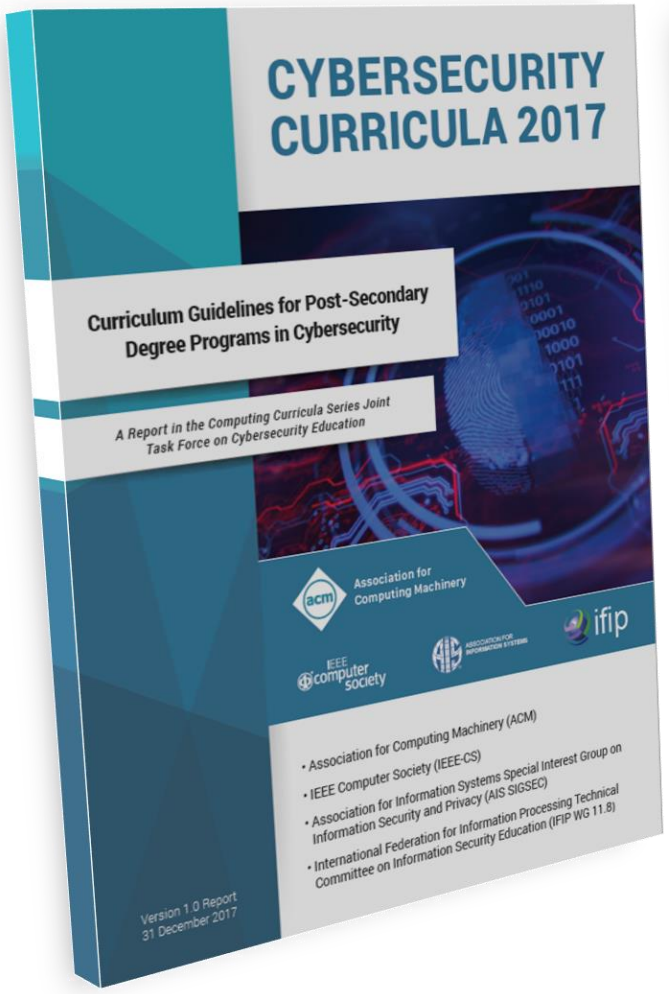
Segurança de Software I

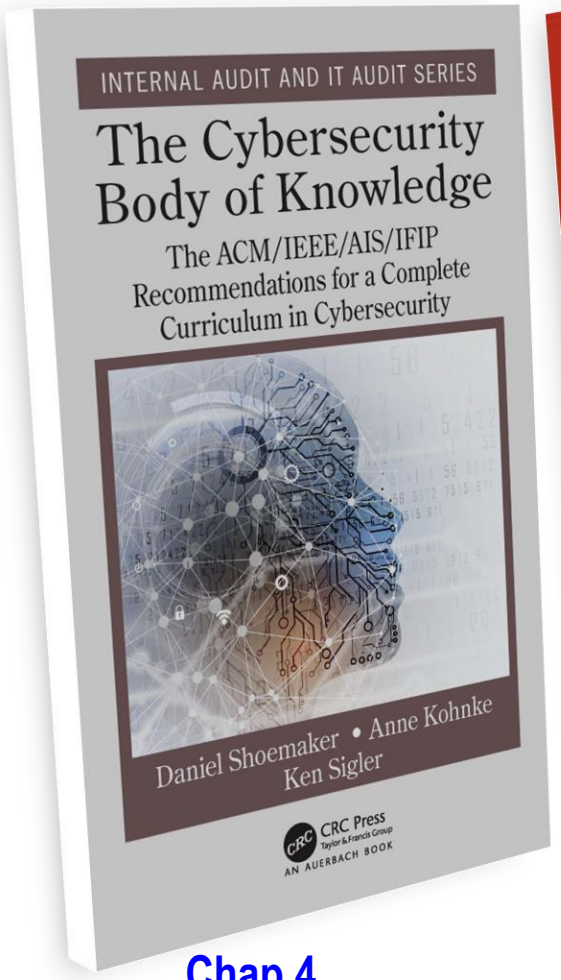
Ausberto S. Castro Vera

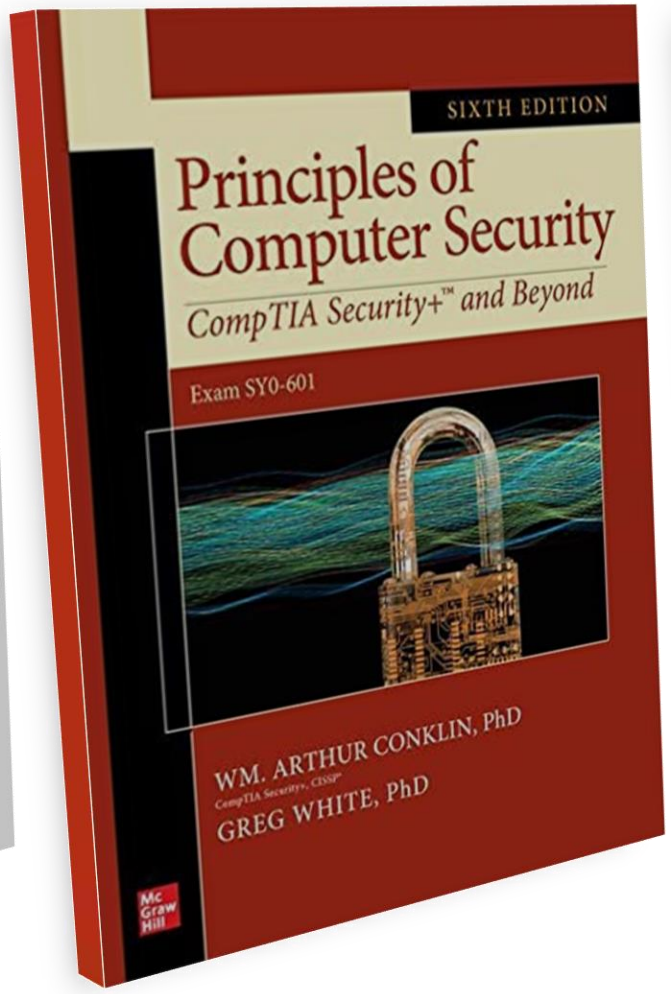
ascv@uenf.br

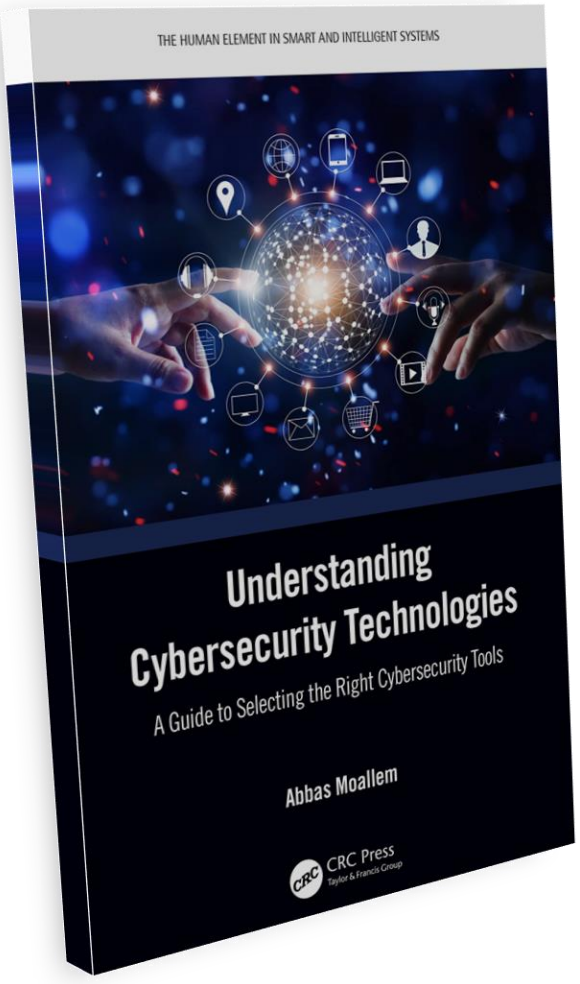
CIÊNCIA DA COMPUTAÇÃO
UENF

Bibliografia Básica



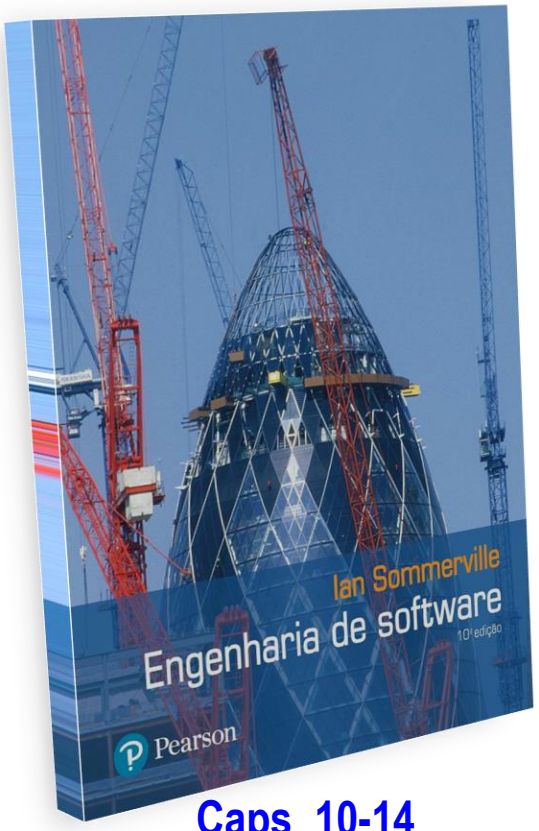




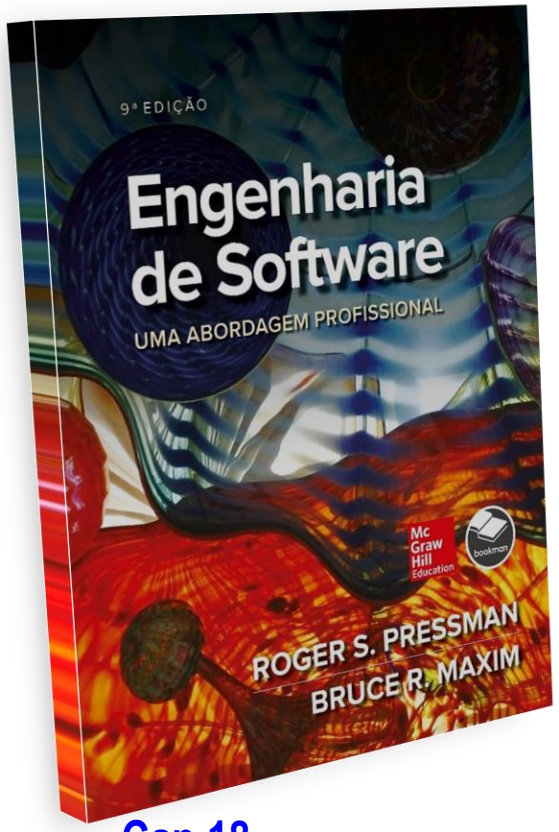


Chap 4
Software Security

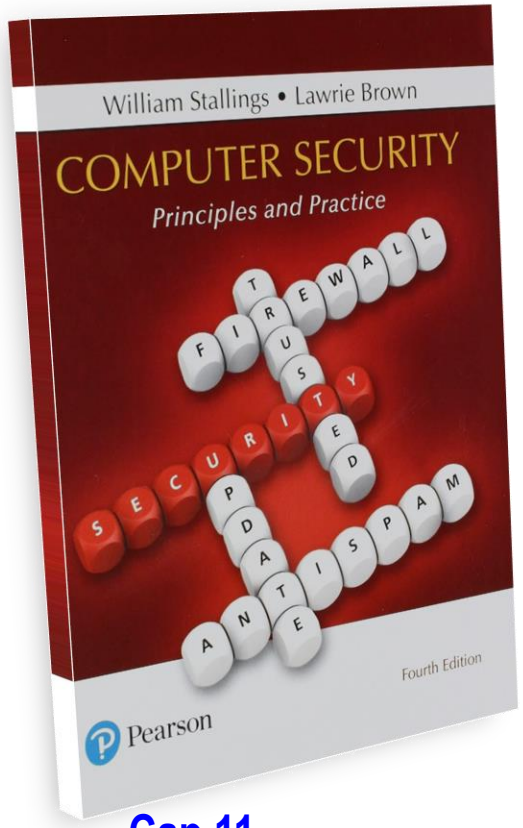
Bibliografia Complementar



Caps 10-14
Dependabilidade
e Segurança

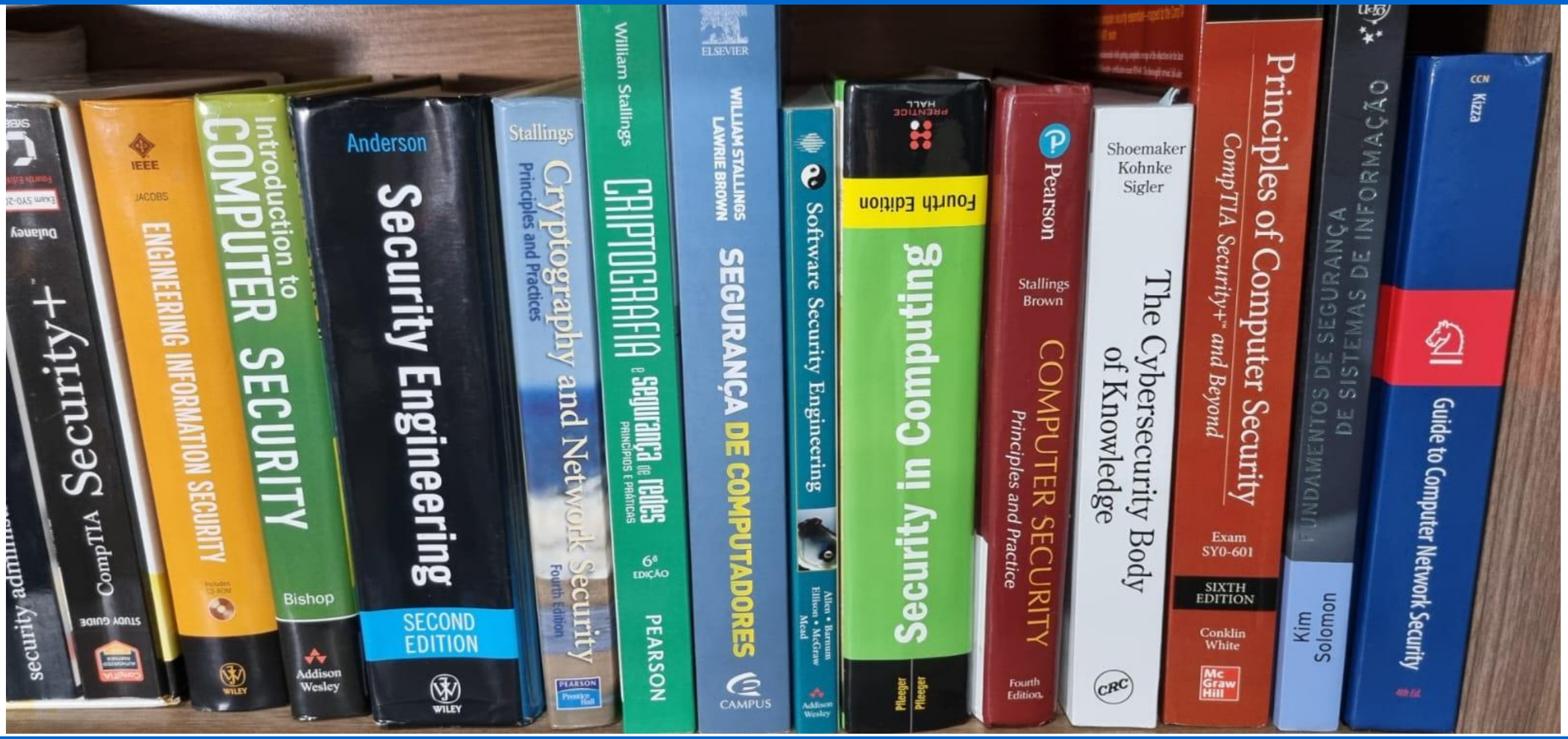


Cap 18
Engenharia de Segurança
de Software

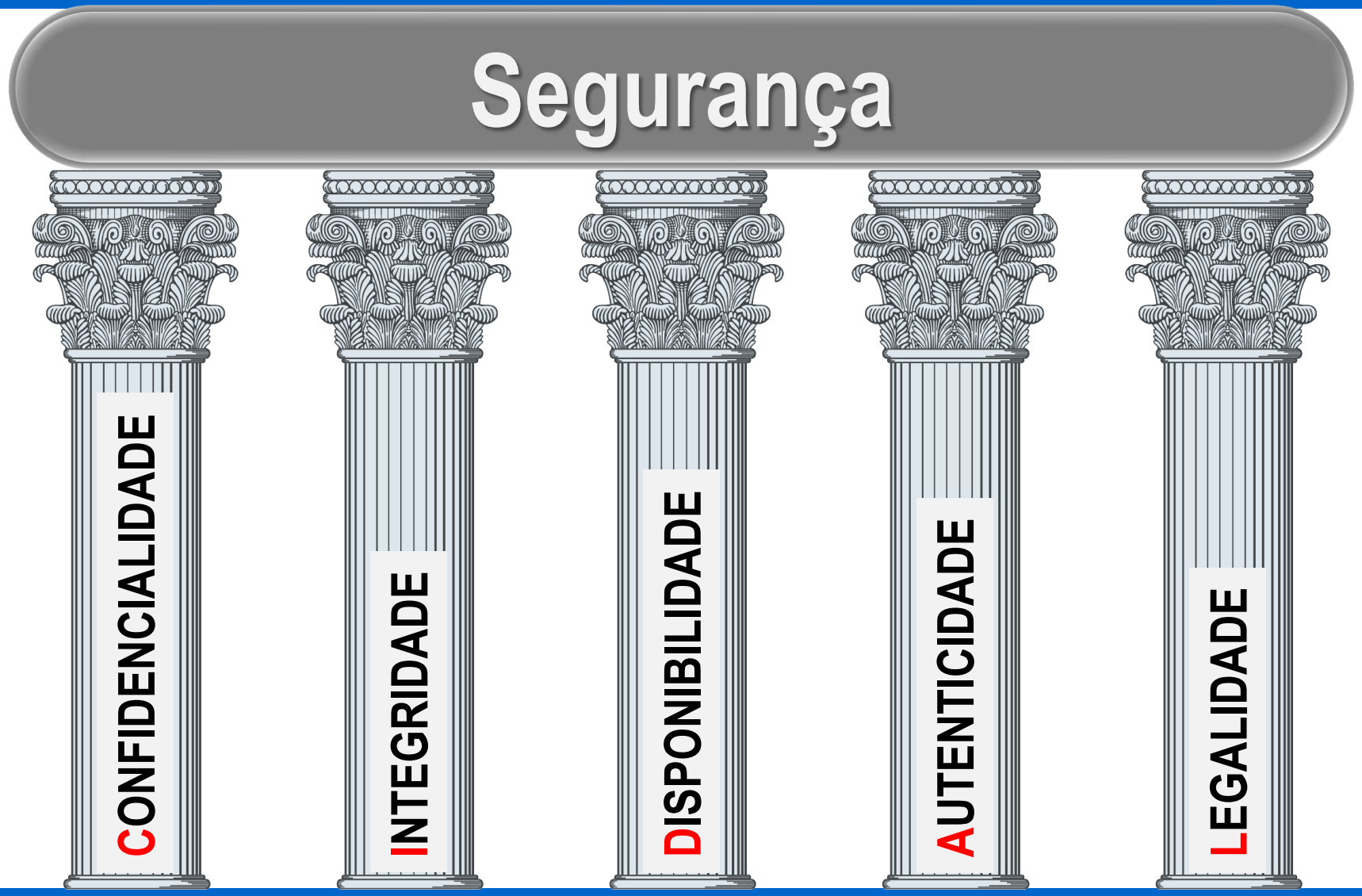


Cap 11
Software Security

Bibliografia Complementar



Arquitetura da Segurança Computacional



Sistema Computacional e Segurança

1. Confidencialidade
2. Integridade
3. Disponibilidade
4. Autencidade
5. Legalidade

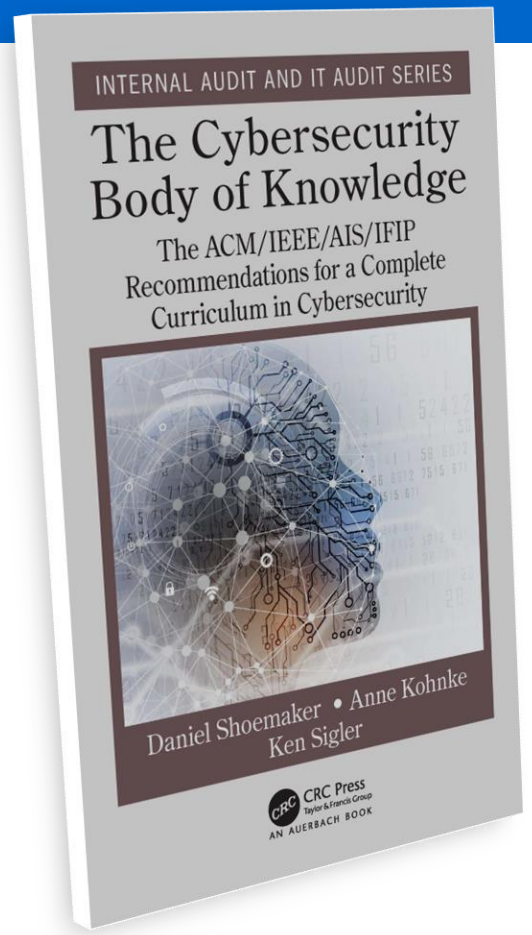
CID

CIDAL

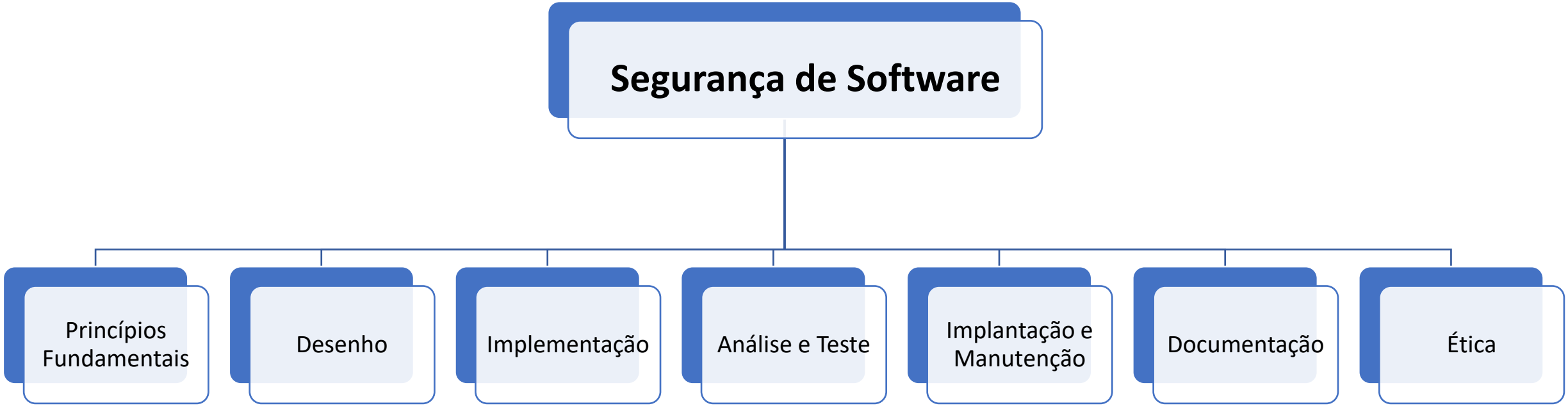
Segurança de Software

Segurança de Software
é o *desenvolvimento* e *aplicação* de políticas por meio de *mecanismos de defesa* sobre *dados* e *recursos*.

- As *políticas* de Segurança de Software especificam o que queremos impor e
- Os *mecanismos de defesa* especificam como aplicamos a política



Segurança de Software: 7 unidades



1. Princípios Fundamentais: 14 conceitos

1. Privilégio Mínimo
2. Padrão a prova de falha
3. Mediação Completa
4. Separação de deveres
5. Minimizar a confiança
6. Economia de Mecanismos
7. Minimizar mecanismo comum
8. Mínimo espanto
9. Projeto aberto
10. Camadas
11. Abstração
12. Modularidade
13. ligação completa
14. design para iteração

1.1 Privilégio Mínimo (POLP)

Privilégio mínimo:

significa que um usuário ou sistema interconectado deve ter apenas os *direitos e privilégios necessários* para executar sua tarefa atual, sem direitos e privilégios adicionais.

POLP = Principe of Least Privilege

A **limitação de privilégios**:

- Limita a quantidade de danos que podem ser causados, limitando assim a exposição do software a danos.
- Se for necessário atribuir diferentes níveis de segurança para tarefas separadas, é melhor alternar os níveis de segurança, em vez de executar o tempo todo com um nível de *privilégio mais alto*.
- *Contexto de segurança* no qual um software é executado: Todos os programas, scripts e arquivos em lote são executados no *contexto de segurança de um usuário específico* em um sistema operacional (SO).
- Evitar *acesso root:* o invasor poderia obter um shell no nível raiz. O software deve ser executado *apenas no contexto de segurança necessário* para que ele desempenhe suas funções com êxito.

1.1 Privilégio Mínimo (POLP) - Exemplos

Exemplos de aplicação do Princípio do Menor Privilégio (PoLP) na Segurança de Software:

1. **Controle de Acesso Baseado em Funções (RBAC):**
   Atribui permissões a usuários com base em suas funções e responsabilidades dentro da organização.
   Exemplo: Um *funcionário* do departamento de RH pode ter acesso ao sistema de folha de pagamento, enquanto um funcionário do departamento de vendas não.

2. **Autenticação Multifator (MFA):**
   Exige mais de um método de autenticação para verificar a identidade do usuário, como senha, impressão digital ou token de segurança.
   Exemplo: Ao fazer *login em um banco online*, o usuário precisa digitar sua senha e, em seguida, inserir um código enviado para seu celular.

3. **Least Privilege Application Security Model (LAPSM):**
   Controla o acesso de aplicativos a recursos do sistema, concedendo apenas os privilégios mínimos necessários para executar suas funções.
   Exemplo: Um *aplicativo antivírus* precisa ter acesso aos arquivos do sistema para detectar malwares, mas não precisa ter acesso às configurações do sistema.

1.1 Privilégio Mínimo (POLP) - Exemplos

Exemplos de aplicação do Princípio do Menor Privilégio (PoLP) na Segurança de Software:

4. **Sandboxing:**
   Executa aplicativos em um ambiente isolado, impedindo que eles acessem ou modifiquem outros arquivos ou processos no sistema.
   Exemplo: Um navegador web pode usar sandboxing para isolar sites maliciosos e proteger o computador do usuário.

5. **Desativação de privilégios administrativos:**
   Os usuários não devem trabalhar com contas administrativas no dia a dia, apenas para tarefas específicas que exigem tais privilégios.
   Exemplo: Um funcionário de escritório deve usar uma conta de usuário normal para navegar na internet e acessar e-mails, e apenas utilizar uma conta administrativa para instalar softwares ou realizar alterações no sistema.

6. **Atualizações de software e patches:**
   Manter o software atualizado com os últimos patches de segurança é crucial para corrigir vulnerabilidades e proteger contra ataques.
   Exemplo: Configurar atualizações automáticas para o sistema operacional e aplicativos garante que as falhas de segurança sejam corrigidas rapidamente.

1.1 Privilégio Mínimo (POLP) - Exemplos

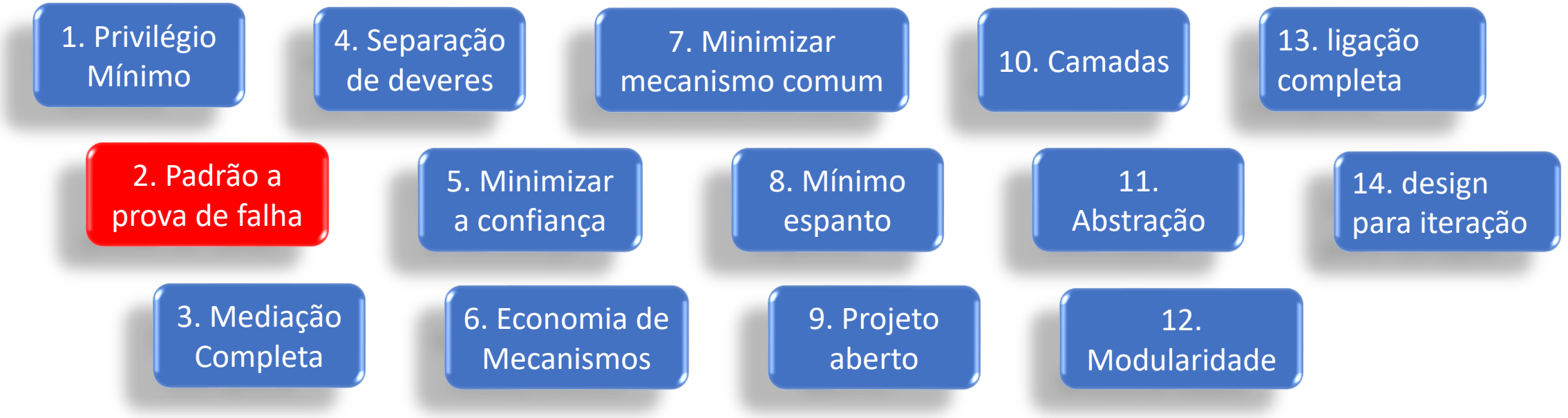
Exemplos de aplicação do Princípio do Menor Privilégio (PoLP) na Segurança de Software:

7. **Monitoramento e registro de atividades:**
   Monitorar e registrar as atividades dos usuários no sistema pode ajudar a detectar comportamentos suspeitos e identificar possíveis violações de segurança.
   Exemplo: Registrar logins, acessos a arquivos e alterações nas configurações do sistema permite analisar anomalias e investigar incidentes de segurança.

8. **Treinamento de *conscientização* em segurança:**
   Educar os usuários sobre os riscos de segurança cibernética e boas práticas de segurança pode ajudá-los a evitar ataques e proteger dados confidenciais.
   Exemplo: Realizar treinamentos regulares sobre phishing, engenharia social e uso seguro da internet conscientiza os funcionários sobre as principais ameaças e como se proteger.

9. **Implementação de uma *política de segurança* de software abrangente:**
   Uma política de segurança clara e bem definida define as regras e procedimentos para proteger os sistemas e dados da organização.
   Exemplo: A *política deve incluir diretrizes* para uso de senhas, controle de acesso, instalação de software, atualização de software, uso de dispositivos móveis e resposta a incidentes de segurança.

1. Princípios Fundamentais: 14 conceitos



1.2 Padrão do Software: a Prova de Falha

- **Verdade fundamental**: todos os sistemas sofrerão falhas.

- O **princípio do projeto à prova de falhas** caracteriza a importância de que, quando um sistema sofre uma falha, ele deve falhar para um *estado seguro*.
  - Uma forma de implementação é usar o conceito de **negação explícita**. Qualquer função que não seja especificamente autorizada é negada por padrão.
  - Quando um sistema *entra em estado de falha*, os atributos associados à segurança, confidencialidade, integridade e disponibilidade precisam ser mantidos de forma adequada.

  - **Disponibilidade** é o atributo que tende a causar maiores dificuldades de projeto.

Benefícios:

- **Redução do tempo de inatividade:** ao prevenir falhas e garantir a continuidade das operações, o software à prova de falhas pode reduzir o tempo de inatividade do sistema, o que aumenta a produtividade e a satisfação do cliente.

- **Melhoria da segurança:** o software à prova de falhas pode ajudar a proteger sistemas contra ataques cibernéticos e outras ameaças, o que protege dados confidenciais e garante a integridade do sistema.

1.2 Padrão do Software: a Prova de Falha

EXEMPLOS:

1. Sistemas de alta disponibilidade
2. Sistemas tolerantes a falhas
3. Software com autocorreção
4. Design de software robusto
5. Arquitetura de microsserviços

1.2 Padrão do Software: a Prova de Falha

Sistemas de alta disponibilidade

- **Replicação de dados:** cópias de dados são armazenadas em diferentes servidores para garantir que os dados estejam sempre disponíveis, mesmo em caso de falha de um servidor.
- **Balanceamento de carga:** distribui o tráfego entre vários servidores para evitar sobrecarga e garantir que o sistema continue funcionando mesmo que um servidor falhe.
- **Failover automático:** em caso de falha de um servidor, outro servidor assume automaticamente as suas funções para garantir a continuidade das operações.

- **Exemplo:** Um site de e-commerce com alto volume de tráfego precisa estar sempre disponível para seus clientes. Através da replicação de dados, balanceamento de carga e failover automático, o site pode garantir a disponibilidade mesmo em caso de falhas de hardware ou software.

1.2 Padrão do Software: a Prova de Falha

Sistemas tolerantes a falhas

- **Detecção de falhas:** o sistema monitora continuamente seus componentes e identifica falhas o mais rápido possível.
- **Isolamento de falhas:** quando uma falha é detectada, o componente defeituoso é isolado para evitar que afete o restante do sistema.
- **Reconfiguração automática:** o sistema se reconfigura automaticamente para continuar funcionando com os componentes restantes, mesmo após a falha de um componente.

Exemplo: Um sistema de controle de tráfego aéreo precisa ser tolerante a falhas, pois qualquer falha no sistema pode ter consequências graves. Através da detecção, isolamento e reconfiguração automática de falhas, o sistema pode garantir a segurança e a continuidade das operações mesmo em caso de falhas de hardware ou software.

1.2 Padrão do Software: a Prova de Falha

Software com autocorreção

- **Verificação de erros:** o software verifica continuamente seus dados e código para identificar erros e falhas.
- **Correção automática de erros:** quando um erro é detectado, o software tenta corrigi-lo automaticamente. Atualiza variáveis a valores iniciais ou fixos
- **Relato de erros:** se um erro não puder ser corrigido automaticamente, o software registra o erro para que seja investigado e corrigido posteriormente (envia log para o responsável.

Exemplo: Um *software de antivírus* precisa ser capaz de detectar e remover malwares automaticamente. Através da verificação de erros, correção automática de erros e relato de erros, o software pode proteger o computador do usuário contra ameaças sem a necessidade de intervenção manual.

1.2 Padrão do Software: a Prova de Falha

Design de software robusto

- **Princípio da mínima surpresa:** o software se comporta da maneira esperada pelo usuário, mesmo em situações inesperadas.
- **Tratamento de exceções:** o software lida com erros e falhas de forma previsível e consistente, evitando que o sistema trave ou se comporte de forma inesperada.
- **Testes rigorosos:** o software é *submetido a testes rigorosos* para identificar e corrigir falhas *antes de ser lançado* ao público.

Exemplo: Um *aplicativo de banco* precisa ser robusto e confiável para garantir a segurança das transações financeiras dos usuários. Através do design de software robusto, o aplicativo pode lidar com erros e falhas de forma segura e previsível, evitando perdas financeiras para os usuários.

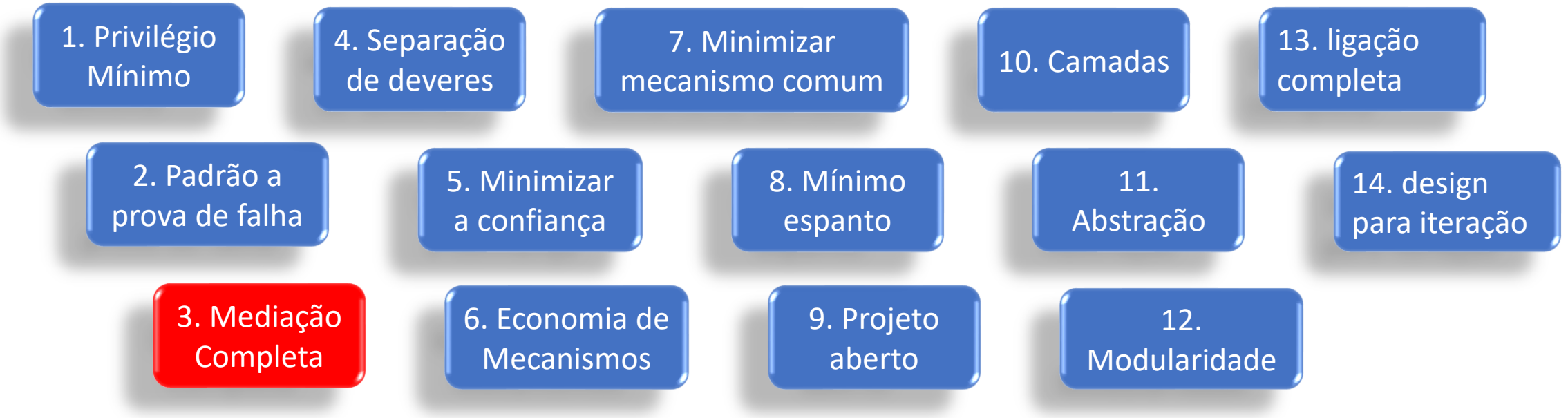
1.2 Padrão do Software: a Prova de Falha

Arquitetura de microsserviços

- **Serviços independentes:** cada serviço é um módulo independente com sua própria responsabilidade, o que facilita a identificação e o isolamento de falhas.
- **Comunicação assíncrona:** os serviços se comunicam entre si de forma assíncrona, o que evita que um serviço lento ou defeituoso bloqueie todo o sistema.
- **Escalabilidade:** a arquitetura de microsserviços permite que o sistema seja facilmente escalonado para atender às demandas de carga, o que aumenta a disponibilidade e a resiliência do sistema.

Exemplo: Uma *plataforma de e-commerce* com alto volume de tráfego precisa ser escalável e resiliente. Através da arquitetura de microsserviços, a plataforma pode distribuir a carga entre vários serviços, o que permite que ela continue funcionando mesmo em caso de aumento repentino de tráfego ou falha de um serviço.

1. Princípios Fundamentais: 14 conceitos

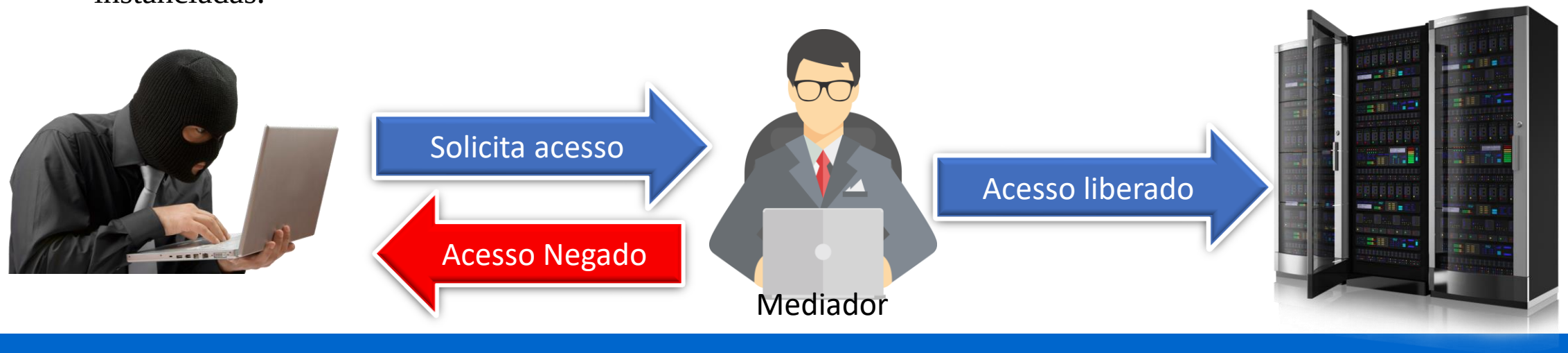


1.3 Mediação Completa

O **princípio da mediação completa**:

quando a autorização é verificada em relação a um **objeto** e a uma **ação**, essa verificação ocorre sempre que o acesso a esse objeto é solicitado.

- O sistema deve ser concebido de modo a que o sistema de autorização nunca seja contornado, mesmo com acessos múltiplos e repetidos.
- Os sistemas operacionais e sistemas de TI modernos com sistemas de autenticação configurados corretamente são muito difíceis de comprometer diretamente, com a maioria das rotas de comprometimento envolvendo o desvio de um sistema crítico como a autenticação.
- é importante durante o projeto examinar possíveis situações de desvio e evitar que elas sejam instanciadas.



1.3 Mediação Completa

A **mediação completa** é um modelo de segurança de software que visa proteger sistemas e dados contra acessos não autorizados, modificações e uso indevido.

Esse modelo se baseia em três princípios fundamentais:

1. **Separação de Sujeitos:**
   Divide os usuários do sistema em *diferentes grupos* com base em suas funções e responsabilidades.
   Exemplo: Um sistema de banco online pode ter grupos de usuários para clientes, gerentes e administradores, cada um com permissões de acesso específicas.

2. **Separação de Objetos:**
   Divide os recursos do sistema em diferentes categorias com base em seu tipo e função.
   Exemplo: Um sistema de gerenciamento de estoque pode ter categorias de objetos para produtos, pedidos e fornecedores, cada um com atributos e funcionalidades específicas.

3. **Controle de Fluxo de Informação:**
   Define regras que controlam como os sujeitos podem acessar e modificar os objetos.
   Exemplo: Um cliente do banco online pode visualizar seu saldo bancário, mas apenas um gerente pode aprovar um pedido de empréstimo.

1.3 Mediação Completa

Exemplos de aplicação da mediação completa em segurança de software:

1. **Sistemas Operacionais:** controlam o acesso a recursos do sistema como arquivos, memória e dispositivos de entrada/saída.
2. **Bancos de Dados:** controlam o acesso a dados armazenados em um banco de dados, como tabelas, linhas e colunas.
3. **Aplicações Web:** controlam o acesso a páginas da web, componentes e dados.
4. **Sistemas Distribuídos:** controlam a comunicação e o acesso a recursos em sistemas distribuídos.

Benefícios da mediação completa:

- **Maior segurança:** protege sistemas e dados contra acessos não autorizados, modificações e uso indevido.
- **Flexibilidade:** permite definir regras de acesso granulares e adaptáveis às necessidades específicas da organização.
- **Escalabilidade:** pode ser usada em sistemas de qualquer tamanho e complexidade.
- **Transparência:** facilita o rastreamento e a auditoria de acessos a recursos do sistema.

Desafios da mediação completa:

- **Implementação complexa:** pode ser difícil de implementar e gerenciar em sistemas grandes e complexos.
- **Gerenciamento de permissões:** a definição e o gerenciamento de permissões de acesso podem ser trabalhosos e demorados.
- **Desempenho:** a verificação de permissões pode afetar o desempenho do sistema, especialmente em sistemas com alto volume de tráfego.

1.3 Mediação Completa

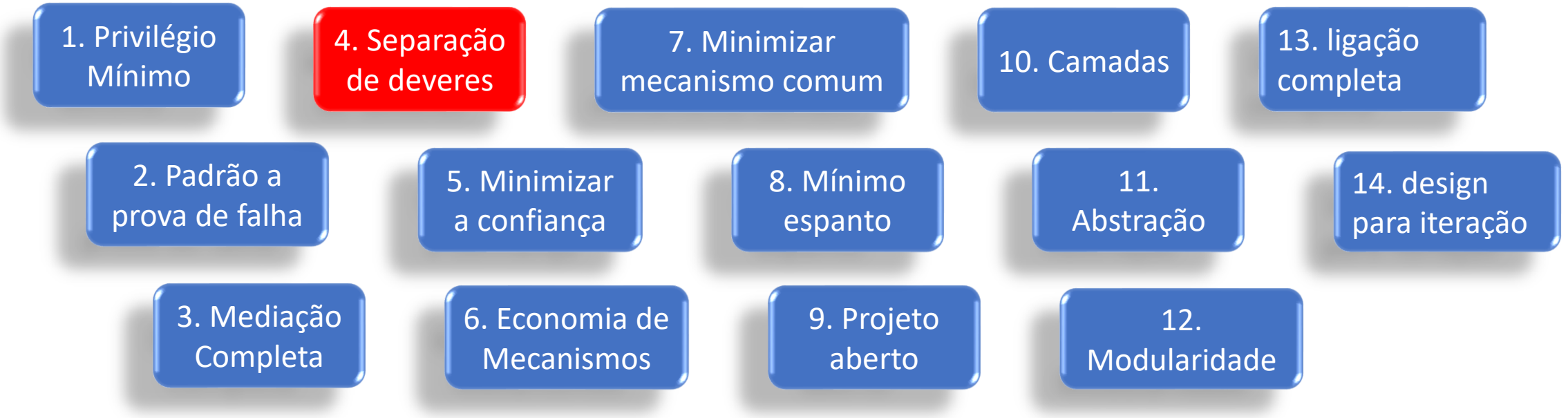
Exemplo prático

Considere um sistema de gerenciamento de notas para uma escola. A **mediação completa** pode ser usada para controlar o acesso às notas dos alunos da seguinte maneira:

- **Sujeitos:** alunos, professores, diretores.
- **Objetos:** notas dos alunos, avaliações, comentários.
- **Regras de acesso:**
  - Alunos: podem visualizar suas próprias notas e comentários.
  - Professores: podem visualizar as notas de seus alunos e adicionar comentários.
  - Diretores: podem visualizar as notas de todos os alunos, adicionar comentários e modificar as configurações do sistema.

Ao implementar essas regras de acesso, a **mediação completa** garante que apenas os usuários autorizados tenham acesso às notas dos alunos, protegendo a privacidade dos alunos e a integridade dos dados.

1. Princípios Fundamentais: 14 conceitos



1.4 Separação de Deveres ou funções

A **Separação de Funções** **(SoD,** *Separation of Duties***)** é um princípio fundamental da segurança de software que visa prevenir fraudes, erros e acessos não autorizados por meio da *divisão de tarefas e responsabilidades entre diferentes usuários ou funções.* Ao implementar a SoD corretamente, as empresas podem fortalecer significativamente a segurança de seus sistemas e dados.

A **separação de funções**

garante que, *para qualquer tarefa*, mais de um indivíduo precise estar envolvido.

O caminho crítico das tarefas é dividido em vários itens, que são então distribuídos por mais de uma entidade. Ao implementar uma tarefa desta maneira, nenhum indivíduo pode abusar do sistema.

Um *exemplo simples* pode ser um sistema em que um indivíduo é obrigado a *fazer um pedido* e uma pessoa separada é necessária para *autorizar a compra*.

Os *componentes de software* impõem a separação de tarefas quando exigem que *diversas condições* sejam atendidas *antes que uma tarefa seja considerada concluída*. Essas múltiplas condições podem então ser gerenciadas separadamente para *impor as* verificações e equilíbrios exigidos pelo sistema.

1.4 Separação de Deveres ou funções

EXEMPLOS

1. **Sistema de Folha de Pagamento:**
   ***Função 1: Lançamento de Horas*:** responsável por registrar as horas trabalhadas pelos funcionários.
   ***Função 2: Cálculo de Salário*:** responsável por calcular o salário dos funcionários com base nas horas trabalhadas e na taxa de pagamento.
   ***Função 3: Aprovação de Folha de Pagamento*:** responsável por aprovar a folha de pagamento antes do pagamento aos funcionários.

2. **Sistema de Compras:**
   ***Função 1: Solicitação de Compra*:** responsável por criar solicitações de compra para produtos ou serviços necessários.
   ***Função 2: Aprovação de Compra*:** responsável por aprovar ou negar solicitações de compra com base no orçamento e na necessidade.
   ***Função 3: Processamento de Compra*:** responsável por realizar o pedido, receber o produto ou serviço e realizar o pagamento.

1.4 Separação de Deveres ou funções

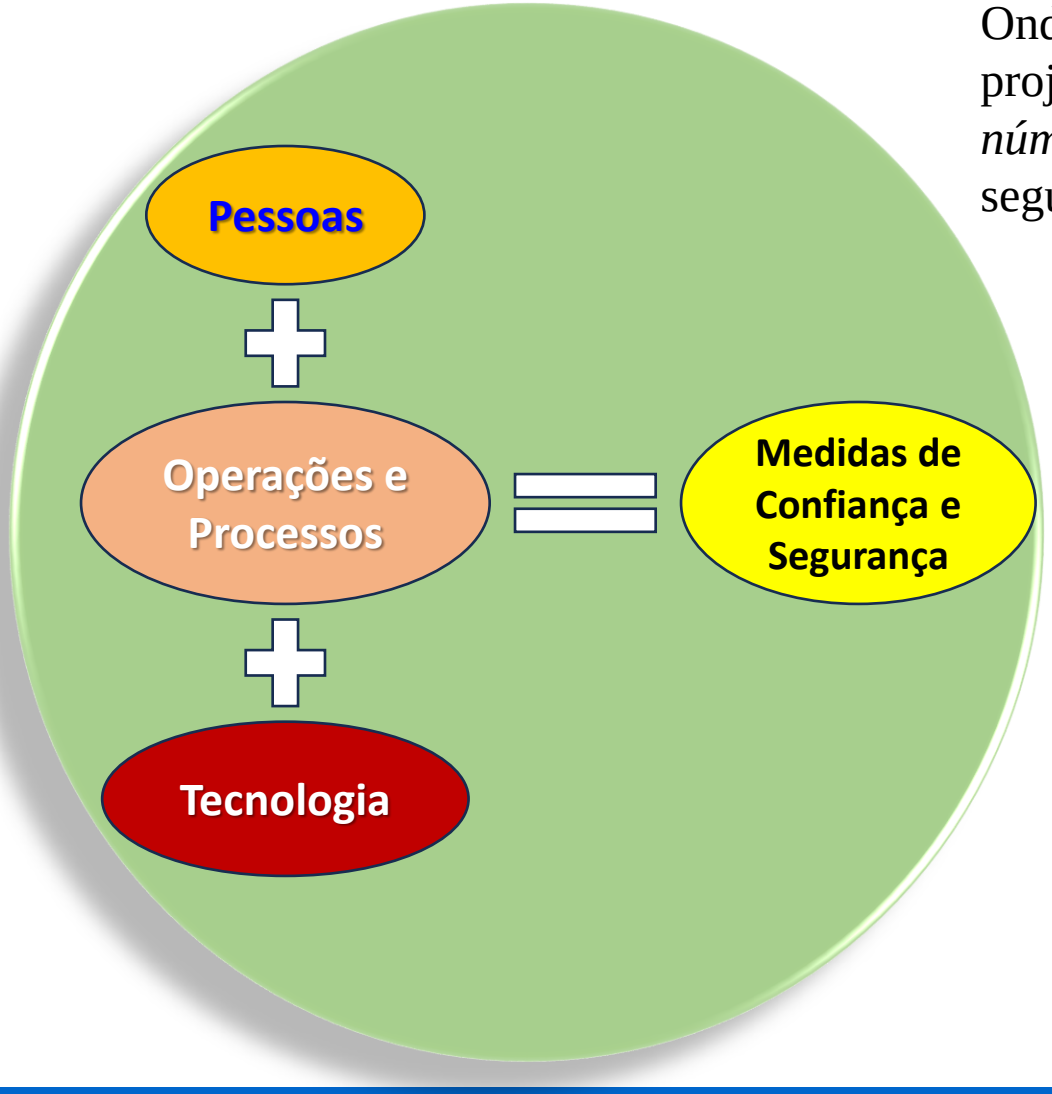
EXEMPLOS

3. **Sistema de Controle de Acesso:**
   ***Função 1: Solicitação de Acesso*:** responsável por solicitar acesso a recursos do sistema para usuários ou funções específicas.
   ***Função 2: Aprovação de Acesso*:** responsável por aprovar ou negar solicitações de acesso com base nas necessidades e no nível de autorização do usuário.
   ***Função 3: Gerenciamento de Acesso*:** responsável por conceder, revogar e modificar permissões de acesso aos recursos do sistema.

4. **Sistema de Vendas Online:**
   ***Função 1: Recebimento de Pedidos*:** responsável por receber e processar pedidos dos clientes online.
   ***Função 2: Separação e Embalagem*:** responsável por separar e embalar os produtos pedidos pelos clientes.
   ***Função 3: Expedição e Entrega*:** responsável por expedir e entregar os produtos aos clientes.

1. Princípios Fundamentais: 14 conceitos

1.5 Minimizar a Confiança



Onde a tecnologia é usada, o *hardware*, o *firmware* e o *software* devem ser projetados e implementados de modo que seja necessário *confiar em um número mínimo de elementos* do sistema, a fim de manter o nível de segurança desejado.

O princípio da **Minimização da Confiança** na segurança de software visa *reduzir o nível de confiança depositado em qualquer componente, recurso ou usuário* do sistema.

Isso significa que o sistema deve ser projetado de forma que nenhum componente ou usuário tenha mais privilégios do que o necessário para realizar suas funções, e que falhas ou comportamentos maliciosos em um componente não afetem todo o sistema

Minimização da confiança

- é assumir que todos os sistemas externos interconectados são inseguros.

- deve haver uma presunção de que as medidas de segurança de um sistema externo são diferentes daquelas de um sistema interno confiável

1.5 Minimizar a Confiança

EXEMPLOS

1. **Sistema Operacional:**
   - **Princípio da Menor Permissão:** conceder aos usuários apenas os privilégios mínimos necessários para realizar suas tarefas.
   - **Sandboxing:** executar aplicativos em ambientes isolados para evitar que eles acessem ou modifiquem outros arquivos ou processos no sistema.
   - **Controle de Acesso Baseado em Função (RBAC):** atribuir permissões a usuários com base em suas funções e responsabilidades dentro da organização.

2. **Rede:**
   - **Segmentação de Rede:** dividir a rede em diferentes segmentos para isolar áreas críticas e limitar o acesso a recursos confidenciais.
   - **Firewalls:** implementar firewalls para controlar o tráfego de rede e bloquear acessos não autorizados.
   - **Criptografia:** criptografar dados confidenciais em trânsito e em repouso para protegê-los contra interceptação e espionagem.

1.5 Minimizar a Confiança

EXEMPLOS

3. **Aplicativos Web:**
   - **Validação de Entrada de Dados:** validar rigorosamente todos os dados inseridos pelos usuários para evitar ataques de injeção de código e outras vulnerabilidades.
   - **Codificação Segura:** usar técnicas de codificação segura para evitar falhas de software que possam ser exploradas por invasores.
   - **Testes de Penetração:** realizar testes de penetração regularmente para identificar e corrigir vulnerabilidades de segurança em aplicativos web.

4. **Sistemas Distribuídos:**
   - **Autenticação e Autorização:** garantir que apenas usuários autorizados tenham acesso aos recursos do sistema.
   - **Criptografia de Comunicação:** criptografar a comunicação entre os componentes do sistema para protegê-la contra interceptação e espionagem.
   - **Tolerância a Falhas:** projetar o sistema para ser tolerante a falhas de componentes, garantindo que ele continue funcionando mesmo em caso de falhas.

1. Princípios Fundamentais: 14 conceitos

1. Privilégio Mínimo
2. Padrão a prova de falha
3. Mediação Completa
4. Separação de deveres
5. Minimizar a confiança
6. Economia de Mecanismos
7. Minimizar mecanismo comum
8. Mínimo espanto
9. Projeto aberto
10. Camadas
11. Abstração
12. Modularidade
13. ligação completa
14. design para iteração

1.6 Economia de Mecanismos

- Significa *manter o design* de software seguro tão *simples e pequeno* quanto possível.

- Este princípio bem conhecido se aplica a qualquer aspecto de um sistema, mas merece ênfase em uma discussão sobre Segurança de Software porque erros de projeto e implementação que resultam em caminhos de acesso indesejados não serão percebidos durante o uso normal (uma vez que o uso normal geralmente não inclui tentativas de utilizar caminhos de acesso impróprios).

- Como resultado, são necessárias *técnicas*
  - inspeção linha por linha de software e
  - *exame físico de hardware* que implementa mecanismos de segurança.

- Para que tais técnicas tenham sucesso, um *design pequeno e simples* é essencial.

1.6 Economia de Mecanismos

Exemplos práticos de Economia de Mecanismos :

1. Design de Software:

- **Princípio do Menor Privilégio:** conceder aos usuários e processos apenas os privilégios mínimos necessários para realizar suas tarefas.
- **Princípio da Defesa em Profundidade:** implementar várias camadas de segurança para proteger o sistema contra diferentes tipos de ataques.
- **Fail-Safe Defaults:** definir configurações padrão seguras que minimizem o risco de erros de configuração.

2. Desenvolvimento de Software:

- **Codificação Segura:** usar técnicas de codificação segura para evitar falhas de software que possam ser exploradas por invasores.
- **Teste de Software Rigoroso:** realizar testes rigorosos de software para identificar e corrigir falhas de segurança antes da implantação do sistema.
- **Monitoramento de Segurança Contínuo:** monitorar os sistemas de segurança em tempo real para detectar e responder a incidentes de segurança rapidamente.

3. Implementação de Segurança:

- **Uso de Ferramentas de Segurança Padrão:** utilizar ferramentas de segurança de código aberto ou comercialmente disponíveis que sejam confiáveis e bem testadas.
- **Automação de Tarefas de Segurança:** automatizar tarefas de segurança repetitivas para reduzir custos e aumentar a eficiência.
- **Treinamento de Conscientização em Segurança:** conscientizar os usuários sobre os riscos de segurança cibernética e boas práticas de segurança.

1. Princípios Fundamentais: 14 conceitos

1.7 Minimizar Mecanismo Comum

A **Minimização do Mecanismo Comum (MMC)** na segurança de software
visa *reduzir a dependência de um único mecanismo de segurança* para proteger um sistema. Isso significa que o *sistema deve ser projetado* utilizando diversos mecanismos de segurança complementares, evitando que a falha ou comprometimento de um *único mecanismo comprometa a segurança geral do sistema*.

Minimização do Mecanismo Comum ("mecanismo menos comum")
é que um método de design deve impedir o *compartilhamento inadvertido* de informações.

- Ter vários processos compartilhando mecanismos comuns leva a um caminho potencial de informações entre usuários ou processos.
- Ter um mecanismo que atenda a uma ampla gama de usuários ou processos impõe uma carga mais significativa ao projeto desse processo para manter todos os caminhos separados.
- Quando confrontado com a escolha entre um único processo que opera em uma variedade de objetos de nível supervisório e subordinado e/ou um processo especializado adaptado a cada um, *escolher o processo separado* é a **melhor escolha**.
- Os conceitos de *mecanismo menos comum* e de *aproveitamento de componentes* existentes podem colocar um projetista em uma encruzilhada conflitante: um conceito defende a **reutilização** e o outro a **separação**. A escolha consiste em determinar o equilíbrio correto associado ao risco de cada um.

1.7 Minimizar Mecanismo Comum

EXEMPLOS

1. Autenticação:

- **Utilizar diversos fatores de autenticação:** combinar senhas com outros fatores de autenticação, como biometria, tokens de segurança ou autenticação de dois fatores (2FA).
- **Implementar autenticação multifator (MFA):** exigir a utilização de pelo menos dois fatores de autenticação para acessar o sistema.
- **Utilizar diferentes métodos de autenticação para diferentes tipos de usuários:** por exemplo, exigir autenticação mais robusta para usuários com acesso a dados confidenciais.

2. Controle de Acesso:

- **Implementar RBAC (Controle de Acesso Baseado em Funções):** atribuir permissões a usuários com base em suas funções e responsabilidades dentro da organização.
- **Utilizar listas de controle de acesso (ACLs):** definir quais usuários e grupos têm acesso a quais recursos do sistema.
- **Implementar o princípio do Menor Privilégio:** conceder aos usuários apenas os privilégios mínimos necessários para realizar suas tarefas.

1.7 Minimizar Mecanismo Comum

EXEMPLOS

3. Proteção de Dados:

- **Criptografar dados em repouso e em trânsito:** proteger dados confidenciais contra acesso não autorizado, mesmo que o sistema seja comprometido.
- **Utilizar técnicas de anonimização e pseudonimização de dados:** reduzir o risco de exposição de dados pessoais em caso de violação de segurança.
- **Implementar controles de acesso rigorosos para dados confidenciais:** limitar o acesso a dados confidenciais apenas aos usuários que realmente precisam deles.

4. Rede:

- **Implementar firewalls:** controlar o tráfego de rede e bloquear acessos não autorizados.
- **Utilizar técnicas de segmentação de rede:** dividir a rede em diferentes segmentos para isolar áreas críticas e limitar o acesso a recursos confidenciais.
- **Implementar IPS (Sistemas de Prevenção de Intrusão):** detectar e bloquear atividades maliciosas na rede.

1. Princípios Fundamentais: 14 conceitos

1.8 Mínimo Espanto

- Às vezes chamado de *"aceitabilidade psicológica"*
- O **Princípio do Menor Espanto** (Least Astonishment Principle - **LAP**) na segurança de software visa projetar sistemas que sejam *intuitivos e fáceis de usar para os usuários*, minimizando a surpresa e a frustração causadas por comportamentos inesperados ou inconsistentes.

- LAP enfatiza que os **usuários** são uma *parte fundamental* do software e de sua segurança.
- Incluir um usuário na segurança de um sistema requer que os aspectos de segurança sejam projetados de forma que sejam de *menor complexidade (o menor espanto) para o usuário*.
- Quando um usuário é apresentado a um sistema de segurança que *parece obstruir sua produtividade*, o resultado será o *usuário contornando os aspectos de segurança* do sistema.
- *Por exemplo*,
  - se um sistema proíbe o envio de certos tipos de anexos por e-mail, o usuário pode criptografar o anexo, mascarando-o da segurança, e executar a ação proibida de qualquer maneira.

- **A *facilidade de uso* tende a superar muitos aspectos funcionais.**

- O *design da segurança em sistemas de software precisa ser transparente para o usuário*, assim como o ar – invisível, mas sempre presente, atendendo às necessidades.
- A segurança é um elemento *funcional crítico*, mas que não deve impor nenhum *ônus ao usuário*.

1.8 Mínimo Espanto - Exemplos

1. Interfaces de Usuário:

- **Design intuitivo e consistente:** utilizar ícones, menus e layouts fáceis de entender e seguir, seguindo padrões e convenções comuns.
- **Feedback claro e informativo:** fornecer feedback claro e informativo aos usuários sobre suas ações, resultados e possíveis erros.
- **Minimizar mensagens de erro:** evitar mensagens de erro genéricas ou confusas e fornecer instruções claras para que os usuários possam resolver problemas.

2. Comportamento do Sistema:

- **Comportamento previsível e consistente:** o sistema deve se comportar de forma previsível e consistente em diferentes situações, evitando surpresas para os usuários.
- **Evitar mudanças inesperadas:** evitar mudanças inesperadas na interface do usuário, no comportamento do sistema ou nas configurações padrão.
- **Fornecer opções de personalização:** permitir que os usuários personalizem algumas configurações do sistema para atender às suas necessidades e preferências.

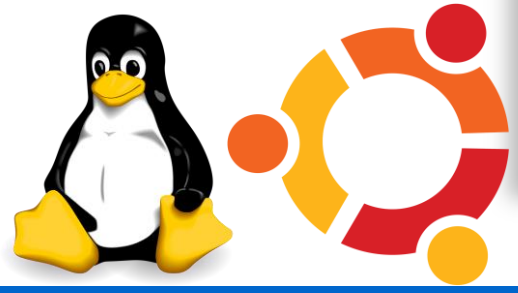
3. Mensagens e Alertas:

- **Linguagem clara e concisa:** utilizar linguagem clara, concisa e fácil de entender nas mensagens e alertas do sistema.
- **Evitar jargões técnicos:** evitar o uso de jargões técnicos ou termos obscuros que os usuários comuns não entenderão.
- **Fornecer contexto e informações adicionais:** fornecer contexto e informações adicionais sobre as mensagens e alertas para que os usuários possam tomar decisões informadas.

1. Princípios Fundamentais: 14 conceitos

1.9 Projeto Aberto

- O conceito de **design aberto** afirma que **a segurança de um sistema deve ser independente da forma do design de segurança**.
- Em essência, o **algoritmo utilizado** será *aberto e acessível* e *a segurança não deve depender do design*, mas sim de um elemento como uma chave.
- A **criptografia moderna** empregou este princípio de forma eficaz: *a segurança depende do sigilo da chave, não do sigilo do algoritmo empregado*.



Ferramentas de código aberto para segurança de software oferecem diversas vantagens, como:

- **Custo-benefício:** são gratuitas ou de baixo custo, o que as torna acessíveis a empresas e indivíduos com orçamentos limitados.
- **Flexibilidade:** podem ser personalizadas e adaptadas às necessidades específicas de cada organização.
- **Transparência:** o código-fonte está disponível para análise e auditoria por qualquer pessoa, o que aumenta a confiabilidade e a segurança das ferramentas.
- **Grande comunidade:** contam com uma grande comunidade de desenvolvedores e usuários que contribuem para o aprimoramento contínuo das ferramentas.

1.9 Projeto Aberto - Exemplos

1. Ferramentas de Análise de Vulnerabilidades
2. Ferramentas de Teste de Penetração
3. Ferramentas de Criptografia
4. Ferramentas de Monitoramento de Segurança
5. Ferramentas de Gerenciamento de Identidade e Acesso (IAM)

1. Ferramentas de Análise de Vulnerabilidades:

- **Nmap:** scanner de rede para identificar hosts ativos, serviços em execução e possíveis vulnerabilidades.
- **OpenVAS:** scanner de vulnerabilidades abrangente que verifica diversos tipos de sistemas e aplicativos.
- **Nikto:** scanner web que verifica vulnerabilidades em servidores web e aplicativos web.
- **Nessus:** scanner de vulnerabilidades comercial com versão gratuita para uso pessoal e comunitário.

2. Ferramentas de Criptografia:

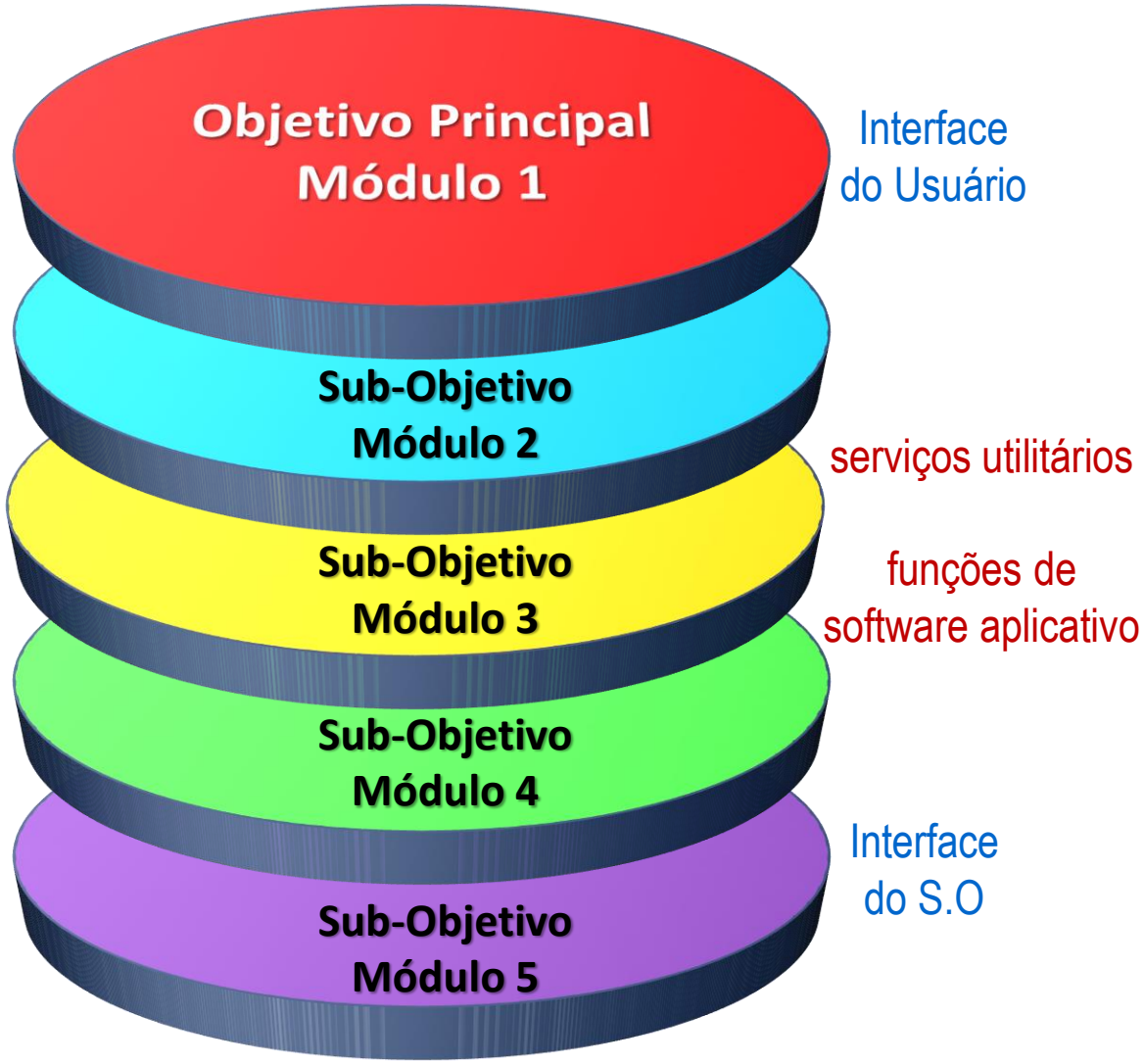
- **GnuPG:** ferramenta de criptografia multiplataforma que suporta diversos algoritmos de criptografia e assinaturas digitais.
- **OpenSSL:** biblioteca de criptografia abrangente que inclui ferramentas para criptografia simétrica, assimétrica, hash e protocolos de segurança.
- **TrueCrypt:** software de criptografia de disco completo que permite criptografar todo o disco rígido ou partições específicas.
- **VeraCrypt:** fork do TrueCrypt que oferece recursos aprimorados de segurança e compatibilidade.

1. Princípios Fundamentais: 14 conceitos

1.10 Camadas

- Utiliza o conceito de de **Separação de Assuntos** (SOC, Separations of Concerns)
- Uma prática comum é projetar o software decompondo o **objetivo principal** em *subobjetivos* e esses subobjetivos em *subsubobjetivos* e continuar até não conseguir mais decompor,
- Implementar atribuindo cada subobjetivo decomposto a uma classe, módulo, função ou qualquer que seja o nível mais baixo de granularidade que sua linguagem suporta.
- Os SOCs entram em ação *mantendo ao mínimo as conexões entre objetivos não relacionados*, de preferência nenhuma. Ao fazer a decomposição do design descobrirá que tem uma *meta de supervisor* que controla vários subobjetivos.

- Modelagem utilizando DFD

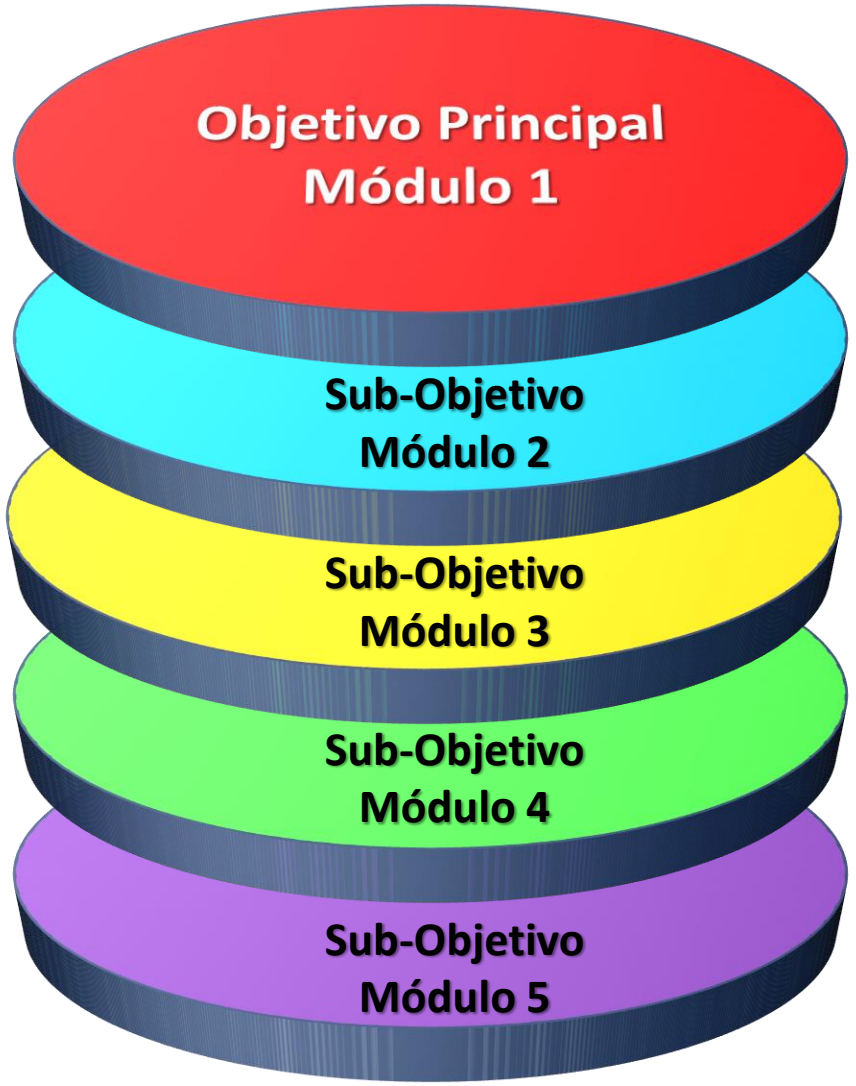


1.10 Camadas

O **princípio de camadas** (layering) é um conceito fundamental em *segurança de software* que divide a segurança em diferentes níveis ou camadas abstratas. Cada camada possui funções e responsabilidades específicas, e as camadas se comunicam entre si de maneira controlada.

Essa abordagem modular oferece diversas vantagens, como:

- **Maior modularidade e flexibilidade:** As camadas podem ser desenvolvidas, testadas e implantadas de forma independente, facilitando a manutenção e a atualização do sistema de segurança.
- **Melhor isolamento de falhas:** Se uma camada falhar, as outras camadas ainda podem funcionar, limitando o impacto da falha no sistema geral.
- **Maior defesa em profundidade:** A implementação de várias camadas de segurança torna mais difícil para um invasor penetrar no sistema, pois ele precisa superar cada camada individualmente.



1.10 Camadas

- **Camada de rede:** Essa camada protege a rede contra ataques, como invasões, interceptação de dados e falsificação de pacotes. Ela pode incluir firewalls, sistemas de detecção de intrusão (IDS) e sistemas de prevenção de intrusão (IPS).

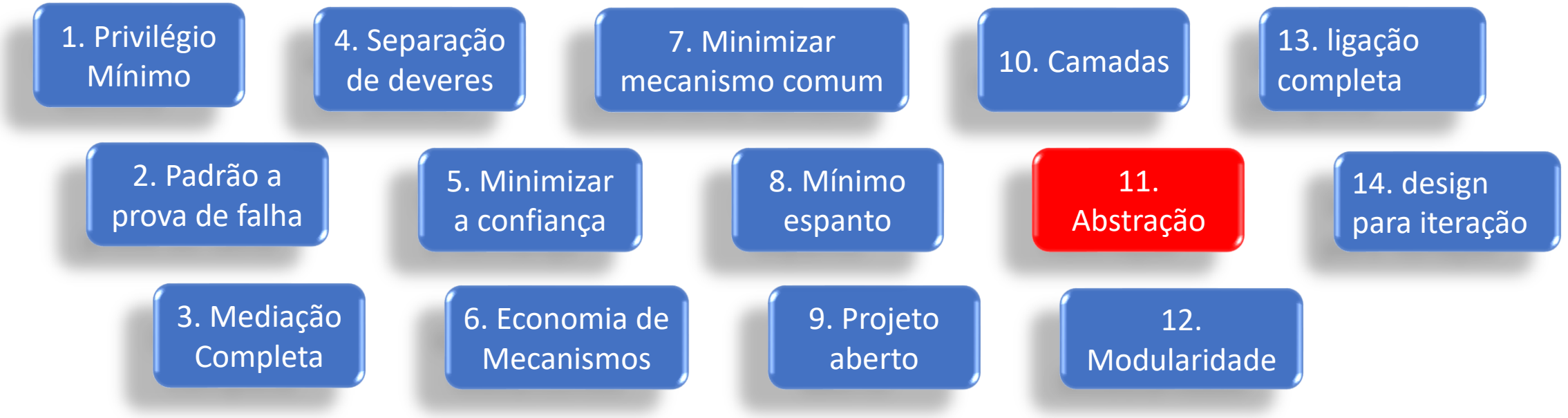
- **Camada de sistema operacional:** Essa camada protege o sistema operacional contra ataques, como malware, exploits e tentativas de acesso não autorizado. Ela pode incluir mecanismos de autenticação e autorização, controle de acesso baseado em função (RBAC), criptografia de disco e software antivírus.

- **Camada de aplicativo:** Essa camada protege os aplicativos contra ataques, como injeção de código, cross-site scripting (XSS) e ataques de força bruta. Ela pode incluir mecanismos de validação de entrada, filtragem de conteúdo, criptografia de dados e firewalls de aplicativos web (WAFs).

- **Camada de dados:** Essa camada protege os dados contra acesso não autorizado, modificação e exclusão. Ela pode incluir criptografia de dados, controle de acesso baseado em rótulo (LBAC) e mascaramento de dados.

EXEMPLOS

1. Princípios Fundamentais: 14 conceitos



1.11 Abstração

- A **abstração** é um conceito fundamental em ciência da computação que permite *ocultar os detalhes de implementação de baixo nível e fornecer uma interface de alto nível mais simples e fácil de usar.*
- Na *segurança de software*, a **abstração** é usada para ocultar os *detalhes complexos dos mecanismos de segurança* subjacentes, permitindo que os desenvolvedores e usuários se concentrem nas tarefas de segurança de alto nível.
- Dois tipos de abstrações:
  - *Abstração procedural*: sequência de instruções que possuem uma função específica e limitada
  - *Abstração de dados*: uma coleção nomeada de dados que descreve um objeto de dados

1.11 Abstração

Exemplos de abstração em segurança de software:

- **Funções de segurança:** fornecem uma *maneira simples de executar tarefas de segurança comuns*, como criptografar dados, verificar assinaturas digitais ou validar credenciais de usuário. As funções de segurança ocultam os *detalhes de implementação dos algoritmos* de segurança subjacentes, permitindo que os desenvolvedores os usem sem precisar entender os detalhes técnicos.

- **Bibliotecas de segurança:** fornecem uma *coleção de funções de segurança abstratas* que podem ser usadas para construir aplicativos seguros. As bibliotecas de segurança ocultam os *detalhes de implementação dos protocolos* de segurança subjacentes, permitindo que os desenvolvedores se concentrem na lógica de negócios de seus aplicativos.

- **Frameworks de segurança:** fornecem uma *estrutura para construir aplicativos* seguros. Os frameworks de segurança geralmente incluem bibliotecas de segurança, funções de segurança abstratas e padrões de design de segurança. Os frameworks de segurança podem ajudar os desenvolvedores a evitar erros comuns de segurança e construir aplicativos mais seguros com mais facilidade.

- **Modelos de segurança:** fornecem uma maneira abstrata de *descrever os requisitos de segurança* de um sistema. Os modelos de segurança podem ser usados para identificar, analisar e mitigar riscos de segurança.

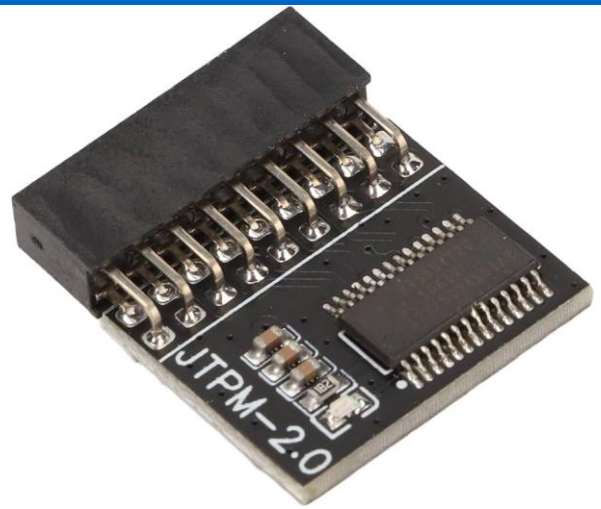
1.11 Abstração

Exemplos de como a abstração é usada em produtos de segurança de software:

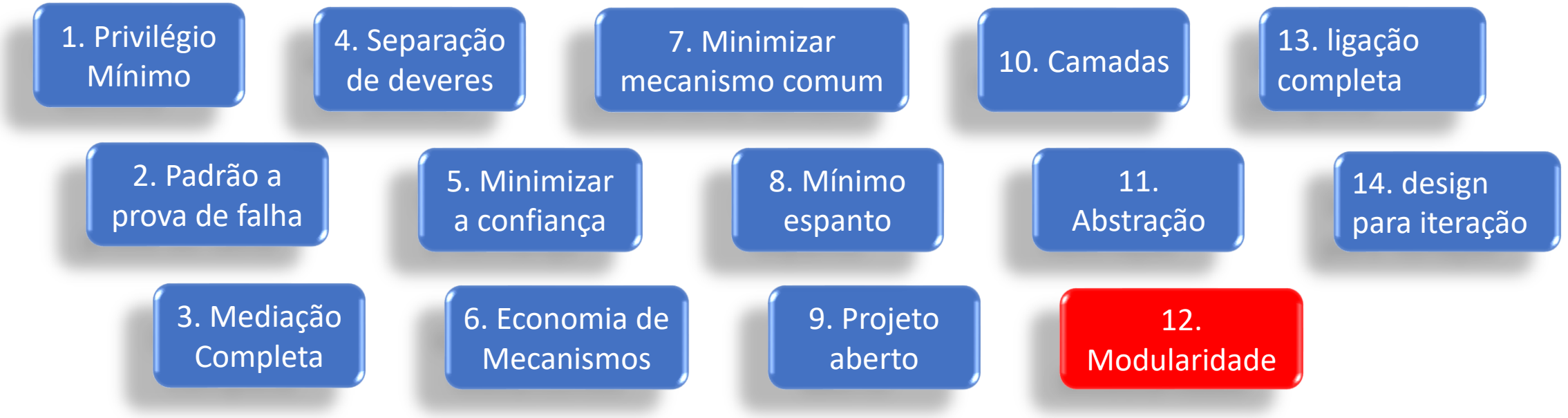
- **Sistemas operacionais:** fornecem uma *abstração de hardware*, permitindo que os aplicativos acessem os recursos de hardware sem precisar entender os detalhes de implementação do hardware. Essa abstração torna os sistemas operacionais mais fáceis de usar e programar.

- **Compiladores:** traduzem o código-fonte de alto nível em código de máquina de baixo nível. Os compiladores fornecem uma *abstração da arquitetura do computador*, permitindo que os programadores escrevam código em uma linguagem de alto nível sem precisar entender os detalhes da arquitetura do computador.

- **Bibliotecas de rede:** fornecem uma abstração dos protocolos de rede subjacentes, permitindo que os aplicativos se comuniquem entre si sem precisar entender os detalhes dos protocolos. Essa abstração torna mais fácil escrever aplicativos de rede.



1. Princípios Fundamentais: 14 conceitos



1.12 Modularidade

A **Modularidade** na *segurança de software* é um princípio fundamental que visa *dividir um sistema em módulos independentes e bem definidos*, com interfaces claras e bem documentadas.

Isso facilita o desenvolvimento, a manutenção, o teste e a implantação de medidas de segurança, além de aumentar a resiliência do sistema a falhas e ataques.

1.12 Modularidade - Exemplos

1. Arquitetura de Software:

- **Arquitetura orientada a serviços (SOA):** os componentes do sistema são implementados como serviços independentes que se comunicam através de interfaces bem definidas.
- **Arquitetura baseada em componentes:** o sistema é composto por componentes reutilizáveis que podem ser facilmente integrados e substituídos.
- **Design modular:** o sistema é dividido em módulos com responsabilidades bem definidas e interfaces claras.

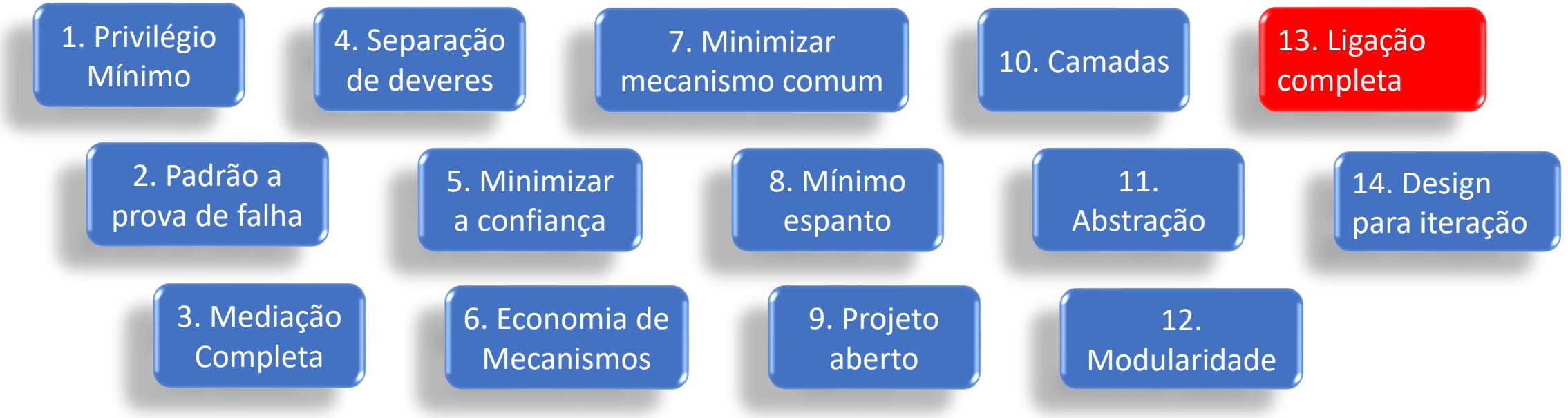
2. Desenvolvimento de Software:

- **Programação modular:** utilizar técnicas de programação modular para dividir o código em módulos com funções específicas e interfaces bem definidas.
- **Bibliotecas e frameworks:** utilizar bibliotecas e frameworks que fornecem módulos prontos para uso para diversas funcionalidades comuns.
- **Padronização de interfaces:** definir padrões para as interfaces dos módulos, garantindo interoperabilidade e facilitando a integração entre diferentes componentes.

3. Implementação de Segurança:

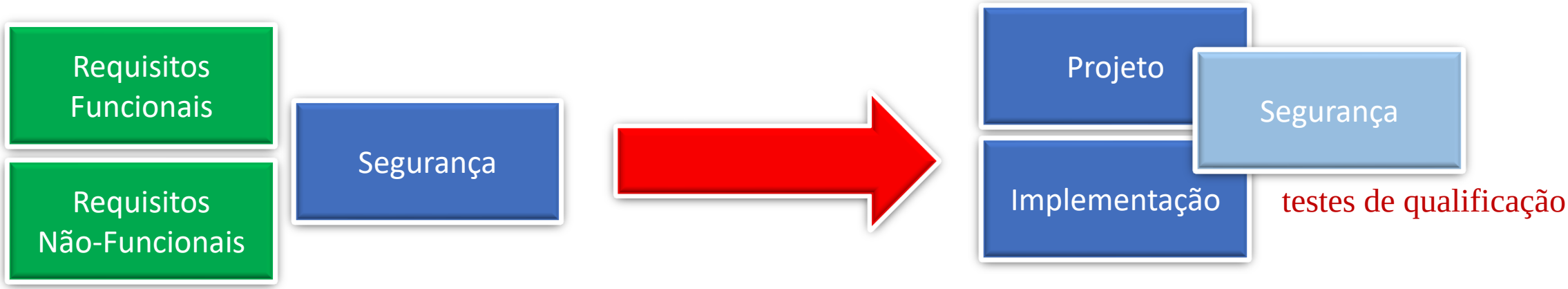
- **Módulos de autenticação e autorização:** implementar módulos específicos para autenticação e autorização de usuários, separando essas funcionalidades do restante do sistema.
- **Módulos de criptografia:** implementar módulos específicos para criptografia de dados, encapsulando as funções de criptografia e descriptografia.
- **Módulos de registro e auditoria:** implementar módulos específicos para registro de eventos de segurança e auditoria de atividades do sistema.

1. Princípios Fundamentais: 14 conceitos



1.13 Ligação Completa

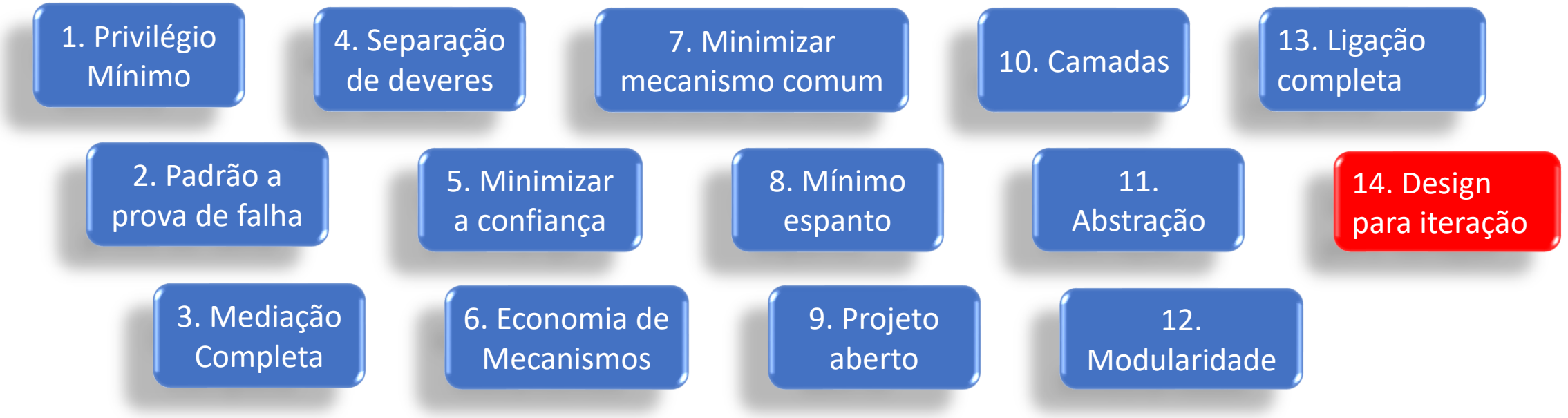
Adicionar *segurança* à combinação de *requisitos funcionais e não funcionais*, a mesma ligação deve existir para garantir que medidas de segurança adequadas tenham sido consideradas, projetadas e implementadas.



testes de qualificação

- **Primeiro**, o software atende aos requisitos especificados na SRS? O processo de resposta a esta pergunta é conhecido como **verificação**.
- **Segundo**, os requisitos e o software resultante atendem às necessidades de qualidade pretendidas da organização adquirente? O processo de resposta a esta pergunta é conhecido como **validação**.

1. Princípios Fundamentais: 14 conceitos

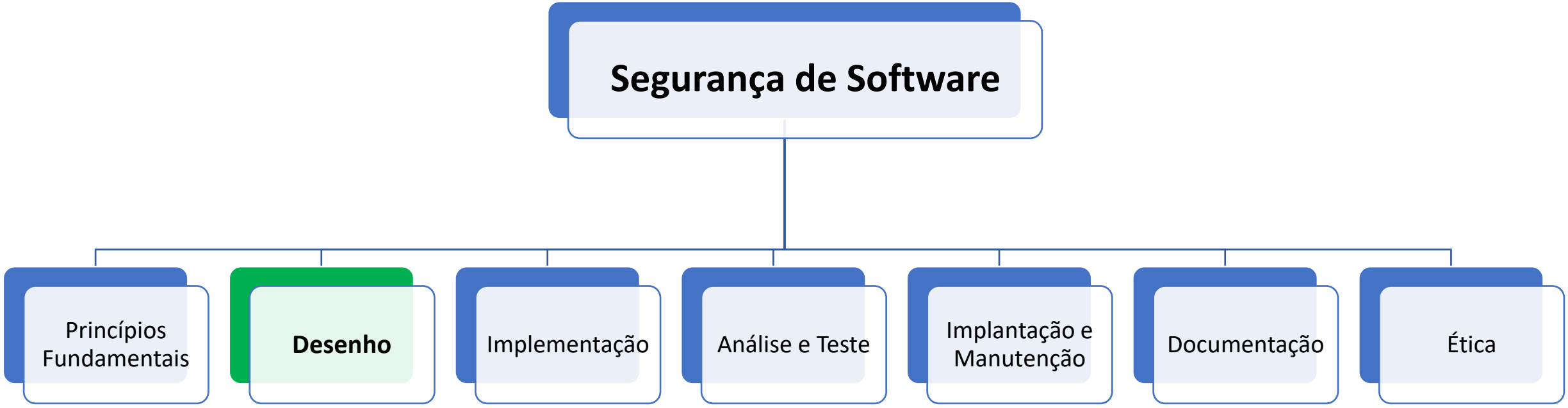


1.14 Design para Iteração

- Todos devem compreender que na engenharia de software as *mudanças no software são inevitáveis*. Essas mudanças podem ocorrer
  - através da regressão das abordagens de iteração para o desenvolvimento de software ou
  - através da fase de manutenção do ciclo de vida do software existente.
- Essas mudanças podem ser tão simples quanto modificar uma interface para simplificar seu uso ou tão complexas quanto corrigir funcionalidades importantes que fornecem interfaces para outros softwares ou mesmo sistemas de parceiros da cadeia de suprimentos.

- Independentemente do propósito e da extensão das alterações, a especificação do projeto para o investimento inicial do software e *cada iteração sucessiva* deve considerar que as alterações no software podem ter um *efeito dramático na segurança*. Com isso em mente, a especificação do projeto deve corresponder constantemente aos *requisitos de segurança do software*, a fim de fornecer consistência para a segurança do ambiente de software existente.

Segurança de Software: 7 unidades



Prof. Dr. Ausberto S. Castro Vera
Ciência da Computação
UENF-CCT-LCMAT
Campos, RJ

ascv@uenf.br
ausberto.castro@gmail.com