

**Seminar : Sicherheit und technischer  
Datenschutz in Informationssystemen  
2006**

---

Entwurf sicherer Systeme

Ausarbeitung

---

**Paul Mandalka**

*Ausarbeitung zum Vortrag vom 26.06.2006*

*Paul.Mandalka@web.de*

*Betreuung: Jutta Mülle, IPD - Lehrstuhl Prof. Böhm*

*Universität Karlsruhe*

# Inhaltsverzeichnis

1	Einleitung .....	3
2	Entwurfsprinzipien .....	3
	2.1 Prinzip der geringsten Rechte .....	3
	2.2 Prinzip der ausfallsicheren Instatierung .....	4
	2.3 Prinzip der Ökonomie des Sicherheitsmechanismus .....	4
	2.4 Prinzip der vollständigen Vermittlung .....	5
	2.5 Prinzip des offenen Entwurfs .....	5
	2.6 Prinzip der Aufteilung der Rechte .....	5
	2.7 Prinzip der psychologischen Akzeptanz .....	6
	2.8 Prinzip der wenigsten gemeinsamen Funktionen .....	6
	2.9 Zusammenfassung Entwurfsprinzipien .....	8
3	Das Ausschlussproblem .....	8
	3.1 Isolation .....	9
	3.2 Versteckte Kanäle .....	10
	3.3 Zusammenfassung Ausschlussproblem .....	13
4	Entwurf von Systemen mit Zusicherungen .....	13
	4.1 Anforderungsdefinition und Analysephase .....	13
	4.2 System- und Softwareentwurf .....	16
	4.3 Implemenierungsphase .....	18
	4.4 Einsatz und Wartung .....	18
5	Schlusswort .....	19

## 1 Einleitung

Die Sicherheit eines Systems hängt oft von einem durchdachten, sicheren Entwurf ab. Dieser Artikel beschreibt einige Vorgehensweisen, die dem Entwickler helfen sollen, ein sicheres System zu entwerfen. Dabei wird zunächst auf allgemeine Entwurfsprinzipien eingegangen um ein Verständnis für mögliche Sicherheitsmängel in Software zu vermitteln. Weiterhin wird auf das Problem des Ausschlusses eingegangen, wobei es insbesondere um den Ausschluss von unauthorisierten Subjekten von vertraulichen Informationen gehen wird. Dabei wird besonders auf die Isolation von Prozessen eingegangen, um versteckte Kanäle zwischen diesen zu verhindern. Der letzte Teil befasst sich mit den einzelnen Entwurfsphasen eines Entwicklungsprozesses und den Maßnahmen, die in den jeweiligen Phasen vorgenommen werden können, um die Sicherheit des Systems zu erhöhen.

## 2 Entwurfsprinzipien

Dieser Abschnitt gibt einen Überblick über acht Entwurfsprinzipien, die dem Entwickler unterstützen sollen, typische Entwurfsfehler bei Entwurf sicherer Systeme zu vermeiden, diese Prinzipien werden von Saltzer und Schroeder ausführlich beschrieben und in [Bis04] zusammengefasst erläutert. Zunächst ein die Definition von zwei Begriffen, die in den Entwurfsprinzipien verwendet werden.

**Definition 1.** *In einem System ist ein Objekt eine Ressource, die Informationen enthält. Dies kann zum Beispiel eine Datei sein.*

**Definition 2.** *Ein Subjekt ist jemand oder etwas das Zugriff auf Objekte im System erhält und diese verarbeiten möchte. Ein Beispiel für ein Subjekt sind Benutzer des Systems oder Prozesse, die auf dem System laufen.*

### 2.1 Prinzip der geringsten Rechte

**Definition 3.** *Das Prinzip der geringsten Rechte gibt an, das ein Subjekt eines Systems nur die Rechte erhalten sollte, die es zum Ausführen seiner Arbeit auch benötigt und diese auch möglichst schnell wieder abgeben sollte.*

Benötigt ein Subjekt keine Rechte für ein Objekt, so sollte es diese Rechte auch nicht haben. Es sollte auch darauf geachtet werden, welche Art von Recht ein Subjekt auf einem Objekt hält, so braucht ein Subjekt z.B. keine Schreibrechte auf ein Objekt, wenn es nur Daten an das Objekt anhängt. Eine derartige Abstufung der Rechte ist jedoch in den meisten Systemen nicht gegeben. Der Zeitraum zwischen dem erhalten eines Rechts und der Abgabe dieses Rechts sollte ebenfalls möglichst klein gehalten werden.

*Example 1.* Ein Mail-Server der Daten von einem Netzwerkinterface in ein Verzeichnis schreibt, benötigt lediglich Leserechte auf das Netzwerkinterface und

er muss berechtigt sein, im entsprechenden Verzeichnis Dateien anzulegen und diese zu schreiben. Er benötigt jedoch keinen Zugriff auf Benutzerdaten und benötigt auch nachdem er eine Datei geschrieben hat keine weiteren Rechte auf dieser Datei.

## 2.2 Prinzip der ausfallsicheren Instatiiierung

**Definition 4.** *Diese Prinzip besagt, das ein Subjekt beim erzeugen möglichst keine Rechte auf Objekte erhalten sollte und diese erst nach und nach bekommen sollte, falls es sie braucht.*

Der Grund für dieses Prinzip ist die Tatsache, das das Subjekt unvorhersehbare Aktionen durchführen könnte. Gehen wir von einem Prozess als Subjekt aus, so könnte dieser Prozess abstürzen oder fehlerhaft laufen. Auch Angriffe auf das System können einen Prozess beeinflussen. Ist der Prozess nun nicht mehr in der Lage sein Aufgabe zu erfüllen, so sollte er sämtlichen Rechte abgeben und sich beenden. Probleme treten nämlich auf, wenn der Prozess versucht, seine Rechte zu erweitern, wie man im folgenden Beispiel sehen kann

*Example 2.* Ein Mail-Server, der im Spool Verzeichnis keine Dateien mehr erstellen kann, sollte nicht versuchen, diese Dateien in einem anderen Verzeichnis zu speichern, da dieses Verhalten von Angreifern ausgenutzt werden könnte, um das System anzugreifen.

## 2.3 Prinzip der Ökonomie des Sicherheitsmechanismus

**Definition 5.** *Dieses Prinzip besagt, das ein Sicherheitsmechanismus so einfach wie möglich sein soll.*

Sicherheitsmechanismen, die unnötig kompliziert sind machen viele Annahmen über das System. Besteht der Sicherheitsmechanismus aus vielen Komponenten, werden Annahmen sowohl über Eingabeparameter, sowie Rückgabewerte gemacht, so das es vorkommen kann, das Eingabeparameter nicht überprüft werden.

Wird der Sicherheitsmechanismus möglichst klein gehalten, gibt es weniger Komponente und weniger Schnittstellen, was die Gefahr von Fehlfunktionen verringern kann.

*Example 3.* Frühere Clients, die das Finger Protokoll implemenierten sind davon ausgegangen, das die Antwort des angefragten Rechners stehts ein wohlgeformtes Ergebnis liefert. Hier entstand eine Sicherheitslücke, den der angefragte Rechner konnte dem anfragenden Rechner einfach Fehlerhafte Antworten senden, die den Rechner dann zum Absturz bringen können, z.B. mit Endlosantworten.

## 2.4 Prinzip der vollständigen Vermittlung

**Definition 6.** *Diese Prinzip gibt an, das bei jedem Zugriff auf ein Objekt die Berechtigung für diesen Zugriff überprüft werden soll.*

Dabei geht es vor allem um wiederholten Zugriff auf Objekte. Greift ein Subjekt auf ein Objekt mehrfach zu, so sollten die Berechtigungen immer überprüft werden, da sich die Berechtigung seit dem letzten Zugriff bereits geändert haben könnte. Leider befolgen einige Systeme dieses Prinzip nicht, da bei Zugriffsberechtigungen meistens ein Cache dafür sorgt, das die Berechtigungen eben nicht jedes mal neu ausgelesen werden müssen, um eine bessere Performance zu erreichen.

*Example 4.* Bei UNIX Systemen kann man mit Hilfe des System-Aufrufs *fopen* eine Datei öffnen. Dabei überprüft das System beim Aufruf von *fopen* die Zugriffsberechtigung. Falls das Subjekt berechtigt ist, erhält es einen Dateidescriptor, mit dem es auf die Datei zugreifen kann. Bei wiederholtem Zugriff auf die Datei, wird dann nur noch dieser Dateidescriptor benötigt und somit werden die Berechtigungen nicht nochmals überprüft.

## 2.5 Prinzip des offenen Entwurfs

**Definition 7.** *Das Prinzip besagt, das die Sicherheit eines Systems nicht davon abhängen soll, das die Implementierung oder der Entwurf des Systems nicht bekannt sind.*

Dies ist vor allem ein wichtiger Punkt, wenn es um Verschlüsselungsalgorithmen geht. Wenn die Sicherheit einer Verschlüsselung nur von der Geheimhaltung des Algorithmus abhängt, wird dieser früher oder später mit Hilfe von *Reverse Engineering* möglicherweise herausgefunden und somit ist der Verschlüsselungsalgorithmus nicht mehr sicher. Das Geheimhalten von Schlüssel ist kein Verstoß gegen dieses Prinzip.

*Example 5.* Ende der 90er Jahr wurde das *Content Scrambling System* entwickelt und als Kopierschutz für DVDs benutzt (siehe [CSS04] und [WikiDE] ). Die Entwickler haben dabei dieses Prinzip gänzlich missachtet und den Algorithmus zum Erstellen ihrer geheimen Schlüssel geheimgehalten, um das System abzusichern. Bereits nach kurzer Zeit wurde dieser geheime Algorithmus mit Hilfe eines Schlüssels und *Reverse Engineering* nach gebaut, was diesen Kopierschutz komplett unwirksam machte.

## 2.6 Prinzip der Aufteilung der Rechte

**Definition 8.** *Dieses Prinzip drückt aus, das die Bewilligung einer Berechtigung nicht auf einer einzigen Bedingung basieren sollte.*

Vor allem systemkritische Rechte, wie z.B. Administratoren Rechte, sollten möglichst mehrfach abgesichert werden, damit nicht bereits mit Hilfe einer Manipulation die Sicherheit des Systems gefährdet wäre.

*Example 6.* UNIX-Benutzer können *Administratoren Rechte* nur erhalten, wenn sie das Administratoren Passwort kennen und zusätzlich in der Administratoren Benutzergruppe sind. Somit muss der Benutzer zwei Bedingungen erfüllen, um ein Recht zu erhalten.

## 2.7 Prinzip der psychologischen Akzeptanz

**Definition 9.** *Dieses Prinzip gibt an, das das System mit der Sicherheitsfunktion immer noch für den Benutzer möglichst einfach zu benutzen sein soll.*

Dies ist ein sehr wichtiges Prinzip, da es den Benutzer direkt einbezieht. Wenn ein Benutzer das Sicherheitssystem für unangemessen betrachtet, wird er wahrscheinlich das Sicherheitssystem womöglich abschalten, d.h. man muss darauf achten, das die Benutzung des Systems für den Benutzer trotzdem möglichst einfach bleibt. Hierbei kann es auch nützlich sein, das der Benutzer bestimmte Einstellungen des Systems speichern kann, damit er nicht jedes mal danach gefragt wird. Dies kann vor allem bei unkritischen Daten wie z.B. *public keys* erfolgen, die allgemein verfügbar sind.

*Example 7.* Bei *ssh* Verbindungen kann man die *public keys* eines Servers speichern, so der der Benutzer beim nächsten mal zwar weiterhin eine sichere Verbindung zum Zielrechner aufbauen kann, aber er nicht durch wiederholtes Nachfragen belästigt wird.

Weiterhin sollten die Sicherheitseinstellungen, die man am System vornehmen kann, möglichst einfach zu konfigurieren sein, denn sind diese zu kompliziert, kann es durchaus zu Einstellungen kommen, die so nicht gewollt waren. Selbst erfahrene Benutzer könnten hier aufgrund der großen Anzahl an Einstellungsmöglichkeiten einen ungewollten Fehler machen.

Ein letzter Teil dieses Prinzips ist, das der Benutzer zwar sprechende Fehlermeldungen erhalten soll, jedoch diese nicht zu viel Informationen über des System preisgeben sollten. Das heißt vor allem, das Benutzer keine Auskunft über Informationen erhalten sollen, die vor ihnen verborgen werden sollten.

*Example 8.* Bei einer Anmeldung am System ist eine Fehlermeldung wie z.B. "Passwort falsche" nicht wünschenswert, da dies dem möglichen Angreifer eventuell verraten würde, das der angegebene Benutzername existiert und nur das Passwort falsch war. Eine bessere Lösung wäre hier, wie es bei UNIX-Systemen der Fall ist, einfach eine Ausgabe der Art : "Anmeldung fehlgeschlagen".

## 2.8 Prinzip der wenigsten gemeinsamen Funktionen

**Definition 10.** *Dieses Prinzip besagt, das Mechanismen für den Zugriff auf Ressourcen möglichst nicht geteilt werden sollten.*

Auf den ersten Blick scheint dieses Prinzip etwas seltsam, da es gerade dem Wiederverwenden von Code, bzw. das gemeinsame Verwenden von Komponenten entgegensteht. Das Problem jedoch beim Verwenden von gemeinsamen Ressourcen ist jedoch, das über diese Ressource Daten zwischen zwei Subjekten

ausgetauscht werden können, die unter Umständen nicht dazu berechtigt sind, Daten auszutauschen. Dabei werden so genannte *versteckte Kanäle* zwischen den Subjekten mit Hilfe der Attribute einer gemeinsamen Ressource erstellt. Diese könnten zur Kommunikation verwendet werden. Das *Ausschlussproblem* beschäftigt sich mit diesem Problem.

*Example 9.* Nehmen wir an, zwei Prozesse *High* und *Low* können keine Daten direkt austauschen und der Prozess *High* möchte Daten an *Low* senden.

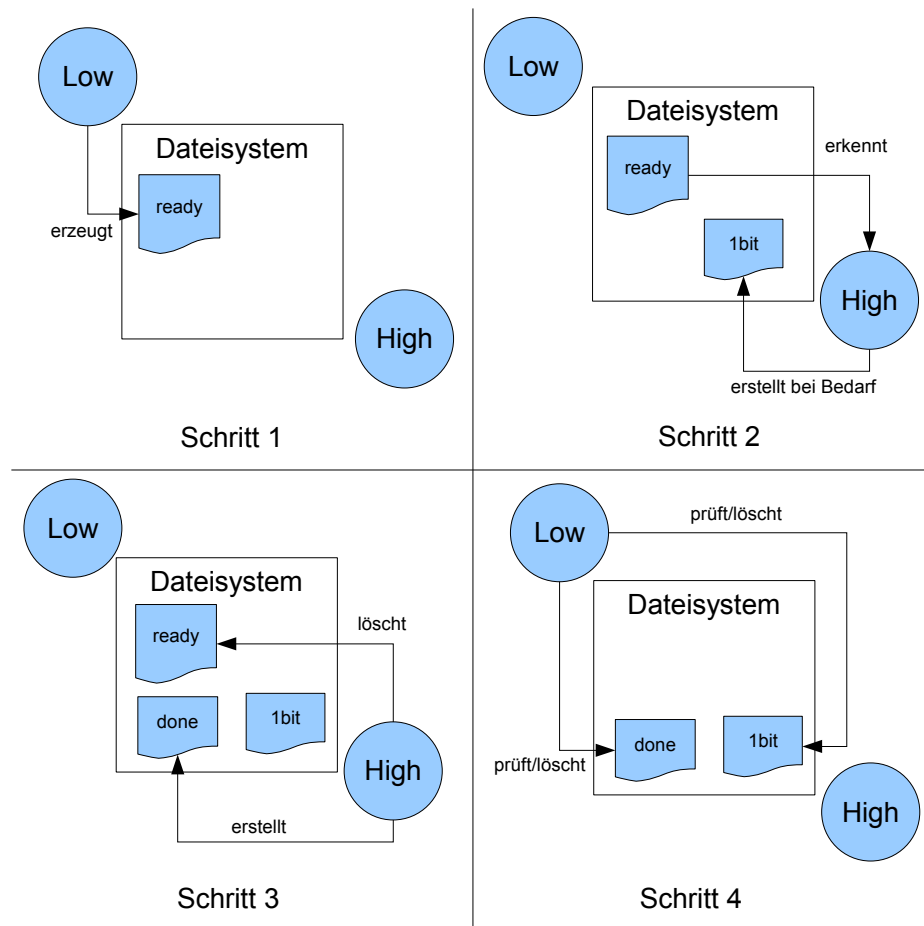


Abbildung 1. Versteckter Kanal über Dateisystem

Mit Hilfe des Dateisystems ist ein Datenaustausch möglich, das folgende Vorgehen illustriert dies. Zu Beginn legt der *Low* Prozess eine Datei namens *ready* an, dies zeigt dem *High* Prozess, dass der *Low* Prozess bereit ist Daten zu empfangen. Nun entscheidet der *High*, Prozess, ob er eine Datei *1bit* anlegt oder nicht,

je nachdem, ob er eine 1 oder eine 0 senden möchte. Danach löscht der *High* Prozess die *ready* Datei und erstellt eine *done* Datei. In periodischen Abständen kontrolliert Prozess *Low*, ob die Datei *done* existiert, findet er diese vor, kann er nun nachprüfen ob die Datei *1bit* existiert oder nicht und daraus schließen, was der Prozess *High* ihm gesendet hat. Danach löscht er die entsprechenden Dateien und erstellt wieder die *ready* Datei und der Vorgang beginnt vom neuen. Abbildung 1 illustriert dieses Verfahren. Dabei ist zu beachten, das das Attribut *file existence* einer Datei mit Hilfe von *delete file* und *create file* ausgelesen werden kann, indem das Ergebnis der jeweiligen Operation ausgewertet wird.

## 2.9 Zusammenfassung Entwurfsprinzipien

Die aufgezählten Entwurfsprinzipien sollen dem Entwickler helfen, typische Fehler zu vermeiden. Das einhalten dieser Prinzipien ist jedoch nicht Hinreichend dafür, das ein System sicher ist. Einer der wichtigsten Entwurfsprinzipien ist hier meiner Meinung nach das *Prinzip der psychologischen Akzeptanz*, da dieses Prinzip den Benutzer in den Entwurf mit einbringt und man sich somit beim Entwerfen eines Systems bewusst werden muss, das dieses System auch möglichst einfach benutzbar sein soll. Diese Entwurfsprinzipien sollen den Entwickler unterstützen Sicherheitsprobleme zu erkennen und müssen, wie man an den Beispielen teilweise gesehen hat, nicht komplett übernommen werden, um ein sicheres System zu entwerfen, den nicht jedes System muss höchsten Sicherheitsanforderungen entsprechen.

## 3 Das Ausschlussproblem

Definiert ein Benutzer bestimmte Objekte eines Systems als vertraulich, so sollen diese Objekte nur von Subjekten benutzt werden können, die entsprechende Berechtigungen haben. Um die Vertraulichkeit der Daten zu garantieren, müssen die Subjekte überwacht werden, damit sie vertrauliche Daten nicht an unberechtigte Subjekte weitergeben können. Dieses Problem des Ausschlusses unberechtigter Subjekte von vertraulichen Daten wird *Ausschlussproblem* genannt (vgl. [Bis04]).

Leider kann man in den meisten Systemen den Datenfluss nicht komplett kontrollieren. So können berechtigte Subjekte auf ein vertrauliches Objekt zugreifen und dieses Daten dann an ein unberechtigtes Subjekt weitergeben, das folgende Beispiel illustriert dies.

*Example 10.* Darf Subjekt A die Datei *geheim.txt* lesen, da es berechtigt ist, könnte es diese Informationen in einer andern Datei *offen.txt* ablegen. Ein weiteres Subjekt B, das keine Berechtigung für die Datei *geheim.txt* hat, können jedoch Zugang zur Datei *offen.txt* haben und somit an die eigentlichen Daten aus der Datei *geheim.txt* kommen.

Hier könnte eine *Low Watermark Policy* verhindern, das Prozess A Daten an Prozess B übergibt, allerdings ist diese Policy meistens nicht praktikabel, da

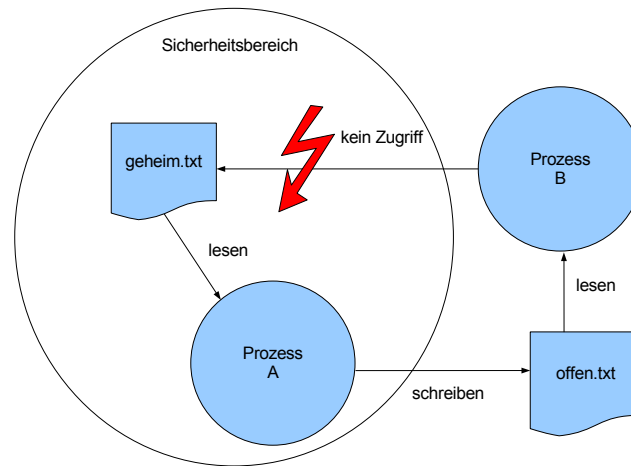


Abbildung 2. Illustration Example 10

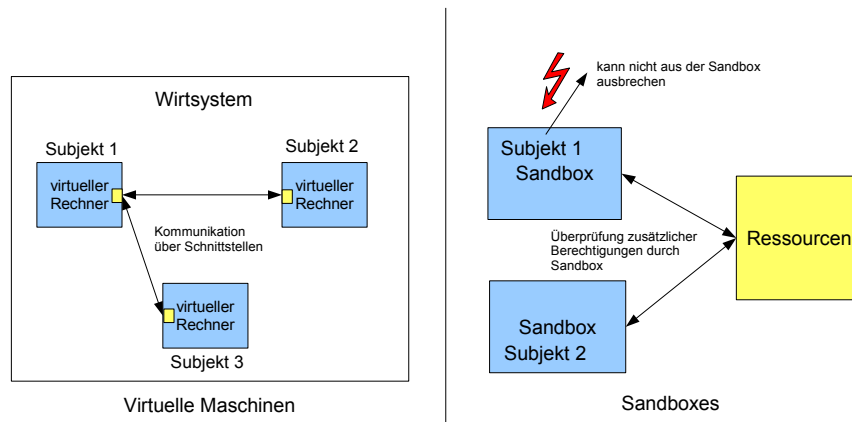
die Berechtigung des Benutzer mit der Zeit immer kleiner wird. Es gibt jedoch einen Ansatz, um Subjekte daran zu hindern, Daten an unberechtigte Subjekte weiterzugeben. Dazu gehört die vollständige *Isolation* der Subjekte im System.

### 3.1 Isolation

Dem Ausschlussproblem kann man mit Hilfe von Isolation der einzelnen Subjekte entgegen wirken. Hier werden die Subjekte komplett voneinander getrennt, so das beide von ihrer Existenz nichts wissen. Dabei gibt es zwei unterschiedliche Ansätze, die *virtuellen Maschinen* und die *Sandboxes*. Beide Verfahren versuchen die Kommunikation zwischen Subjekten zu überprüfen um die Weitergabe von vertraulichen Dokumenten an unberechtigte Subjekte zu verhindern.

**Virtuelle Maschine** Hierbei hat jedes Subjekt auf dem System einen eigenen virtuellen Rechner, auf dem das Subjekt sämtliche Tätigkeiten ausführen kann. Wollen nun zwei Subjekte Daten austauschen, müssen sie dies über entsprechende Schnittstellen der virtuellen Maschinen machen. Somit kann die Berechtigungen überprüft werden, da nur über diese Schnittstellen Daten ausgetauscht werden können. Virtuelle Maschinen erlauben es auch zusätzliche Sicherheitseinstellungen zu unterstützen, die das Wirtssystem an sich nicht unterstützen würde.

*Example 11.* Die Java Virtual Machine kann z.B. überprüfen, ob das ausgeführte Programm ein Applet ist um dann bestimmte Zugriffe auf den Rechner für Applets zu verbieten.



**Abbildung 3.** Virtuelle Maschinen vs. Sandboxes

Das Problem an virtuellen Maschinen ist jedoch, dass es immer noch zu einem Informationsfluss kommen kann, indem die Subjekte so genannte versteckte Kanäle benutzen könnten, um Daten auszutauschen, da sie gemeinsam auf einem Rechner laufen und somit gemeinsame Ressourcen des Rechners nutzen.

**Sandboxes** Bei Sandboxes wird das System an sich verändert und um zusätzliche Prüfungen von Berechtigungen erweitert. Dies kann auf Bibliotheks Ebene, z.B. im Kernel, passieren oder als zusätzlicher Echtzeit Debugger, der bei Systemaufrufen das Programm anhält und die Berechtigungen überprüft.

Nachteil an Sandboxes ist jedoch, dass sie die Performance des Systems beeinträchtigen und ähnlich wie bei den virtuellen Maschinen eine Übertragung der Daten über versteckte Kanäle nicht verhindern können.

**Zusammenfassung** Die beiden Ansätze verhindern zwar einen direkten Datenaustausch zwischen Subjekten, können jedoch versteckte Kanäle nicht ganz verhindern. Außerdem können Fehler in den Implementierungen beider Ansätze eine falsche Sicherheit vermitteln.

### 3.2 Versteckte Kanäle

**Definition 11.** *Versteckte Kanäle sind Kommunikationspfade, die nicht zur Kommunikation entwickelt wurden.*

Man unterscheidet dabei zwei Arten von versteckten Kanälen. Die *versteckten Speicher Kanäle* benutzen Attribute von gemeinsamen Ressourcen, um Daten auszutauschen. Diese sollen mit dem oben erwähnten Entwurfsprinzip *Prinzip der wenigsten gemeinsamen Funktionen* eben verhindert werden.

Die zweite Art von versteckten Kanälen sind Kanäle durch *zeitliche Abstimmung*, hierbei wird das Laufzeitverhalten von Komponenten oder Programmen mit Hilfe von Zeitgebern, wie der Echtzeituhr, analysiert.

Ein Beispiel für versteckte *Speicher Kanäle* wurde bereits im Abschnitt zum Entwurfsprinzip *Prinzip der wenigsten gemeinsamen Funktionen* gezeigt.

Die Zeit kann als verteckter Kanal benutzt werden, dabei wird die Laufzeit eines Programms oder auch einer Komponente untersucht. Anhand dieser Analyse kann man dann Rückschlüsse auf die Engabeparameter ziehen.

*Example 12.* Folgendes Programm enthält eine Schwachstelle, mit deren Hilfe man Rückschlüsse auf die Eingabedaten ziehen kann. Das Programm berechnet  $x = a^z$ , diese Berechnung wird z.B. bei Berechnung von Schlüsseln in der Verschlüsselung verwendet.

```
x := 1; atemp := a;
  for i := 0 to k-1 do begin
    if  $z_i = 1$  then
      x := (x * atemp) mod n;
      atmp := (atmp * atmp) mod n;
    end;
  result = x;
```

Die Schwachstelle des Programms liegt in der if-Anweisung. Ist  $z_i = 1$ , dann wird eine Multiplikation  $x := (x * atemp) \bmod n$ ; ausgeführt, ansonsten wird dieser Teil übersprungen. Wird das Laufzeitverhalten des Programms nun analysiert, so lassen sich recht einfach Laufzeitunterschiede feststellen, da bei  $z_i = 1$  zwei Multiplikationen erfolgen und bei  $z_i = 0$  nur eine, wobei man somit auf die Zahl  $z$  schließen könnte, die womöglich geheim gehalten werden sollte.

Aber nicht nur die Laufzeit von Programmen kann analysiert werden, man auch die zeit einer Komponente bewusst so verändern, das man Informationen austauschen kann.

*Example 13.* Weiß man, das eine Festplatte die SCAN Strategie verfolgt, kann man dies ausnutzen, um mit Hilfe der Zugriffsdauer Daten auszutauschen. Je nach dem, ob das Subjekt eine 0 oder eine 1 senden will, werden bestimmte Sektoren auf der Platte ausgelesen, so das sich wegen der Strategie der Festplatte immer entsprechende Laufzeiten ergeben. Dieser Unterschied in den Laufzeiten lässt dann darauf schließen, ob eine 0 oder eine 1 gesendet wurde.

**Auffinden von versteckten Kanälen** Um versteckte Kanäle zu finden gibt es mehrere Methoden. Mit der Hilfe von formalen Methoden kann man versteckte Kanäle finden, was jedoch sehr umfangreich sein kann, und deswegen in der Praxis nicht oft zum Einsatz kommt. Die *Shared Resource Matrix* (SRM) Methode ist recht einfach und wird im folgendem vorgestellt. Weitere Methoden sind *Informationsflussanalysen* und die *versteckten Fluss Bäume*, die der SRM Methode ähnelt. Die SRM und die *versteckten Fluss Bäume* bauen beide auf der

Idee auf, die gemeinsamen Ressourcen und ihre Operationen zu untersuchen, beide können zu beliebigen Zeitpunkten der Entwicklung eingesetzt werden. Der Vorteil der *versteckte Fluss Bäume* Methode im Gegensatz zu SRM und der Informationsflussanalyse ist, das diese ermittelt, welche Reihenfolge von Operationen zu einem versteckten Kanal führt.

*Shared Resource Matrix Methode* Die Shared Resource Matrix Methode eignet sich ganz gut, um zu überprüfen, ob sich versteckte Kanäle überhaupt finden lassen. Dazu werden die Attribute einer Ressource als Zeilen und die Operationen auf der Ressource als Spalten in einer Tabelle aufgeschrieben. Dabei unterscheidet man, ob eine Operation ein Attribut Lesen oder Schreiben (Ändern) kann. Befinden sich dann in der Tabelle Attribute die von einer oder mehreren Methoden sowohl geschrieben als auch gelesen werden können, kann man davon ausgehen, das sich hier womöglich ein versteckter Kanal befindet.

In der folgenden Tabelle sieht man, das die Operationen `delete_file` und `create_file` das Attribut `file existence` sowohl schreiben als auch lesen können. Somit gibt es hier einen versteckten Kanal, der weiter untersucht werden kann.

	read_file	write_file	delete_file	create_file
file existence	R	R	<b>R,M</b>	<b>R,M</b>
file owner		R	M	
file label	R	R	R	M
file size	R	M	M	M

**Tabelle 1.** Beispiel für Shared Resource Matrix bei einem Dateisystem

**Analyse von verteckten Kanälen** Hat man einen versteckten Kanal erst einmal gefunden, muss untersucht werden in wie weit dieser für eine Datenübertragung nützlich ist. Dazu kann misst man die Kapazität des Kanals und kann dann abwägen, ob dieser eine Gefahr für das System darstellen könnte. Dabei muss man auch beachten, wie schnell die Kommunikation stattfinden kann. Liegt die Rate bei einem Bit pro Stunden, ist dieser Kanal eher harmlos, ist die Rate jedoch sehr hoch z.B. 1.000.000 Bits pro Sekunde, könnte dieser Kanal durchaus gefährlich sein.

**Zusammenfassung** Versteckte Kanäle kann man leider nur sehr schwer beseitigen, vor allem wenn man auf gemeinsame Ressourcen zugreift. Zum Auffinden von versteckten Kanälen gibt es mehrere Verfahren, von denen hier die Shared Resource Matrix Methode vorgestellt wurde. Da sich versteckte Kanäle oft nicht verhindern lassen, ist es eine gängige Methode, die versteckten Kanäle mit zufälligen Werten absichtlich zu verrauschen, um die Kapazität des Kanals zu verringern bzw. den Informationsaustausch komplett zum Erliegen zu bringen.

### 3.3 Zusammenfassung Ausschlussproblem

Das Ausschlussproblem ist gerade bei Systemen mit hohen Sicherheitsanforderungen problematisch. Hier gibt es verschiedene Möglichkeiten versteckte Kanäle, z.B. durch verrauschen des Kanals, zu beseitigen. Andere Methoden wie die Isolation der einzelnen Subjekte durch virtuelle Maschinen oder Sandboxes bieten ebenfalls eine Möglichkeit Daten vertraulich zu halten und können dort eingesetzt werden, wo das Wirtssystem keine entsprechenden Sicherheitsfunktionen bietet. In den meisten gängigen Systemen ist es jedoch nicht möglich zu garantieren, dass vertrauliche Daten nicht von unberechtigten Subjekten auf Umwegen gelesen werden können (vgl. [Bis04]).

## 4 Entwurf von Systemen mit Zusicherungen

Dieser Teil des Dokuments befasst sich mit den einzelnen Entwicklungsphasen in einem Software Projekt. Dabei wird gezeigt was in den einzelnen Phasen beachtet werden sollte, um ein sicheres System zu entwickeln. Es werden außerdem einige Methode zur Validierung der einzelnen Phasen vorgestellt, die die Sicherheit in einem System erhöhen können (vgl. [Bis04]).

### 4.1 Anforderungsdefinition und Analysephase

Bei der Anforderungsdefinition und Analysephase wird zunächst einmal untersucht, welche Anforderungen an das zu entwickelnde System gestellt werden. Dabei muss man im Bezug auf die Sicherheitsfunktionen zwei Grundlegende Fragen klären. Als erstes muss überlegt werden, in welchen Teil der Software man die Sicherheitsfunktionen einbauen möchte. Die meisten Systeme sind in Schichten unterteilt, so dass man zunächst klären muss, in welche Schicht die Sicherheitsfunktionen eingefügt werden sollen. Es kann durchaus sinnvoll sein, die Sicherheitsfunktion in die Anwendungsschicht einzubauen. Dabei geht es meistens um Sicherheitsfunktionen, die die Anwendung selbst betreffen, wie zum Beispiel ein Login Mechanismus an der Anwendung. Andere Sicherheitsfunktionen kann man vom darunter liegenden Schichten übernehmen, so wie zum Beispiel die Sicherheitsfunktionen des Betriebssystems für den Dateizugriff.

Weiterhin muss geklärt werden, wann die Sicherheitsfunktionen in die Anwendung aufgenommen werden sollen. Wird eine Anwendung von Grund auf neu geschrieben, ist es durchaus sinnvoll die Sicherheitsfunktionen direkt beim Entwurf mit einzuplanen. Dies erfordert zwar unter Umständen einen Mehraufwand bei der Planung, kann die Sicherheit der Anwendung durchaus erhöhen, da Entwurfsentscheidungen entsprechend der Sicherheitsanforderungen getroffen werden.

Problematisch ist hier jedoch, wenn Sicherheitsanforderungen erst im Laufe des Projekts hinzukommen oder erweitert werden.

Eine Alternative dazu bietet das Einfügen der Sicherheitsfunktionen im Nachhinein, d.h. die Anwendung wird zunächst ohne Sicherheitsfunktionen implementiert und am Ende der Implementierung, wenn sämtliche Funktionen bereits fertig sind, werden Sicherheitsfunktionen eingefügt.

Bei Anwendungen, die bereits implementiert sind, kann dieses Vorgehen durchaus Zeit sparen, da hier nicht die gesamte Anwendung neu implementiert werden muss. Jedoch kann es sein, dass Entwurfsentscheidungen, die bereits getroffen wurden, die Sicherheit des Systems beeinflussen können.

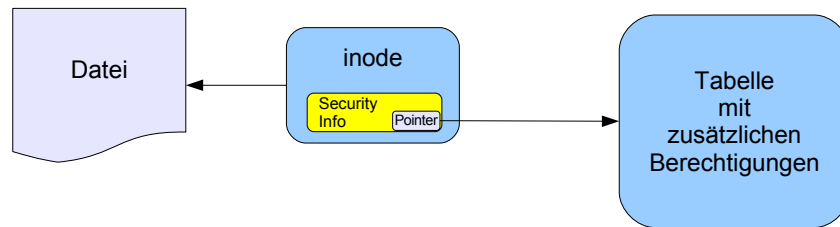


Abbildung 4. 1. Ansatz AT&T

*Example 14.* Ende 1980 Anfang 1990 wollte die Firma AT&T ihr Unix V System um einige Sicherheitsfunktionen im Bereich des Dateisystems erweitern. Es wurden zwei Ansätze verfolgt.

Beim ersten Ansatz hatte man sich zum Ziel gesetzt, möglichst viele Teile des alten Systems wiederzuverwenden. Dazu wurden die alten Datenstrukturen angeschaut. Die Entwickler fanden in der *inode* Struktur des Dateisystems eine freie, unbenutzte Speicherstelle, die sie dazu verwendeten um auf eine Systemtabelle zu verweisen, die die zusätzlichen Berechtigungen enthielt (Abb. 4). Leider hat dieser Entwurf einige Schwachstellen und wurde deswegen fallengelassen. Eine dieser Schwachstellen, ist die schwache Bindung zwischen der Datei und der zusätzlichen Tabelle im Gegensatz zur *inode* und der Datei, denn wird die Tabelle aus dem System entfernt, kann man trotzdem mit Hilfe der *inode* auf die Datei zugreifen, im Gegensatz dazu ist ein Zugriff auf die Datei ohne *inode* nicht ohne weiteres möglich. Weitere Schwächen sind Probleme mit der Konsistenz der Sicherheitseinstellungen, da Änderungen an der *inode* auch Änderungen auf der Tabelle nach sich ziehen und somit eine Inkonsistenz entstehen kann, des Weiteren ist durch eine fehlerhafte Tabelle die Sicherheit des gesamten Dateisystems beeinflusst.

Der zweite Ansatz war eine kompletter Neuentwurf der *inode* Struktur. Diese wurde einfach direkt um die gewünschten Sicherheitsfunktionen erweitert. Somit hatte man die Nachteile des ersten Ansatzes eliminiert, allerdings wird durch eine komplett neue Implementierung der *inode* Struktur die Kompatibilität zu älteren Systemen aufgebrochen. Daran sieht man, dass gerade beim späteren Einfügen von Sicherheitsfunktionen durchaus Probleme entstehen können.

**Entwurf von Richtlinien** Das Entwerfen von korrekten und vollständigen Richtlinien für eine Anwendung kann sehr kompliziert werden. Deswegen gibt es einige Strategien, die es erleichtern sollen, Richtlinien zu erstellen. Zunächst gibt es die Möglichkeit die Richtlinien aus ähnlichen Projekten, die man bereits durchgeführt hat, zu übernehmen. Dies macht vor allem Sinn, wenn die Anwendungen sehr ähnliche Anforderungen haben. Eine weitere Möglichkeit wäre das Durchführen einer Bedrohungsanalyse. Mit Hilfe dieser Analyse ermittelt man die Bedrohungen, die man für die Anwendung erwartet. Nach dieser Analyse kann man wieder rum aus anderen Projekten die Richtlinien übernehmen und an die Bedrohungen anpassen. Ein sehr verbreiteter Weg ist es, die Anwendung auf ein formales Modell abzubilden. Dazu werden die einzelnen Komponente und Funktionen auf das formale Modell abgebildet. Ist der Nachweis für das formale Modell erbracht, das es sicher ist, so kann man davon ausgehen, das die eigene Anwendung, die man auf das Modell abgebildet hat, ebenfalls sicher ist. Ein bekanntes Modell, ist z.B. das Bell-LaPadula Modell.

**Verifikation** Am Ende jeder Entwicklungsphase sollte das entstandene Dokument überprüft werden, ob es mit der Anforderungen übereinstimmt. Für diese Phase der Entwicklung eignet sich das *Information Technology Security Evaluation Criteria (ITSEC)*. Hierbei werden die Bedrohungen, Anforderungen und Annahmen über das System in Tabellarischer Form gegenüber gestellt, so das man genau sehen kann, welchen Bedrohungen welche Anforderungen und Annahmen gegenüberstehen. Auf diese Weise kann Überprüfen, ob alle Bedrohungen abgewendet werden. ITSEC ist ein europäischer Standard für Sicherheit und definiert mehrere Stufen an Sicherheit und welche Vorkehrungen für die einzelnen Stufen erforderlich sind.

*Example 15.* Angenommen für ein Projekt wurde eine Bedrohungsanalyse durchgeführt und es wurden drei Bedrohungen gefunden und formal Beschrieben. Für eine kompakte Darstellung werden die Bedrohungen (Threats) (T1, T2, T3), sowie die Anforderungen (A1 - A4) und die Annahmen über das System (IA1, IA2) durchnummeriert. Nun kann eine Tabelle angelegt werden, die Anzeigt welche Maßnahmen getroffen werden, um eine Bedrohung entgegen zu wirken.

Threat	Security Target	Reference
T1	IA1, IA2, A2, A3, A4	
T2	IA2, A1	
T3		

**Tabelle 2.** Beispiel einer ITSEC-Tabelle

Diese Tabelle zeigt, dass die Bedrohungen T1 und T2 bereits durch die Anforderungen an das System abgedeckt wurden. T3 wird jedoch nicht abgedeckt und somit erfordert dies eine Überarbeitung der Anforderungsdefinition. Anhand

der Tabelle kann man auch sehen, auf welche Bedrohungen sich Änderungen an der Anforderungsdefinition auswirken können.

## 4.2 System- und Softwareentwurf

Die Sicherheit eines Systems hängt von vielen Faktoren ab, einer davon ist die Architektur des Softwareentwurfs. Es hat sich gezeigt, dass verstärkte Modularisierung und die Schichtenarchitektur die Sicherheit eines Systems erhöhen können. Wird die Anwendung in Module eingeteilt, werden diese kompakter, was sie für die Entwickler verständlicher macht. Des Weiteren sind solche kleineren Module einfacher zu warten und ermöglichen ein besseres *Data Hiding*, da die Module die Daten über bestimmte Schnittstellen austauschen. Somit kann eingeschränkt werden, welche Daten ein Modul an ein anderes weitergibt. Außerdem fördert die Modularisierung das Entwurfsprinzip der *geringsten Rechte*, da die einzelnen Module weniger Rechte benötigen und diese auch schneller abgeben können, im Vergleich zu einem großen Modul, das evtl. gleichzeitig viele Rechte halten muss.

**Entwurfsdokumentation** Die Entwurfsdokumentation sollte das Ergebnis dieser Phase sein. Hier sollten die Sicherheitsfunktionen dokumentiert werden, wobei nicht nur die Beschreibung der einzelnen Funktionen, sondern auch die Zusammenarbeit der einzelnen Funktionen dokumentiert werden sollte. Außerdem ist eine Abbildung der Sicherheitsfunktionen auf die Sicherheitsanforderungen nötig, um festzustellen, ob sämtliche Sicherheitsanforderungen durch die Funktionen auch abgedeckt werden.

Ein weiterer Teil der Dokumentation ist die Beschreibung der Funktionen. Die Funktionsbeschreibung sollte aus der Beschreibung der Eingabeparameter sowie Rückgabewerte bestehen, außerdem sollten weiterhin die möglichen Ausnahmen die durch die Funktion verursacht werden können beschrieben werden. Natürlich sollte auch der Effekt der Schnittstelle beschrieben werden.

Nächster Teil der Entwurfsdokumentation ist die *interne Entwurfsbeschreibung*. Hier gibt man eine Übersicht über die Elternkomponenten der einzelnen Komponenten sowie eine detaillierte Beschreibung der Komponenten selbst. Des Weiteren wird hier die Sicherheitsrelevanz der Komponente beschrieben.

Der letzte Teil der Entwurfsdokumentation ist die *interne Entwurfsspezifikation*. Sie ist informeller als die anderen Teile der Entwurfsdokumentation und wird meistens für die Low-Level Dokumentation der Funktionen verwendet.

*Example 16.* Hier ein Beispiel für eine Funktionsbeschreibung:

### Interface Name

```
error_t add_logevent (
handle_t handle,
data_t event
);
```

**Input Parameter**

handle - a valid handle returned from a previous call to open\_log  
 event - the buffer of event data with event records in logevent format

**Exceptions**

Caller does not have permission to add to EVENT file.  
 There is inadequate memory to add to an EVENT file.

**Effects**

Event is added to EVENT log.

**Output Parameters**

status :  
 status\_ok , routine completed successfully  
 no\_memory , routine failed due to insufficient memory  
 permission\_denied , routine failed, caller does not have permission

**Note**

add\_logevent is a user-visible interface.

**Verifikation** Die die Entwurfsdokumentation meistens informell ist, ist eine formale Überprüfung nicht möglich. Allerdings haben sich die Reviews als nützlich erwiesen. Ein Review dient der Überprüfung des Dokuments, ob es die Spezifikationen erfüllt, der Ablauf ist im folgenden Beschrieben. Zunächst übergibt der Autor des Dokuments dieses an den Moderator des Reviews. Ist dieser mit den Status des Dokuments zufrieden, legt er fest, wie lange das Review dauern soll und versorgt die Reviewer mit Materialien für den Review. Diese Materialien enthalten Grundregeln und die Spezifikation auf die das Dokument untersucht werden soll. Daraufhin starten die Reviewer das technische Review, d.h. sie analysieren das Dokument anhand der Grundregeln und der Spezifikation und erstellen Kommentare zu dem Dokument. Dieser Kommentare der einzelnen Reviewer werden dann im *Review Meeting* gesammelt. Dabei gibt jeder Reviewer seine Kommentare möglichst prägnant wieder. Wichtig ist, das der Moderator darauf achtet, das die Kommentare der einzelnen Reviewer nicht erniedrigend oder persönlich gegenüber dem Autor sind, außerdem muss er darauf achten, das hier lediglich die Kommentare gesammelt werden und keine Lösungen diskutiert werden. Nach dem Review Meeting kommt es zur Konfliktlösungsphase. Da es sein kann, da die Kommentare der einzelnen Reviewer sich widersprechen, muss den Reviewern eine Möglichkeit geboten werden, die Kommentare zu diskutieren und über die Kommentare abzustimmen. Bestehen keine Konflikte zwischen den Kommentaren, so werden sie dem Autor übergeben, der nun das Dokument gemäß der Kommentare versucht abzuändern. Nach den Änderungen am Dokument überprüfen die Reviewer und der Autor ob sämtliche Änderungen am Dokument auch den Kommentaren entsprechen. Sind sämtliche Kommentare abgearbeitet und alle Konflikte gelöst ist das Review beendet.

Reviews können durchaus auch auf andere Arten von Dokumenten angewendet werden. So kann man auch den Quellcode einer Anwendung als Dokument ansehen und einen Review für diesen durchführen.

### 4.3 Implementierungsphase

Auch in der Implementierungsphase müssen Entscheidungen getroffen werden, die die Sicherheit eines Systems beeinflussen können. So kann die Wahl der Programmiersprache die Sicherheit beeinflussen. Systeme die mit low level Sprachen geschrieben sind, neigen eher dazu, Sicherheitslücken zu enthalten. Dieses Phänomen liegt daran, dass die low level Sprachen wie z.B. C keine Typenprüfung anbieten, außerdem werden hier oft Grenzen von Datenstrukturen nicht überprüft, so dass es zu Überläufen kommen kann. Es gibt jedoch Fälle, bei denen man gezwungen ist low level Sprachen einzusetzen wie z.B. beim Ansprechen von Hardware. Wichtig hierbei ist, dass die Entwickler sich an bestimmte Coding Standards halten, damit der Code einfacher zu überprüfen ist und auch allgemeine Fehler vermeiden kann.

In der Implementierungsphase sollte man auf jeden Fall ein Konfigurationsmanagement benutzen. Damit können Änderungen am System protokolliert werden. Will man das System später überprüfen, so genügt es die Änderungen am System zu überprüfen. Außerdem kann man genau verfolgen, welche Annahmen über das System gemacht werden.

**Verifikation** Die Überprüfung der Implementierung lässt sich in zwei Teile einteilen. Zunächst einmal die funktionalen Tests (black box), bei denen überprüft wird, ob die Spezifikationen an das System durch die Implementierung eingehalten werden. Hierbei werden vor allem die Schnittstellen auf ihre Funktionalität getestet. Dazu kommen die *Struktur Tests*, hierbei wird der Code an sich analysiert und entsprechende Test Szenarien entwickelt, um diesen zu testen. Dabei wird oft auf Unit Tests zurückgegriffen, da diese sich automatisieren lassen und somit bei Änderungen an der Software einfach ausgeführt werden können. Des Weiteren minimieren die Unit Tests fehlerhafte Eingaben der Testdaten und verringern über längeren Zeitraum hinweg den Testaufwand.

### 4.4 Einsatz und Wartung

Wenn das System ausgeliefert ist und bereits eingesetzt wird, befindet man sich in der Einsatz- bzw. Wartungsphase. Auch wenn man sämtliche bekannten Fehler aus dem Programm entfernt hat, werden trotzdem einige Fehler erst nach dem Einsatz gefunden. Um diese Fehler zu beheben gibt es zwei Strategien, erstens die *Hot fixes*, dies sind schnelle Lösungen für Systemkritische Funktionen. *Hot fixes* haben die Aufgabe Sicherheitsmängel sowie Fehler in der Funktionsweise, die das System gefährden, möglichst schnell zu lösen. *Maintenance Releases* sind wie die *Hot fixes* ebenfalls dazu gedacht, Fehler in der System zu beheben. Sie sind

jedoch eher dafür gedacht, unkritische Fehler zu beheben oder auch Verbesserungen an *Hot fixes* vorzunehmen. Es kann zum Beispiel durch aus sinnvoll sein, eine Funktion des Systems mit einem *Hot fix* zu deaktivieren, um die Systemsicherheit zu erhöhen. Hier könnte ein *Maintenance Release* die Funktion entsprechend um Sicherheitsfunktionen erweitern und dann wieder aktivieren. Wichtig für die Sicherheit des System ist, das nach jedem *Hot fix* oder *Maintenance Release* die Sicherheit des Systems überprüft wird, um nicht neue Sicherheitslücken zu öffnen. Hierzu eignet sich das bereits oben erwähnte Konfigurationsmanagement gut, da hier die Änderungen genaustens protokolliert werden.

## 5 Schlusswort

Die Sicherheit eines Systems hängt stark von dem Entwurf des Systems ab. Hier wurde gezeigt, wie man mit Hilfe von einigen Entwurfsprinzipien typische Fehler vermeiden kann (vgl. [Bis04]). Außerdem wurde hier angesprochen, das auch vermeidlich sichere Funktionen zu einen Sicherheitsproblem führen können, wenn jemand versteckte Kanäle verwendet. Die einzelnen Maßnahmen die hier in den Entwurfsphasen angesprochen wurden, können nützlich sein, um die Sicherheit des Systems zu verbessern. Wichtig ist vor allem, da am Ende einer Entwurfsphase eine Überprüfung dieser Phase vorgenommen wird, da ansonsten Fehler an die folgenden Phasen übergeben werden.

## Literatur

- [Bis04] Matt Bishop: Computer Security – Art and Science. Addison-Wesley, 2004
- [ITSec] <http://www.bsi.de/zertifiz/itkrit/itsec.htm>
- [CSS04] Content Scramble System Specifications, Document Version 2.3, 2004
- [WikiDE] <http://www.wikipedia.de>