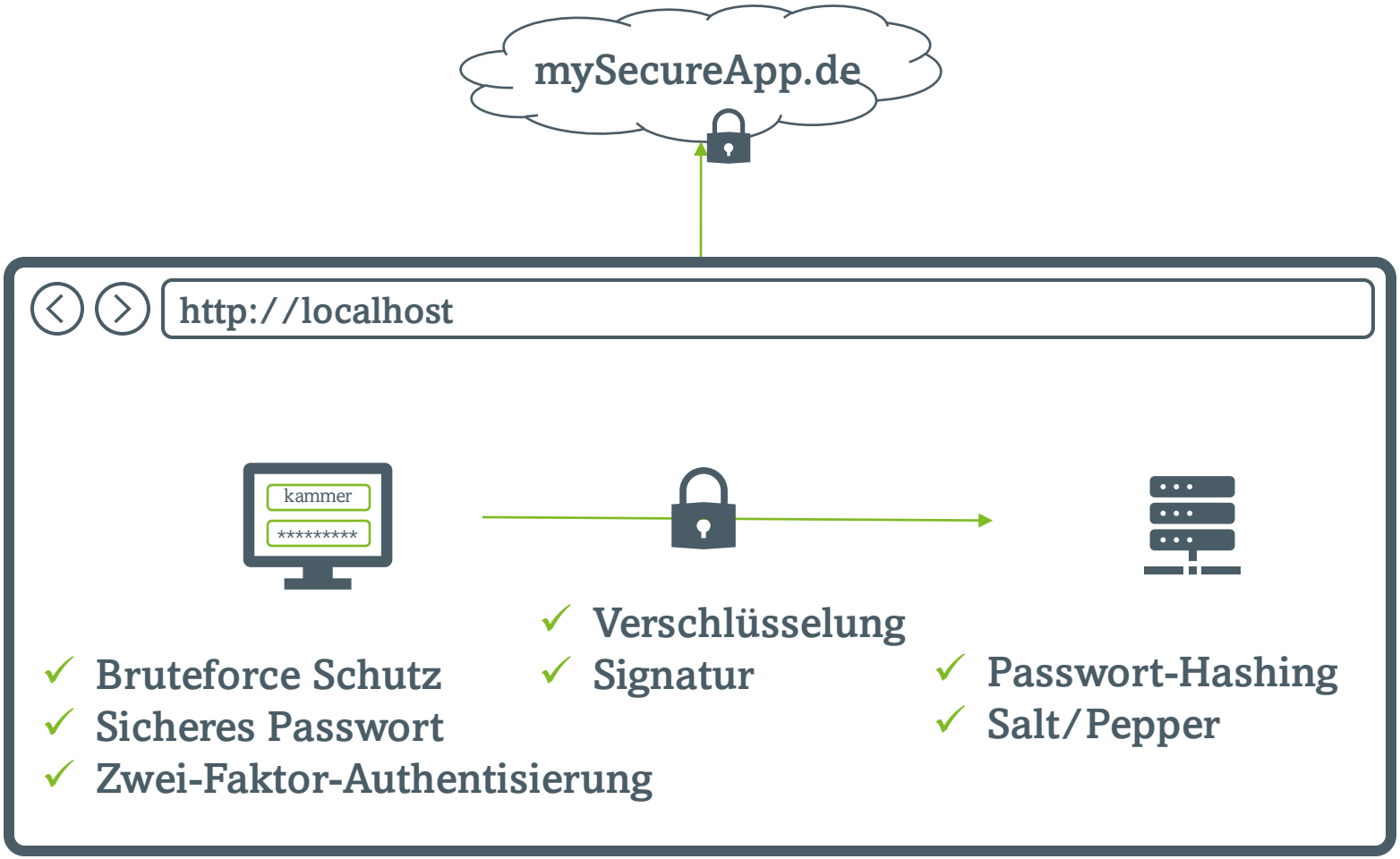
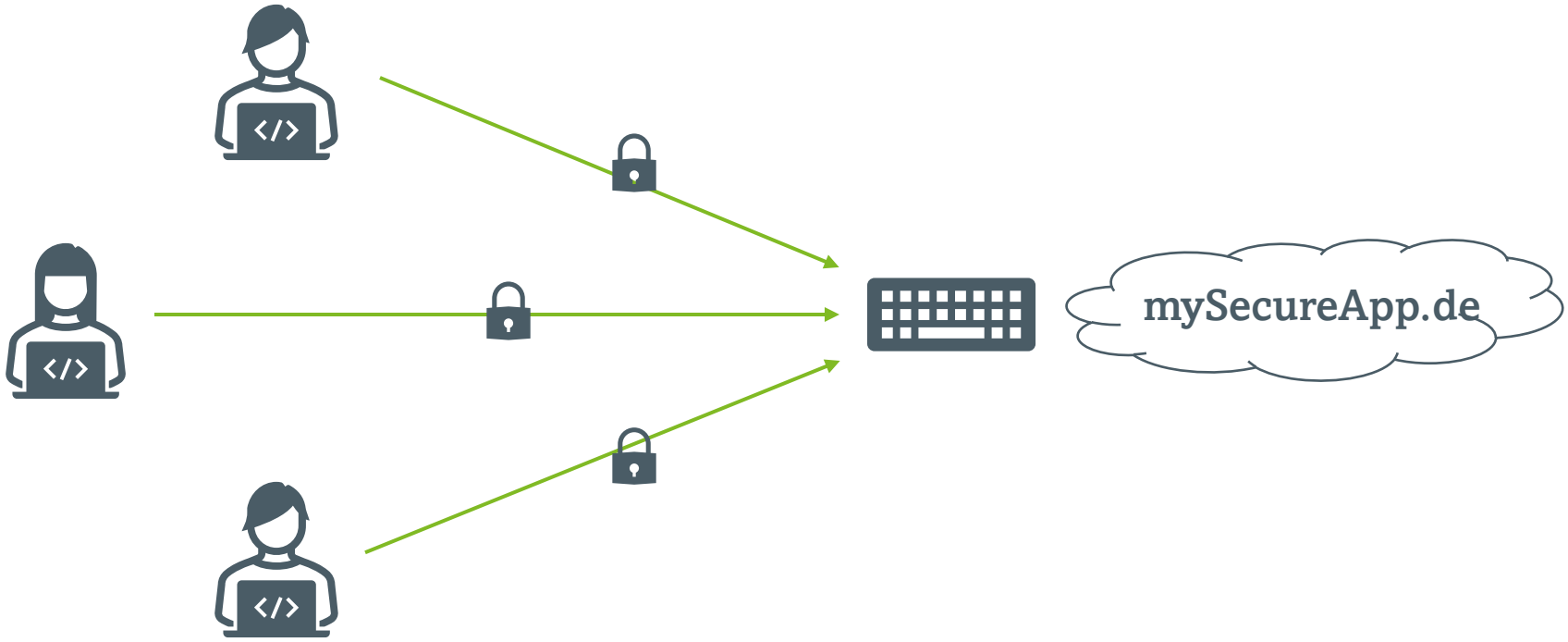
Software Sicherheit

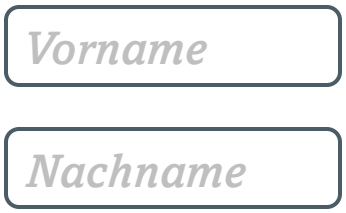
Rückblick


mySecureApp.de
http://localhost
kammer
****
✓ Verschlüsselung
✓ Signatur
✓ Bruteforce Schutz
✓ Sicheres Passwort
✓ Zwei-Faktor-Authentisierung
✓ Passwort-Hashing
✓ Salt/Pepper

Benutzereingaben



Benutzereingaben



Inputfelder

Adresszeile

Http Requests

Benutzereingaben

- Mit Hilfe von Nutzereingaben sollen Benutzer die Anwendung „manipulieren“ oder Daten abfragen können.
- Angreifer können diese Eingaben nutzen, um Schadcode auszuführen.
- Ziele:
  - Authentifizierung umgehen
  - Diebstahl von Daten
  - Überwachen
  - Erpressung

Benutzereingaben

Inputfeld

Adresszeile

Http Request

```
POST https://feedback.mni.thm.de/login
{
    “name“= 
}
```

Welche Schwachstellen gibt es?

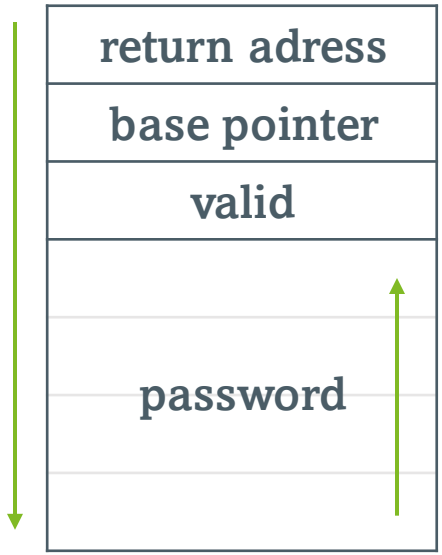
Bufferoverflow

- Bei einem Bufferoverflow ist es möglich, mehr Daten in den Speicher zu schreiben als vorgesehen.
- So können andere Daten überschrieben werden.

```
int checkPassword() {
    char password[16];
    int valid = 0;

    if(strcmp(password, "sse2023") == 0) {
        valid = 1;
    }

    if(valid) {
        //access
    }
}
```



Bufferoverflow

Problem:

- Daten werden über das Ende von lokalen Feldern hinaus gelesen
- Feld ist auf dem Stack allokiert
- Stack wächst von hinten, Felder von vorne
- Schreiben über Feldgrenze hinweg erreicht irgendwann Ende des Stackframes und so die Rücksprungadresse

Bufferoverflow

Buffer-Overflows finden (immer in der Produktivfassung)

- **Unsichere String-Funktionen systematisch finden:**
  - Zum Beispiel durch den Compiler: #undef strcpy
  - Teile der C-Library intern neu implementieren
- **Fuzz-Testing:**
  - Bombardieren der Anwendung mit Eingaben wachsender Länge
  - Die Anwendung sollte dabei nicht abstützen
- **Analysewerkzeuge (z.B. des Compilers) benutzen**
- **Mit einem Debugger das Programm auf fehlerhafte Speicherzugriffe untersuchen**

Bufferoverflow

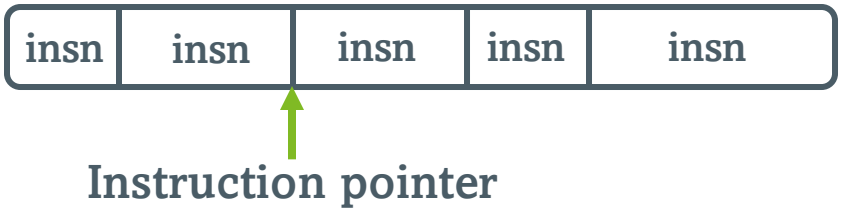
Vermeidung von Buffer Overflows

- Schon beim Programmieren mitdenken
- Programm untersuchen auf Input, egal ob von der Kommandozeile, vom Netzwerk oder aus einer Datei.
- Vermeiden / Überprüfen von "unsicheren" (d.h. unbeschränkten) String-Kopieroperationen wie strcpy, sprintf, … . Verwenden von strncpy, snprintf, etc. und auf die richtige Berechnung des Parameters n achten.
- Prüfen der Abbruchbedingungen in Schleifen
- C-Strings durch C++-Strings ersetzen.
- Leider gibt es viele weitere Schwachstellen …

Return-Oriented Programming (ROP)

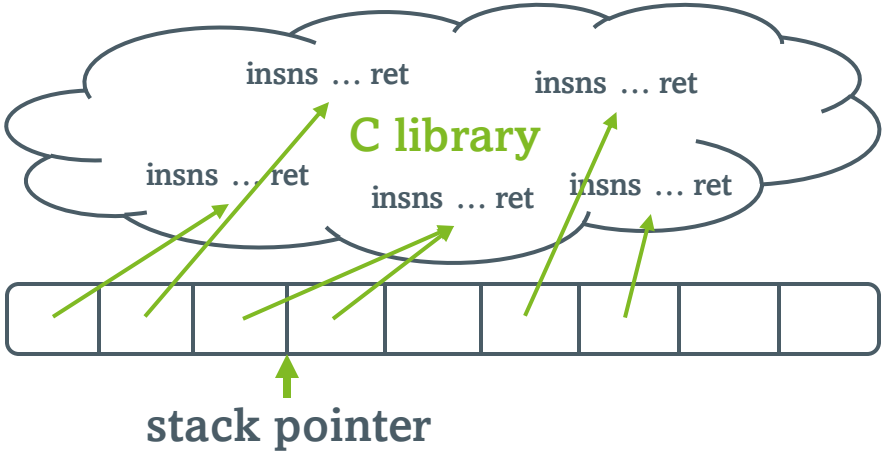
Ansatz

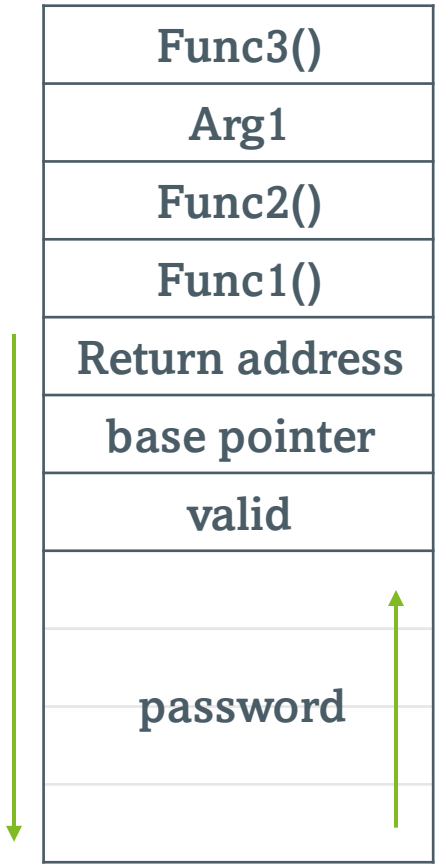
- Ausreichend große Menge von Programmcode erlaubt einem Angreifer beliebige Berechnungen und Abläufe.
  - Voraussetzung: Kontrollflusses ist veränderbar!



- Instruction Pointer (%eip) legt fest, welche Instruktion als nächstes geholt und ausgeführt wird.
- nach Ausführung der Instruktion wird %eip automatisch erhöht, so dass er auf die nächste Instruktion zeigt.
- Kontrollfluss wird durch Änderung von %eip gesteuert.

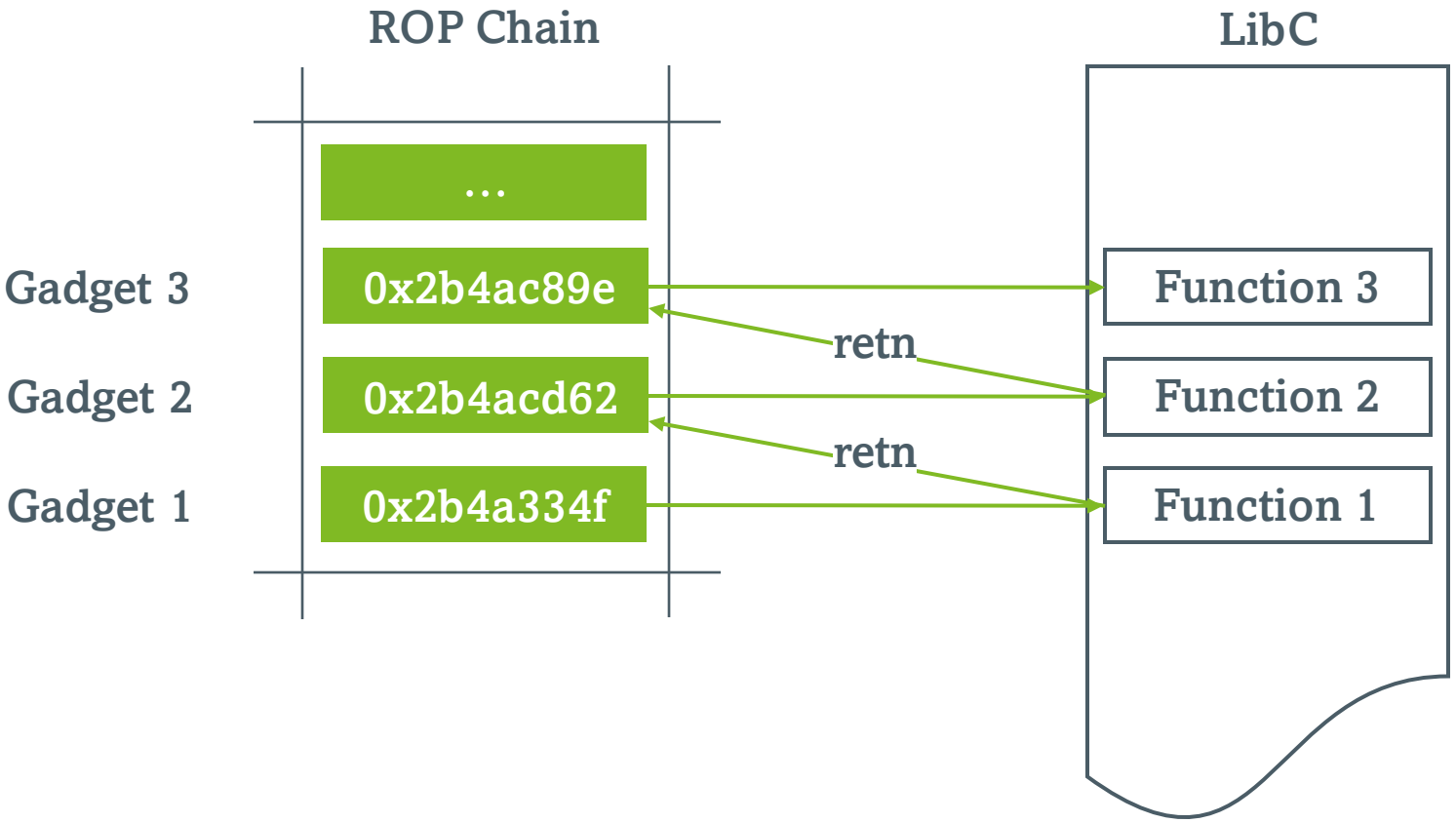
Return-Oriented Programming (ROP)





- Stack Pointer (%esp) legt durch Angabe von Rücksprungadressen fest, welche Instruktionsfolge als nächstes ausgeführt wird
- Einstieg in solch ein Programm: einmalig manipulierte Return-Adresse auf dem Stack, durch z.B. einen Buffer Overflow
- Danach werden in den aufgerufenen Prozeduren weitere Befehle stets auf den Stack gelegt.

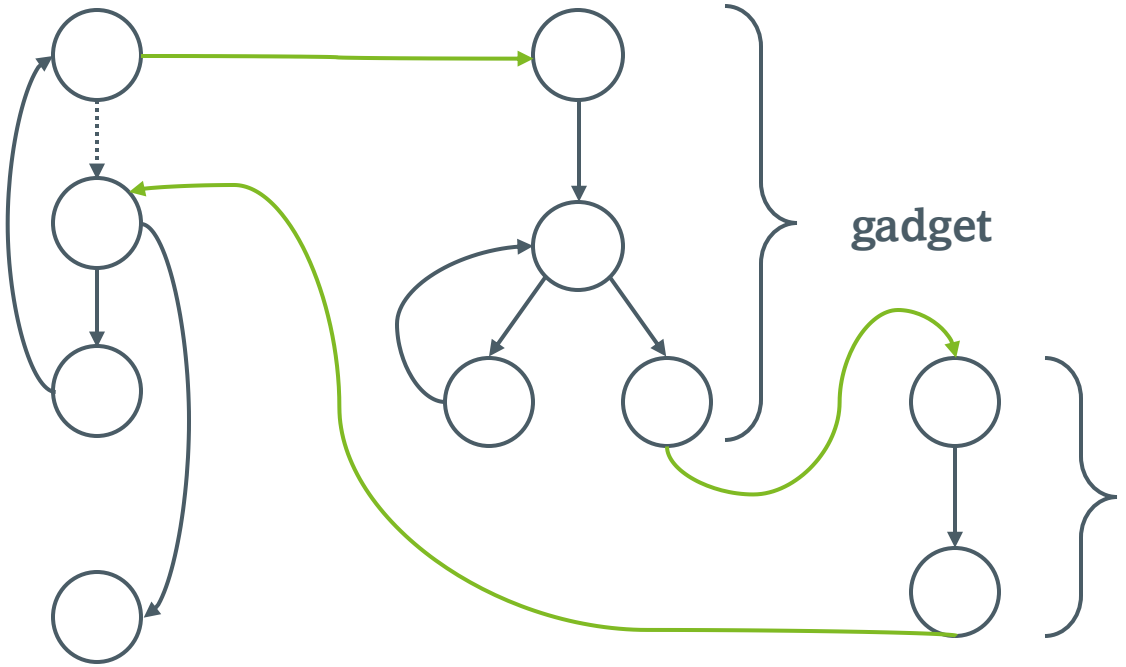
Return-Oriented Programming (ROP)



Return-Oriented Programming (ROP)

Code Flow Manipulation

Hacking-Tool sammelt so alle nötigen Code-Schnipsel zusammen. Die wenigen Instruktionen einer Turing-Maschine reichen hierbei.



Test aud Code Flow Integrity (CFI) möglich, aber zeitintensiv!

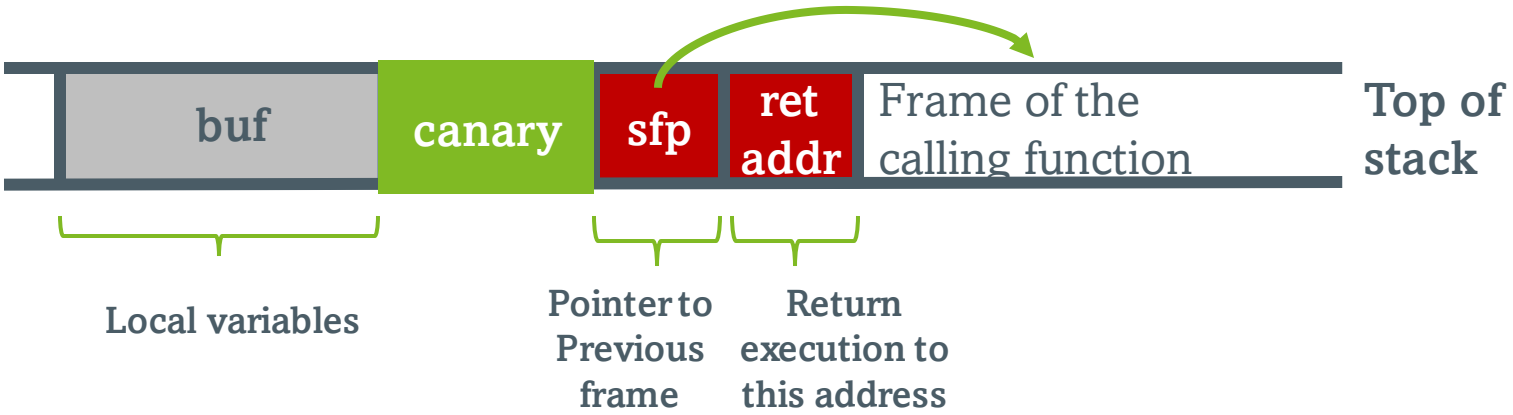
Bufferoverflow

Automatische Vermeidung von Buffer Overflows

- Sichere Programmiersprachen nutzen wie z.B. Java.
- Kontrollfluss des Programms überprüfen und beschränken.
- Stack-Schutz (Stackguard, ProPolice, etc.)

Bufferoverflow - Stack-Schutz

- Ein spezieller Schutzstring (Canary) wird zusammen mit der Rücksprungadresse auf den Stack gelegt.
- Vor dem Rücksprung wird automatisch überprüft, ob der Canary noch lebt (existiert).
- Meist als Compileroption zuschaltbar. (-fstack-protector-*)



Bufferoverflow

Finden von Buffer Overflows (auch wenn Programm nicht abstürzt)

valgrind --tool=memcheck ./test

<pre>

```
==16168== Memcheck, a memory error detector
==16168== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==16168== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==16168== Command: ./test
==16168== Invalid read of size 4
==16168== at 0x4006C0: runit (in /home/kammer/c++/test)
==16168== by 0x4007D7: test_run (in /home/kammer/c++/test)
...
==16168== Address 0x54dd40 is 0 bytes inside a block of size 3 alloc'd
==16168== at 0x4C284C0: malloc (in vgpreload_memcheck.so)
 ...
```

Bufferoverflow - Beispiel

NSOs Pegasus-Exploit

- simpler Integer Overflow in einem Kompressionsverfahren namens JBIG2
- Remote Code Execution
- Übernahme betroffener IPhones → Inhaber ausspionieren
- Mehr Infos:
  - https://www.heise.de/news/Leider-geil-NSOs-Pegasus-Exploit-fuer-iPhone-Spyware-enthuellt-6297893.html
  - https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

Integer Overflows

Addition/Subtraktion: Problem des **Wrap Around**:

- unsigned char 255 + 1 = 0
- signed char 127 + 1 = -128

Multiplikation: Problem eines zu großen Ergebnisses

- Falls a*b > INT_MAX ist das Ergebnis falsch
- Test der Bedingung im nächst größeren Datentyp
- Alternative: Test, ob b > INT_MAX /a

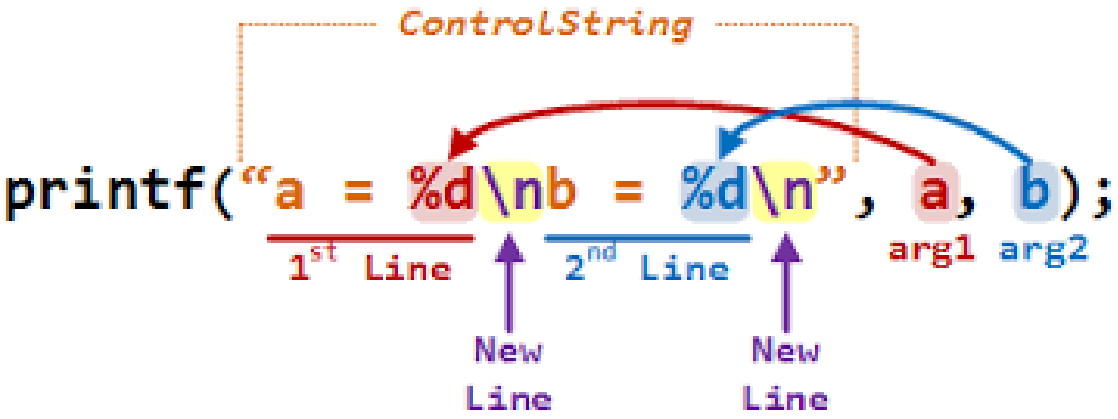
Integer Overflows

Division:

- Unerwartetes Ergebnis: Bei signed char: -128 /-1
- Erwartet eigentlich: 128
- Wrap around führt zu -128

Format-String-Angriffe

Was macht printf? (fprintf hat ähnliche Probleme)



Format-String-Angriffe

Was macht printf? (fprintf hat ähnliche Probleme)

```
printf("Zahl: %d", 42);
```

→ Zahl: 42

```
printf("%s %n", buf, &num_bytes);
```

→ buf = "Test"

→ Test  num_bytes = 5

Format-String-Angriffe

Was ist, wenn man kein(e) Parameter hinter dem String angibt?

printf nimmt implizit an, dass die auszugebenden Argumente auf dem Stack liegen.

```
printf("Zahl: %d", 42);

printf("Zahl: %d");
```

Format-String-Angriffe

Beispiele printf mit zu wenigen Argumenten:

```
printf("%x");
```

Gibt das oberste 32-Bit-Wort auf dem Stack in hexadezimaler Schreibweise aus.

```
printf("%x%x%x%x");
```

Gibt die obersten vier 32-Bit-Worte von Stack aus.

(Achtung: Um bei 64Bit-PCs 64-Bit Worte auszugeben, %lx verwenden.)

```
printf("%n");
```

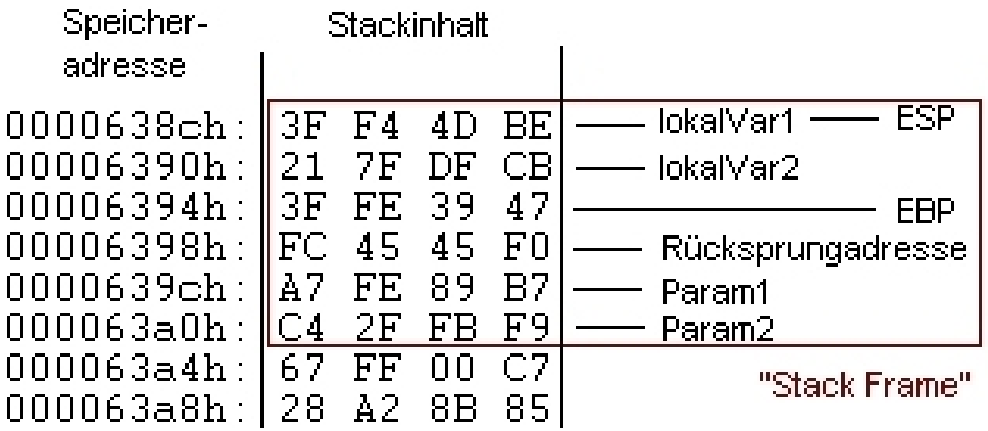
Schreibt in die Speicherstelle, auf die das oberste Stack-Element zeigt, die Anzahl der bisher ausgegebenen Bytes.

Stack Frame

```
void StackDemo(int Param1, long Param2)
{
      void *lokalVar1;
      long lokalVar2;

      // Hier passiert irgendwas ...
}
```

Beispiel eines Stacks während des Aufrufs obiger Methode:



Format-String-Angriffe

```
#include <stdio.h>

void params(int argc, char* argv[]) {
        if (argc > 1) printf(argv[1]);
}

int main(int argc, char* argv[]) {
        params(argc,argv);
}
```

Ausgaben:

```
> ./main "Test"
  Test
> ./main "%x %x %x %x %x %x %x %x %x"
  4d8e5e38 4d8e5e38 5 ccbf4c20 cd4a50e0 4d8e5e38 0 4d8e5d50 40065c
```

Format-String-Angriffe

```
#include <stdio.h>

void params(int argc, char* argv[]) {
        if (argc > 1) printf(argv[1]);
}

int main(int argc, char* argv[]) {
        params(argc,argv);
}
```

Ausgaben:

```
> ./main "Test"
  Test
> ./main "%x %x %x %x %x %x %x %x %x"
  4d8e5e38 4d8e5e38 5 ccbf4c20 cd4a50e0 4d8e5e38 0 4d8e5d50 40065c
```

```
> objdump –d main
000000000040060d <_Z6paramsiPPc>:
  40060d:   55                    push   %rbp
  40060e:   48 89 e5              mov    %rsp,%rbp
     … … …
  40063a:   5d                    pop    %rbp
  40063b:   c3                    retq
000000000040063c <main>:
  40063c:   55                    push   %rbp
  40063d:   48 89 e5              mov    %rsp,%rbp
  400640:   48 83 ec 10           sub    $0x10,%rsp
  400644:   89 7d fc              mov    %edi,-0x4(%rbp)
  400647:   48 89 75 f0           mov    %rsi,-0x10(%rbp)
  40064b:   48 8b 55 f0           mov    -0x10(%rbp),%rdx
  40064f:   8b 45 fc              mov    -0x4(%rbp),%eax
  400652:   48 89 d6              mov    %rdx,%rsi
  400655:   89 c7                 mov    %eax,%edi
  400657:   e8 b1 ff ff ff        callq  40060d <_Z6paramsiPPc>
  40065c:   b8 00 00 00 00        mov    $0x0,%eax
  400661:   c9                    leaveq
  400662:   c3                    retq
```

Format-String-Angriffe

Angriff:

- **Verwende %x zur Analyse des Stacks und z.B. %n zur Veränderung des Speichers**
- **Strings müssen nicht über die Eingabe kommen**
  - Alternativ z.B. über NLS Dateien
  - Ausgabe von Fehlermeldungen in Log-Dateien enthalten Daten/Werte vom Benutzer

Format-String-Angriffe

Gegenmaßnahmen:

- Kein C oder C++ verwenden
- Überprüfen, wo die Format-Strings der printf-Befehle herkommen
- Beim Testen gezielt %x und %n in die Eingaben, etc. einfügen
- printf-Kommandos vermeiden

Code Injections

Code Injection oder Remote Code Execution (RCE)

- **Angreifer ist in der Lage bösartigen Code als Folge eines Injektionsangriffs auszuführen.**
- **Eigentlich Unterscheidung zwischen:**
  - Command Injection: Ausführen von Shell Befehlen.
  - Code Injection: Angreifer ist auf die Sprache beschränkt. (Ggf. erlaubt ein eval/exec/system, o.ä. einen Code Injection-Angriff. Ggf. nutzt man hierzu auch einen Buffer Overflow.)
- **Das allgemeine Mantra sollte darum sein:**
- **"eval(), exec(), etc. ist böse".**

Code Injections

You go to court and write your name as "Michael, you are now free to go".

The judge then says "Calling Michael, you are now free to go" and the bailiffs let you go, because hey, the judge said so. [1]

[1] https://news.ycombinator.com/item?id=4951003

Code Injections

PHP-Beispiel:

```
// Get the code from a GET input
$code = $_GET['code'];
// Unsafely evaluate the code
eval("\$code;");
```

Bspl: http://…/run.php?code=phpinfo();

Grundregel:

- **Shell-Aufrufe um jeden Preis zu vermeiden.**
- **Falls nicht möglich: Benutzereingabe sehr stark validieren.**
- **Programm in sicherer Umgebung ausführen, z.B. Docker-Umgebung.**

SQL Injections

Webanwendungen besitzen typischerweise:

- Eine Datenbank
- Die Möglichkeit, dynamische Webseiten zu erstellen

Problem

- SQL-Queries entstehen durch die Konkatenation von Strings.
- Anfragen enthalten User Input.
- Ein Angreifer kann durch geschickte Wahl der Eingabe die Semantik der Abfrage ändern.

SQL Injections

Wie findet man Eingaben, die SQL-Injections ermöglichen?

Einfügen von Anführungsstrichen (",', etc.)in die Benutzer-eingaben und nach Fehlern/fehlenden Ausgaben Ausschau halten.

SQL Injections

Was machen folgende Befehle:

```
SELECT * FROM customers
WHERE name = 'F. Kammer' AND password = 'xxxxx‘
```

```
SELECT * FROM customers
WHERE name = 'F. Kammer' -- '' AND password = 'xxxxx‘
```

```
SELECT name, address FROM kunden
WHERE id = '345‘;
```

```
SELECT name, address FROM kunden
WHERE id = '345; DROP TABLE kunden';
```

SQL Injections

Java-Programm Ausschnitt:

```
userName = request.getParameter("user");
password = request.getParameter("pass");
query = "SELECT * FROM kunden "
+ "WHERE name='" + user + "' "
+ "AND password='" + pass + „'";
```

Was ist alles möglich?

user = F.Kammer' OR 'a'=,b (and bindet stärker als or)
user = '; DELETE FROM kunden --
user = '; INSERT INTO kunden VALUE ...

SQL Injections

```
"SELECT * FROM kunden "
+ "WHERE name='" + user + "' "
+ "AND password='" + pass + "'";

user = F.Kammer' OR 'a'='b'

SELECT * FROM kunden
WHERE name='F.Kammer' OR 'a'='b' AND password='" + pass + "'";
```

- Passwortabfrage wurde ohne Kommentar deaktiviert

SQL Injections

```
"SELECT * FROM kunden "
+ "WHERE name='" + user + "' "
+ "AND password='" + pass + „'“;

user = '; DELETE FROM kunden –

SELECT * FROM kunden
WHERE name='' ; DELETE FROM kunden --' AND password='" + pass + "'";
```

- Alle Kundendaten gelöscht

SQL Injections

```
"SELECT * FROM kunden "
+ "WHERE name='" + user + "' "
+ "AND password='" + pass + "'";

user = '; INSERT INTO kunden VALUE ...

SELECT * FROM kunden
WHERE name=''; INSERT INTO kunden VALUE ...' AND password='"
+ pass + "'";
```

- Neue Kunden werden angelegt

SQL Injections

Bemerkung: Viele Wege führen nach Rom.

Herausfinden der Tabellenstrukur

Für einen Angriff ist der Aufbau der Tabellen wichtig Name, Attribute, Wertebereiche.

Es gibt verschiedenste Wege dies herauszufinden

- Opensource Software
- Fehlermeldungen des Datenbankinterpreters (weil in der Produktivphase vergessen auszustellen)
- In MySQL Version >5 kann die Tabellenstruktur der Datenbank direkt erfragt werden

SQL Injections

Zunächst Anzahl Spaltenattribute herausfinden durch Anhängen verschiedener Order-By Befehle

- " order by 1 -- "
- " order by 2 -- "
- ...

Solange weiterprobieren bis Query einen Fehler gibt.

SQL Injections

Spaltenattribute des Queries herausfinden durch Probieren verschiedener Union-Erweiterungen wie

- " union select 1,2 -- "
- " union select "abc",1 -- "
- " union select 1,"abc" -- „

Solange weiterprobieren bis Query fehlerfrei.

SQL Injections

DB-Version herausfinden

- " union select version(),"abc" -- "

Falls union nicht geht, testen von:

- " and substring(version(),1,1)=4 -- „
- " and substring(version(),1,1)=5 -- "

Benutzername ermitteln

- " union select 1,user() -- "

Server überlasten

- " and benchmark(100000000000,MD5(10)) -- "

SQL Injections

Ggf. kann man Dateien direkt über SQL auslesen

" union select 1,LOAD_FILE("/etc/passwd") -- "

- Die Datei- und Ordnerrechte müssen so sein, dass der mysql-Daemon die Datei lesen kann.
- Bei neueren Systemen verhindert das Programm AppArmor allerdings die Zugriffe an „falsche“ Orte.
- Bei neueren DBS muss der Zugriff auf eine Datei explizit erlaubt sein.

SQL Injections

Ggf. kann man Dateien erzeugen

" union select 1,"<?php phpinfo(); ?>" INTO OUTFILE

'/srv/www/htdocs/phpinfo.php';

und dadurch an weitere Informationen kommen.

SQL Injections

Ist es MySQL Vers. >=5.x, kann vieles einfach erfragt werden. Z.B.

→ Name der Datenbank:

```
" union select 1, schema_name
from information_schema.SCHEMATA -- "
```

→ Tabellen der Datenbank:

```
" union select 1,TABLE_NAME FROM
information_schema.COLUMNS where TABLE_SCHEMA
!="information_schema" GROUP by TABLE_NAME -- "
```

→ Tabellenstruktur einer Tabelle:

```
" union select 1,concat(DATA_TYPE," ",COLUMN_NAME) from
information_schema.COLUMNS where TABLE_NAME= ...
```

SQL Injections

Anführungszeichen kann man auch beim Hacken vermeiden: String hexen

- table_schema="Datenbankname„
- table_schema=0x446174656E62616E6B6E616D65

Beschränkte Anzeige der Resultset

- An die Queries ein Limit x,1 anhängen, um die x-te Zeile einer Tabelle zu erhalten. Verschiedene x einfach probieren
- concat() nutzen, um Spalten zu konkatenieren.
- group_concat() nutzen und Zeilen aneinanderhängen.

Boolean-Based SQL-Injection

- Funktionen "ASCII()" und "Substring()" nutzen, um auf ein Zeichen eines kompletten Strings zu testen

SQL Injections

Wie verhindert man solche SQL-Injections?

- **Eine Gefahr ist überall da vorhanden,**
  - wo Benutzerinput entgegengenommen wird
  - Dieser Input nicht auf Validität geprüft wird
  - Input zur Abfrage einer Datenbank verwendet wird
  - String-Konkatenation oder String-Ersetzen zur Konstruktion der Abfrage genutzt wird
- **Alle Datenbankabfragen müssen im Code Review geprüft werden.**
- **Beim Testen vorgehen wie beim Fuzzing**
  - Große Menge an zufällig generierten SQL-ähnlichen Befehlen mit wahllos eingestreuter Interpunktion
- **Tools zum Finden von SQL-Injections**

SQL Injections

Wie kann man SQL-Injection vermeiden?

- Input immer Überprüfen, Eingabe niemals vertrauen
- Am besten: mit regulären Ausdrücken so viel Struktur im Input prüfen, wie möglich
- Eingaben richtig escapen (PHP z.B: Mysql_real_escape_string())
- Insbesondere: Eingaben mit Anführungszeichen verbieten, wenn möglich Niemals String-Konkatenation oder String-Ersetzen zum Aufbau von SQL- Statements verwenden
- Fehler loggen und passende Meldungen an den Hacker.

mysql_Query("SELECT....") or die (echo "Es wurde eine illegale Aktivität festgestellt. Anzeige wird erstattet. Ihre IP-Adresse lautet:"; echo getenv("REMOTE_ADDR");mysql_Query("INSERT INTO ip_log ...."););

- Prepared SQL-Statements verwenden

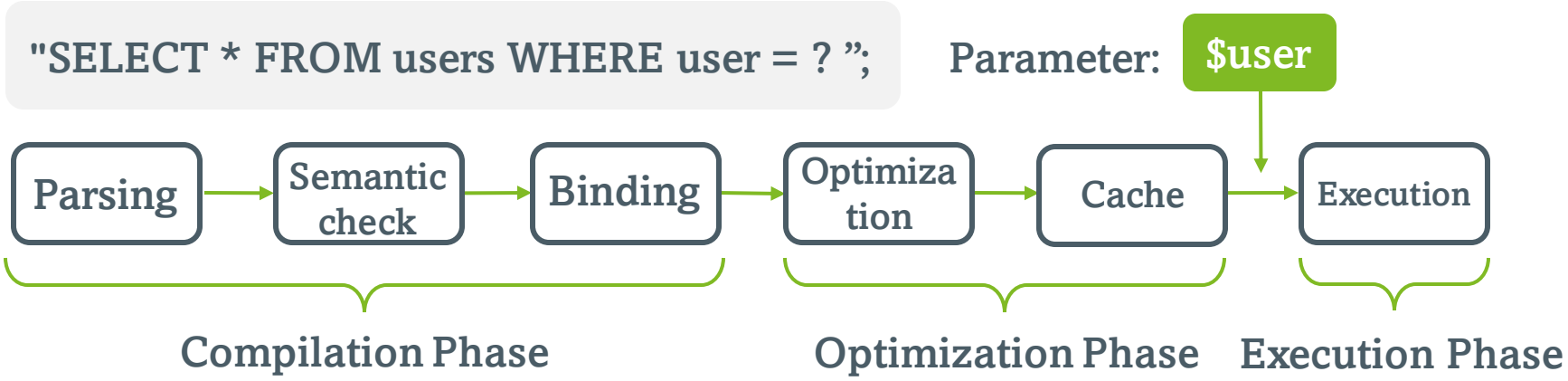
Prepared Statements

Eine Prepared Statement ist eine Funktion, mit der ähnliche SQL-Anweisungen wiederholt mit der gleichen Weise und hoher Effizienz ausgeführt werden

Funktionsweise

- Vorbereiten: Eine SQL-Anweisungsvorlage wird erstellt und an die Datenbank gesendet. Bestimmte Parameter sind mit "?" gekennzeichnet
- Die Datenbank analysiert, kompiliert und führt eine Abfrageoptimierung für die SQL-Anweisungsvorlage durch
- Ausführen: Zu einem späteren Zeitpunkt bindet die Anwendung die Werte an die Parameter und die Datenbank führt die Anweisung aus

Prepared Statements



Prepared Statements

Vorteile

- Geringere Parsingzeit
- Weniger Kommunikation zum Server
- (Eigentlich) keine SQL-Injections möglich

Nachteil

- Etwas höherer Implementierungsaufwand

Prepared Statements

```
// prepare and bind
$stmt = $conn->prepare("INSERT INTO MyGuests (firstname, lastname,
email) VALUES (?, ?, ?)");
$stmt->bind_param("sss", $firstname, $lastname, $email);

// set parameters and execute
$firstname = "Frank";
$lastname = „Hammer";
$email = "frank@hammer.com";
$stmt->execute();
```

Mit $stmt->get_result() holt man sich bei SELECT die Resultset.

Cross-Site-Scripting (XSS)

- **Schadcode in eine vermeintlich Vertrauenswürdige Umgebung einbetten**
- **Angriffsarten**
  - reflektiertes XSS
    - Serverseitige Manipulation für speziellen Nutzer
  - persistentes XSS
    - Serverseitige Manipulation für alle Nutzer
  - lokales XSS
    - Manipulation des Clients

Cross-Site-Scripting (XSS)

- Bei einer Webanwendung wird der Code nur vom Webserver geladen. Ausgeführt wird er lokal auf dem Rechner des Nutzers.

Cross-Site-Scripting (XSS)

- Bei einer Webanwendung wird der Code nur vom Webserver geladen. Ausgeführt wird er lokal auf dem Rechner des Nutzers.

Cross-Site-Scripting (XSS)

HTML

statisch

```
<!DOCTYPE html>
<html lang="de">
<head>
    <title>Document</title>
</head>
<body>
    <div id=“sse”>...</div>
</body>
</html>
```

JavaScript

Dynamisch

- `document.getElementById("sse")`
- **Web APIs**
  - https://developer.mozilla.org/en-US/docs/Web/API?retiredLocale=de
- `localStorage.getItem("sse");`

Angreifer möchten schädlichen JavaScript-Code in den HTML Code einbetten.

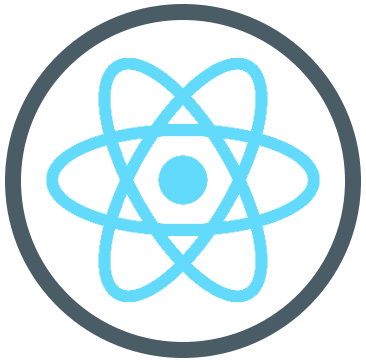
XSS Prävention

- Sicherheit durch Frameworks
- Kodierung der Ausgabe
- HTML Sanitization
- Safe Sinks
- Weitere Möglichkeiten
  - Cookie Attributes
  - Content Security Policy
  - Web Application Firewalls

Sicherheit durch Frameworks

- React, Vue und Angular bieten einen eingebauten Schutz vor XSS Attacken
- z.B. durch Verwendung von Templates und Auto-Escaping
- Über die Framework Protection informieren!

Sicherheit durch Frameworks



- Kein HTML in React
- Eigene Syntaxerweiterung
- dangerouslySetInnerHTML()

Kodierung der Ausgabe

- HTML Contexts
- HTML Attribute Contexts
- JavaScript Contexts
- CSS Contexts
- URL Contexts
- Dangerous Contexts

HTML Contexts

- Einfügen von Variablen zwischen HTML-Tags

```
<div>$variable</div>
```

```
<div><script>...</script></div>
```

- HTML entity encoding

&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&#x27;

```
<script>...</script>
```

```
&lt;script&gt;...&lt;/script&gt;
```

HTML Attribute Contexts

- Einfügen von Variablen in einem HTML-Attributwert

```
<div attr="$variable">
```

```
<div attr="x" onblur="alert(1)">
```

- Da umschließen der Variable kann XSS erschweren aber nicht verhindern
- Bei Setzen durch JavaScript
  - Element.setAttribute()

JavaScript Contexts

- Einfügen von Variablen in Inline-JavaScript

```
<script>x='$variable'</script>
```

- Nur Variablen in Anführungszeichen sind „sicher“

CSS Contexts

- Einfügen von Variablen in Inline-CSS
- Tritt häufig bei Styleanpassung durch den Benutzer auf

```
<span style="property : $variable">Oh no</span>
```

```
<style> selector { property : $varUnsafe; } </style>
```

- Bei Setzen durch JavaScript
  - style.property = x

URL Contexts

- Einfügen von Variablen in einer URL
- Parameter URL encoden

```
url = "https://site.com?data=" + urlencode(parameter)
```

- HTML attribute encoding

```
<a href='attributeEncode(url)'>link</a>
```

- Bei Setzen durch JavaScript
  - window.encodeURIComponent(x)

HTML Sanitization

- Gibt Anwendungsfälle bei denen HTML gerendert werden soll
- Bereinigen von HTML-Code => Entfernen von gefährlichem HTML

- Aufpassen:
  - Inhalte nach dem bereinigen nicht mehr ändern
  - Sicherstellen, dass bereinigtes HTML, das an eine Bibliothek übergeben wird von dieser nicht verändert wird
  - Bei der Verwendung von HTML-Sanitization-Bibliotheken diese regelmäßig updaten

Safe Sinks

- Source
  - Orte die Benutzereingaben in den DOM übernehmen

```
https://example.org/test#section
```

```
windows.location.hash
```

- Sink
  - Orte an denen Benutzereingaben ausgeführt werden

```
element.innerHTML = section
```

```
element.textContent = section
```

Goldene Regel

Traue niemals Benutzereingaben!

Alle Nutzereingaben sollten als „böse“ betrachtet werden, bis diese validiert wurden.