentlich nicht.

Schutz: `eintragen()` (1)

Mögliche Angriffe verhindern:

- Spoofing:
  - Gefälschte IP-Adresse ist nicht erkennbar

Schutz: `eintragen()` (2)

- Tampering:
  - `$titel, $eintrag, $name`:
    - SQL-Injection: Escapen oder Prepared Statement mit parametrisierten Aufruf
    - XSS: Filtern
  - `$link`:
    - Ebenso oder auf Korrektheit prüfen
  - `$mail`:
    - Auf Korrektheit prüfen, verhindert auch SMTP-Injection
  - `$_SERVER['REMOTE_ADDR']`:
    - Auf Korrektheit prüfen

Schutz: `eintragen()` (3)

- Denial of Service:
  - Eingaben auf Größe prüfen,
  - CAPTCHA verhindert automatisiertes Überfluten mit Einträgen

Schutz: `anzeigen()` (1)

- Tampering:
  - `$id` kann in Integer umgewandelt werden, danach keine Manipulation möglich
- Information Disclosure:
  - Wenn moderiert wird, auf Zulässigeit prüfen
- Denial of Service:
  - CAPTCHA verhindert automatisierte Abfragenflut

Schutz: `admin()` (1)

- Spoofing:
  - Sichere Authentifizierung nötig
- Tampering:
  - `$id` kann in Integer umgewandelt werden, danach keine Manipulation möglich
- Denial of Service:
  - CAPTCHA verhindert automatisierte Abfragenflut
- Elevation of Privilege:
  - Sichere Authentifizierung und Session-Verwaltung

Das Ergebnis (1)

Das sieht schlimm aus - aber das macht nur die Präsentation

Wir haben

- eine Hand voll Parameter
- zwei Hände voll Schutzmaßnahmen

Parameter	Angriff	Schutzfunktion / Gegenmaßnahme
$titel	XSS	• HTML-Tags ausfiltern
	SQL Injection	• Eingabe escapen oder • Prepared Statement mit parametrisierten Aufruf verwenden
$eintrag	XSS	• HTML-Tags ausfiltern
	SQL Injection	• Eingabe escapen oder • Prepared Statement mit parametrisierten Aufruf verwenden
$name	XSS	• HTML-Tags ausfiltern
	SQL Injection	• Eingabe escapen oder • Prepared Statement mit parametrisierten Aufruf verwenden
$link	XSS	• HTML-Tags ausfiltern oder • Korrektheit prüfen
	SQL Injection	• Eingabe escapen oder • Prepared Statement mit parametrisierten Aufruf verwenden oder • Korrektheit prüfen
$mail	XSS	• HTML-Tags ausfiltern oder • Korrektheit prüfen
	SQL Injection	• Eingabe escapen oder • Prepared Statement mit parametrisierten Aufruf verwenden oder • Korrektheit prüfen
	SMTP Injection	• Korrektheit prüfen
$_SERVER['REMOTE_ADDR']	XSS	• Korrektheit prüfen
	SQL Injection	• Korrektheit prüfen
$id	Information Disclosure	• ggf. prüfen, ob der ausgewählte Eintrag ausgegeben werden darf (zur Zeit nicht relevant, da keine Moderation stattfindet)
	alle anderen	• In Integer umwandeln

Das Ergebnis (2)

War das alles?

Ist das Ergebnis korrekt?

Ja!

Unerwünschtes

Fast:

Spam und allg. unerwünschte Einträge wurden nicht berücksichtigt

Passt nicht in STRIDE

Aber die CAPTCHA gegen DoS helfen auch gegen Spam

Die Umsetzung

Schritt 3: Implementierung

- Möglichst bewährte Lösungen
- Neu-/Eigenentwicklungen sorgfältig testen

Seien Sie konsequent!

Wenn Sie sich für eine Lösung entschieden haben, bleiben Sie dabei:

- z.B. entweder Escapen oder Prepared Statements – nie beides gemischt
- Schutzmaßnahmen möglichst gebündelt & immer an der gleichen Stelle
  - Gleich kommt ein schlechtes Beispiel!

Implementierung (1)

Einziges Beispiel: `$eintrag`

XSS:

```
htmlentities(strip_tags($_POST["eintrag"]));
```

oder

```
strip_tags($_POST["eintrag"], ″<p>,<b>″);
```

oder

BBCode

Implementierung (2)

SQL-Injection

```
// Prepared Statement vorbereiten:
$statement = mysqli_prepare($link, "INSERT INTO
text_tabelle (Eintrag) values (?)");
// Parameter daran binden:
mysqli_stmt_bind_param($statement, 's', $eintrag);
// Parameter füllen:
$eintrag = htmlentities(strip_tags($_POST["eintrag"]));
// Vorbereitetes Statement ausführen:
mysqli_stmt_execute($statement);
```

Implementierung (2)

SQL-Injection

```
// Prepared Statement vorbereiten:
$statement = mysqli_prepare($link, "INSERT INTO text_tabelle
(Titel, Eintrag, Name, Link, Mail, IP) values
(?, ?, ?, ?, ?, ?)");
// Parameter daran binden:
mysqli_stmt_bind_param($statement, 'ssssss', $titel, $eintrag,
$name, $link, $mail, $ip);
// Parameter füllen:
$titel = htmlentities(strip_tags($_POST["titel"]));
...
// Vorbereitetes Statement ausführen:
mysqli_stmt_execute($statement);
```

Implementierung (3)

Das für alle Parameter wiederholen

=> Implementierung (fast) fertig

Was fehlt?

Die Authentifizierung!

Implementierung (4)

- Authentifizierung einfach:
  Eigenes Skript in eigenem Verzeichnis, dafür HTTP-Basic-Authentication
- Authentifizierung anspruchsvoll:
  Eigene Passwortabfrage & Sessionverwaltung
  - Neue Parameter => ab nach oben!

Agenda

- Microsofts SDL
- Sicherheit von Anfang an
- Ab in die Praxis
- **Zurück zur Theorie**

Zurück zur Theorie (1)

Was haben wir bisher gemacht?

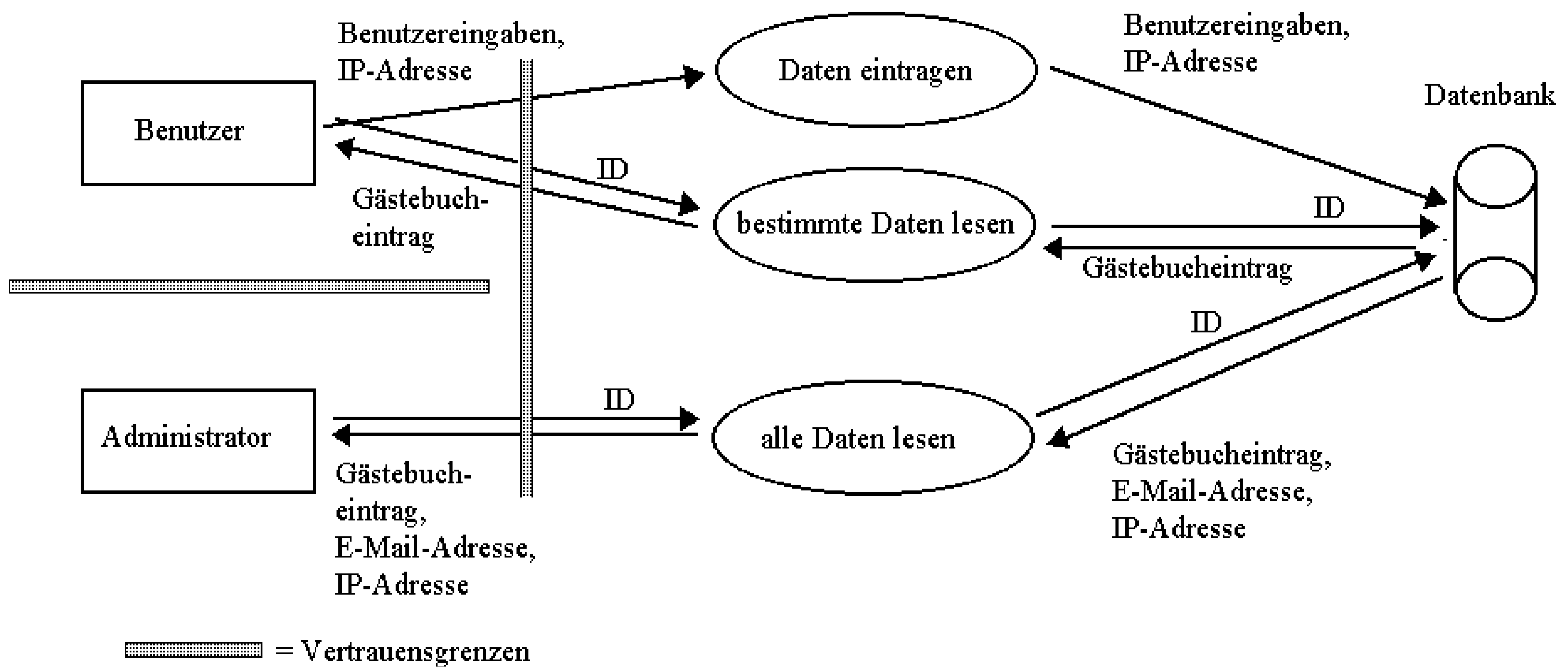
1. Angriffe/Schwachstellen gelernt
2. Bedrohungsmodellierung
3. Implementierung

Zurück zur Theorie (2)

Nochmal zur Bedrohungsmodellierung:

Muss die so aufwändig sein?

Zurück zur Theorie (3)



Zurück zur Theorie (4)

Alter Grundsatz der Web-Sicherheit:
„Traue nie dem Client!“

oder besser:

„Traue keinen Daten von Dritten!“

Zurück zur Theorie (5)

Der Vorteil der Webanwendung:

Alles, was wir nicht selbst machen, ist potentiell bösartig!

Folgerung:

Alle Daten, die wir von Dritten bekommen, müssen wir prüfen!

Zurück zur Theorie (6)

Bedrohungsmodellierung à la Web:

- Für jeden Eingabeparameter STRIDE durchgehen
- Für Tampering eine Liste mit Angriffen aufstellen
  (siehe z.B. Security Workshop im PHP Magazin 2.2012)
- Ggf. Authentifizierung/Autorisierung absichern
  (Wer ist das? / Darf der das?)

Zurück zur Theorie (7)

Was haben wir bisher gemacht?

1. Angriffe/Schwachstellen gelernt
2. Bedrohungsmodellierung
3. Sichere Implementierung

War es das?

Zurück zur Theorie (8)

Die Hälfte ist geschafft:

1. Angriffe/Schwachstellen gelernt ✓
2. Bedrohungsmodellierung ✓
3. Sichere Implementierung ✓
4. Sichere Default-Installation & -Konfiguration
5. Schwachstellentests
6. Kommunikation!

Per Default Sicher (1)

- Früher bei PHP z.B. kein `register_globals`
- Welche Rechte werden benötigt? Minimalismus rulez!
- Haben alle Verzeichnisse und Skripte die richtigen Rechte?
- Wird das Installationsskript nach der Installation automatisch gelöscht?
- ...

Per Default Sicher (2)

Machen Sie die Konfiguration so einfach wie möglich!

Lassen Sie einen ~~DAU~~Laien testen!

Per Default Sicher (3)

Es gibt keinen Fehler, der nicht gemacht wird - und am Ende ist Ihre Anwendung Schuld, wenn der Server kompromittiert wird!

Zurück zur Theorie (9)

Was haben wir bisher gemacht?

1. Angriffe/Schwachstellen gelernt ✓
2. Bedrohungsmodellierung ✓
3. Sichere Implementierung ✓
4. Sichere Default-Installation & -Konfiguration ✓
5. Schwachstellentests
6. Kommunikation!

Schwachstellentests (1)

Prüfen Sie für jeden Parameter (und jedes Skript), ob wirklich alle ermittelten Angriffe abgefangen werden.

Testen Sie mit Tools & manuell!

Schwachstellentests (2)

I.A. keine schlechte Idee:

zusätzlicher Test mit unsicherer Konfiguration

Worst-Case - Szenario

Zurück zur Theorie (9)

Was haben wir bisher gemacht?

1. Angriffe/Schwachstellen gelernt ✓
2. Bedrohungsmodellierung ✓
3. Sichere Implementierung ✓
4. Sichere Default-Installation & -Konfiguration ✓
5. Schwachstellentests ✓
6. Kommunikation!

Kommunikation (1)

Schwachstellen entstehen nicht, wenn sie gefunden werden, sondern wenn sie programmiert werden!

Kommunikation (2)

Ihre Anwendung,

Ihr Fehler,

Ihre Schwachstelle!

Akzeptieren Sie ihre Verantwortung!

Kommunikation (3)

Wird eine Schwachstelle gefunden, sind Sie u.U. der letzte, der davon erfährt!

Machen Sie das Melden einfach!

Kommunikation (4)

Warten Sie nicht, bis die Schwachstelle zu Ihnen kommt!

Beobachten Sie zumindest die Exploit-DB

Kommunikation (5)

Wird Ihnen eine Schwachstelle bekannt, informieren Sie die Benutzer!
(Bevor es ein anderer tut!)

Auf Website & Mailingliste!

Kommunikation (6)

Informieren Sie Ausführlich!

- Was kann passieren?
- Workaround
- Patch(termin)

Zurück zur Theorie (11)

Was haben wir bisher gemacht?

1. Angriffe/Schwachstellen gelernt ✓
2. Bedrohungsmodellierung ✓
3. Sichere Implementierung ✓
4. Sichere Default-Installation & -Konfiguration ✓
5. Schwachstellentests ✓
6. Kommunikation! ✓

6 Schritte zum sicheren Programm

1. Angriffe/Schwachstellen kennen
2. Bedrohungsmodellierung
3. Sichere Implementierung
4. Sichere Default-Installation & -Konfiguration
5. Schwachstellentests
6. Kommunikation!

REPEAT UNTIL...

Und bei Änderungen geht es bei Punkt 2 von vorne los...

Bsp. Gästebuch:

Bei Moderation muss diese Funktion vor CSRF geschützt werden

Oder noch einfacher:

Alles was Sie entwickeln, wird ein Angreifer gegen sie verwenden.

Überlegen Sie bei jedem Schritt auch, wie er sich missbrauchen lässt.

Sorgen Sie dafür, dass das nicht möglich ist.

Fragen?

Vielen Dank...

... für Ihre Aufmerksamkeit

Material und Links auf

www.ceilers-it.de/konferenzen/

The End