Programação Segura

Semana 7: Abuso de APIs, Características de Segurança

Profª Denise Goya
Denise.goya@ufabc.edu.br – UFABC - CMCC

Classes de Erros de Segurança

- Validação e representação dos dados de entrada
- Abuso de API
- Características de segurança
- Tempo e estado
- Tratamento de exceções
- Qualidade do código
- Encapsulamento
- Ambiente

Abuso de API

- Uma API é uma **interface** para acesso a uma biblioteca de funções (*Application Programming Interface*)
  - é um contrato entre quem chama e quem é chamado
  - Abuso de API é uma classe de erros em que esse contrato é quebrado

Casos Comuns em Abuso de APIs

- Funções perigosas: não usar (ex: **gets**, strcpy)
- Erros relacionados ao mal uso dos argumentos das funções (ex: **printf** sem string de formatação)
- Não verificação de valor de retorno das funções
- Exceção não capturada
- Más práticas relacionadas ao uso da J2EE
- Várias outras

Casos Comuns em Abuso de APIs

- Funções perigosas: não usar (ex: **gets**, strcpy)
- Erros relacionados ao mal uso dos argumentos das funções (ex: **printf** sem string de formatação)
- Não verificação de valor de retorno das funções
- Exceção não capturada
- Más práticas relacionadas ao uso da J2EE
- Várias outras

APIs: valor de retorno não verificado

- Ex: read()

java.io

Class FileInputStream

read

```
public int read(byte[] b)
         throws IOException
```

Reads up to `b.length` bytes of data from this input stream into an array of bytes. This method blocks until some input is available.

Overrides:

`read` in class `InputStream`

Parameters:

`b` - the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or `-1` if there is no more data because the end of the file has been reached.

APIs: valor de retorno não verificado

- Ex: read()

Example Language: **Java**

```
FileInputStream fis;
byte[] byteArray = new byte[1024];
for (Iterator i=users.iterator(); i.hasNext();) {
  String userName = (String) i.next();
  String pFileName = PFILE_ROOT + "/" + userName;
  FileInputStream fis = new FileInputStream(pFileName);
  fis.read(byteArray); // the file is always 1k bytes
  fis.close();
  processPFile(userName, byteArray);
```

Este código **não trata o retorno da função read()** e assume que o arquivo sempre terá 1KB de tamanho. Se um atacante puder forçar o uso de um arquivo com menos que 1KB de tamanho, lixo na memória (por exemplo, do último arquivo válido lido) pode ser reutilizado pelo atacante.

APIs: valor de retorno não verificado

- Ex: malloc

function
malloc <cstdlib>

```
void* malloc (size_t size);
```

Allocate memory block

Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block.

The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.

If *size* is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.

Return Value

On success, a pointer to the memory block allocated by the function.
The type of this pointer is always `void*`, which can be cast to the desired type of data pointer in order to be dereferenceable.
If the function failed to allocate the requested block of memory, a *null pointer* is returned.

APIs: valor de retorno não verificado

- Ex: malloc

Example Language: **C**

```
buf = (char*) malloc(req_size);
strncpy(buf, xfer, req_size);
```

Este código não trata o retorno da função malloc;

Se ocorrer erro na alocação, *buf* ficará **null** o que provocará falha no programa na instrução seguinte: ao tentar realizar a cópia de strings com strcpy

APIs: valor de retorno não verificado

- Ex: fgets

function

fgets <cstdio>

```
char * fgets ( char * str, int num, FILE * stream );
```

Get string from stream

Reads characters from *stream* and stores them as a C string into *str* until (*num*-1) characters have been read or either a newline or the *end-of-file* is reached, whichever happens first.

A newline character makes `fgets` stop reading, but it is considered a valid character by the function and included in the string copied to *str*.

A terminating null character is automatically appended after the characters copied to *str*.

Return Value

On success, the function returns *str*.
If the *end-of-file* is encountered while attempting to read a character, the *eof indicator* is set (feof). If this happens before any characters could be read, the pointer returned is a null pointer (and the contents of *str* remain unchanged).
If a read error occurs, the *error indicator* (ferror) is set and a null pointer is also returned (but the contents pointed by *str* may have changed).

APIs: valor de retorno não verificado

- Ex: fgets
- Se ocorre um erro de I/O ou é encontrado final-de-arquivo sem nenhum caractere lido:
  - fgets retorna null, sem escrever **nada** (nem o caractere finalizador de string) no destino

Return Value

On success, the function returns *str*.
If the *end-of-file* is encountered while attempting to read a character, the *eof indicator* is set (feof). If this happens before any characters could be read, the pointer returned is a null pointer (and the contents of *str* remain unchanged).
If a read error occurs, the *error indicator* (ferror) is set and a null pointer is also returned (but the contents pointed by *str* may have changed).

APIs: valor de retorno não verificado

- Ex: fgets
- Se ocorre um erro de I/O ou é encontrado final-de-arquivo sem nenhum caractere lido:
  - fgets retorna null, sem escrever **nada** (nem o caractere finalizador de string) no destino

Example Language: **C**

```
char buf[10], cp_buf[10];
fgets(buf, 10, stdin);
strcpy(cp_buf, buf);
```

Se isso ocorre neste exemplo, a variável *buf* não é preenchida e pode não conter o finalizador de string: potencial overflow na cópia (strcpy)

APIs: valor de retorno não verificado

- Ex: envolvendo memcpy

function

memcpy <cstring>

```
void * memcpy ( void * destination, const void * source, size_t num );
```

Copy block of memory

Copies the values of *num* bytes from the location pointed by *source* directly to the memory block pointed by *destination*.

The underlying type of the objects pointed by both the *source* and *destination* pointers are irrelevant for this function; The result is a binary copy of the data.

The function does not check for any terminating null character in *source* - it always copies exactly *num* bytes.

To avoid overflows, the size of the arrays pointed by both the *destination* and *source* parameters, shall be at least *num* bytes, and should not overlap (for overlapping memory blocks, memmove is a safer approach).

Parameters

num

Number of bytes to copy.
size_t is an unsigned integral type.

APIs: valor de retorno não verificado

- Ex: envolvendo memcpy

Example Language: **C**

```
int returnChunkSize(void *) {
  /* if chunk info is valid, return the size of usable memory,
   * else, return -1 to indicate an error
   */
  ...
}
int main() {
  ...
  memcpy(destBuf, srcBuf, (returnChunkSize(destBuf)-1));
  ...
}
```

A função *returnChunkSize* **retorna -1** em caso de bloco inválido. Como esta chamada não verifica o retorno da função, pode ser passado o valor **-2** à *memcpy*. Como o 3º parâmetro de *memcpy* é **sem sinal**, **-2** é interpretado como um inteiro com valor grande (MAXINT-1): overflow

APIs: exceção não capturada

- Ex: getByName()

getByName

```
public static InetAddress getByName(String host)
                            throws UnknownHostException
```

Determines the IP address of a host, given the host's name.

The host name can either be a machine name, such as "`java.sun.com`", or a textual representation of its IP address. If a literal IP address is supplied, only the validity of the address format is checked.

Throws:

`UnknownHostException` - if no IP address for the `host` could be found, or if a scope_id was specified for a global IPv6 address.
`SecurityException` - if a security manager exists and its checkConnect method doesn't allow the operation

APIs: exceção não capturada

- Ex: getByName()

Example Language: **Java**

```
protected void doPost (HttpServletRequest req, HttpServletResponse res) throws IOException {
  String ip = req.getRemoteAddr();
  InetAddress addr = InetAddress.getByName(ip);
  ...
  out.println("hello " + addr.getHostName());
}
```

Este código contém uma chamada ao método getByName, sem capturar exceções possíveis o que pode levar a um comportamento não previsto.

APIs: más práticas com J2EE

- Ex: gerenciamento de conexões:
  - em J2EE não se recomenda o gerenciamento **direto de conexões**: deve-se usar os recursos já implementados para esse tratamento
- Ex: uso direto de sockets
  - J2EE padrão permite o uso de sockets apenas com o propósito de se comunicar com aplicações legadas

APIs: más práticas com J2EE

- Ex: gerenciamento direto de conexões

Example Language: **Java** *(Bad Code)*

```
public class DatabaseConnection {
  private static final String CONNECT_STRING = "jdbc:mysql://localhost:3306/mysqldb";
  private Connection conn = null;

  public DatabaseConnection() {
  }

  public void openDatabaseConnection() {
    try {
      conn = DriverManager.getConnection(CONNECT_STRING
    } catch (SQLException ex) {...}
  }

  // Member functions for retrieving database connection and accessing database
  ...
}
```

(Good Code)

```
public void openDatabaseConnection() {
  try {
    InitialContext ctx = new InitialContext();
    DataSource datasource = (DataSource) ctx.lookup(DB_DATASRC_REF);
    conn = datasource.getConnection();

  } catch (NamingException ex) {...}
  } catch (SQLException ex) {...}
}
```

APIs: más práticas com J2EE

- Ex: gerenciamento direto de sockets

Example Language: **Java** *(Bad Code)*

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
  // Perform servlet tasks.
  ...

  // Open a socket to a remote server (bad).
  Socket sock = null;

  try {
    sock = new Socket(remoteHostname, 3000);

    // Do something with the socket.
    ...
  } catch (Exception e) {
    ...
  }
}
```

Abuso de API

- Outros exemplos (inclusive em outras linguagens de programação):

http://cwe.mitre.org/data/definitions/227.html

CARACTERÍSTICAS DE SEGURANÇA

Características de Segurança

- Implementar características de segurança em um software não implica que o está tornando seguro
- Erros nesta classe estão relacionados à má implementação de características de segurança para obtenção de:
  - Autenticação
  - Controle de acesso
  - Confidencialidade
  - Gestão de privilégios
  - Uso de criptografia

Características de Segurança

- Exemplos de problemas comuns:
  - Senhas literais no código (hardcoded) ou arquivos
  - Uso de gerador de números aleatórios fracos (valor gerado é previsível)
  - Uso de cifra de bloco fraca (DES ou 3DES)
  - Tamanhos de chaves pequenos e inseguros
  - Modo de operação fraco para cifra de bloco (ECB)
  - Cifra sem integridade
  - Uso de cifras no modo determinístico (RSA, AES, etc)
  - Vetor de inicialização estático
  - etc

Segurança: senhas literais no código

- Senhas nunca podem ser escritas no código:
  - Códigos são compartilhados dentro da empresa e são de fonte aberto
  - É possível se recuperar strings literais a partir de código executável ou de bytecodes
  - É preciso usar mecanismos de **armazenamento seguro de senha** (hash+sal)
  - Não pode senha em claro em arqs

```
private String SECRET_PASSWORD = "letMeIn!";

Properties props = new Properties();
props.put(Context.SECURITY_CREDENTIALS, "p@ssw0rd");
```

Segurança: gerador de nº aleatório

- Algoritmos geradores de números pseudo-aleatórios são **determinísticos** (dependem de um valor inicial: semente)
- Geradores **fracos** (como Random de java.util) podem ser usados para sorteios quaisquer, não para uso em criptografia (como parte do procedimento de geração de chaves secretas, token para cifra de fluxo, reset de senha enviado por email, etc)

Segurança: gerador de nº aleatório

- Deve-usar algoritmos mais fortes, como **SecureRandom** de java.security

```
String generateSecretToken() {
    Random r = new Random();
    return Long.toHexString(r.nextLong());
}
```

(Good Code)

```
import org.apache.commons.codec.binary.Hex;

String generateSecretToken() {
    SecureRandom secRandom = new SecureRandom();

    byte[] result = new byte[32];
    secRandom.nextBytes(result);
    return Hex.encodeHexString(result);
}
```

Segurança: cifra de bloco fraca

- A cifra de bloco **DES é muito fraca** para o poder computacional de hoje (na época em que foi criada e tornada padrão, era suficiente)
- **3DES** é uma variante de DES que aplica o algoritmo 3 vezes (deu uma sobrevida ao DES, mas ainda **é fraca para os padrões atuais**): em java DESede (DES encrypt-decrypt-encrypt)
- Padrão atual para cifra de bloco (Estados Unidos) é o AES (Advanced Encryption Standard)

Segurança: cifra de bloco fraca

```
Cipher c = Cipher.getInstance("DESede/ECB/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

(Good Code)

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

Segurança: tamanho de chave pequeno

- Cada algoritmo para cifra de bloco ou cifra de chave pública é acompanhado de uma série de **especificações para implementação segura**
- O tamanho da chave secreta é determinante do nível de segurança: valores abaixo do mínimo recomendável não é aceitável quando se implementa características de segurança
- Os algoritmos em geral são projetados para trabalharem com vários tamanhos de chave, inclusive para tamanhos que **hoje ou amanhã serão especificados como inseguros**

Segurança: tamanho de chave pequeno

- Exemplo com RSA, em Java:

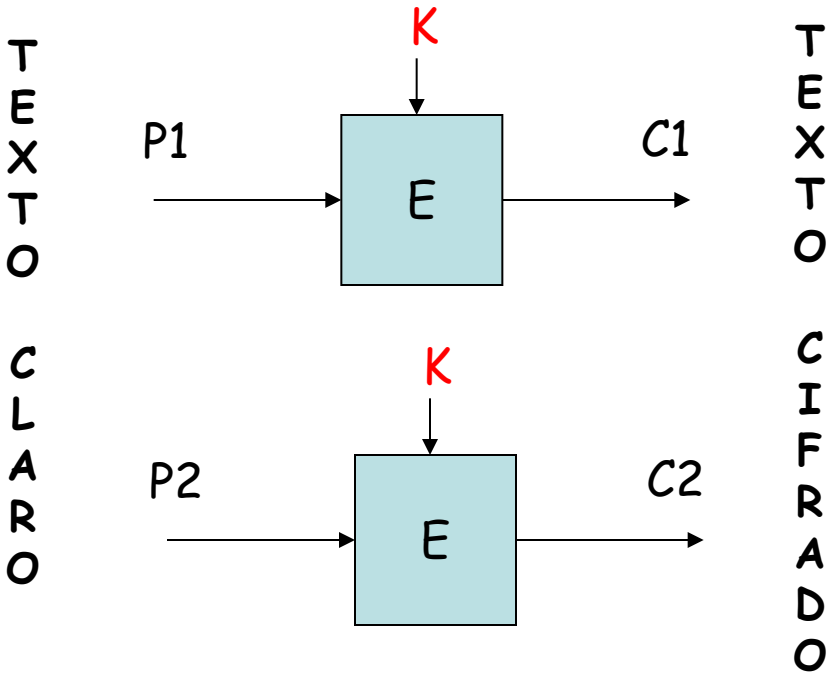
```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(512);
```

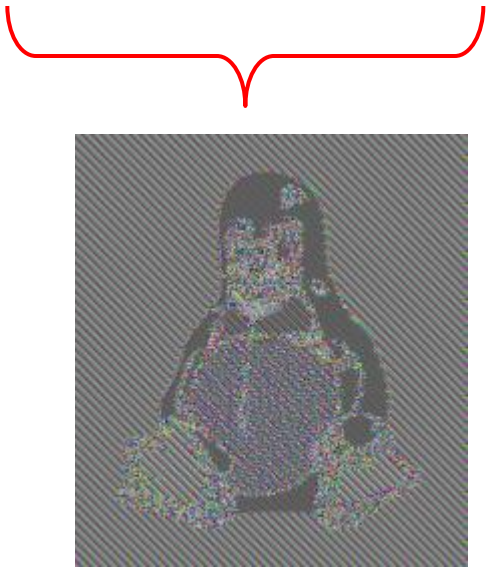
(Good Code)

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(2048);
```

Segurança: modo de operação

- Modo de operação **fraco** para cifra de bloco, na maioria das situações: ECB é vulnerável para arquivos grandes e estruturados (imagens, p.ex)





Segurança: modo de operação

- O modo ECB produz a mesma saída para blocos iguais, cifrados com a mesma senha: modo GCM é mais indicado

```
Cipher c = Cipher.getInstance("AES/ECB/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

(Good Code)

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

Segurança: modo de operação

- Mesmo o modo CBC (que é mais seguro que o ECB) pode sofrer ataque de **integridade**: recomendável o uso do modo GCM, que embute HMAC para detecção de adulteração na cifra

```
Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

(Good Code)

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

Segurança: cifras determinísticas

- O uso de cifras no modo determinístico é inseguro:
  - Cifras determinísticas produzem saídas iguais para entradas iguais (mesma mensagem e mesma chave): suscetíveis a ataques de repetição ou de cálculo da chave secreta a partir de grande coleta de dados cifrados com a mesma chave
- Grande parte dos algoritmos clássicos e padronizados são determinísticos, mas possuem versão não-determinística que deve ser usada

Segurança: cifras determinísticas

- Exemplos de cifras operando não-deterministicamente (seguras):
  - RSA no modo OAEP
  - Cifras de bloco com chave + sal (novo aleatório para cada cifra nova)

```
Cipher.getInstance("RSA/NONE/NoPadding")
```

(Good Code)

```
Cipher.getInstance("RSA/ECB/OAEPWithMD5AndMGF1Padding")
```

Segurança: vetor de inicialização (IV)

- Vetor de inicialização (IV) é um valor inicial que alguns algoritmos necessitam para iniciar a operação, como:
  - Cifras de fluxo (IV para a primeira iteração)
  - Modo de operação CBC (IV para o 1º bloco)
- O valor do vetor de inicialização **IV não pode ser estático**, pois enfraquece o algoritmo (em alguns casos, a previsibilidade da saída permite se recuperar dados secretos (chave ou texto confidencial)

Segurança: vetor de inicialização (IV)

- Recomendação: gerar novo IV sempre

```
private static byte[] IV = new byte[16] {(byte)0,(byte)1,(byte)2,[...]};

public void encrypt(String message) throws Exception {

    IvParameterSpec ivSpec = new IvParameterSpec(IV);
```

(Good Code)

```
public void encrypt(String message) throws Exception {

    byte[] iv = new byte[16];
    new SecureRandom().nextBytes(iv);

    IvParameterSpec ivSpec = new IvParameterSpec(iv);
```

Outras Características de Segurança

- Gestão de senhas deve ser planejada para ser segura:
  - Controlar requisitos mínimos para especificação de senhas e impedir uso de senhas fracas
  - Controlar mudança de senhas (de tempos em tempos é preciso alterá-las)
  - Mecanismo de recuperação de senha (esquecida) deve ser robusto

Outras Características de Segurança

- Privilégios e Permissões
  - Problemas de segurança podem surgir se forem atribuídos **privilégios inadequados** (ex: violação do princípio de privilégio mínimo) ou serem concedidas **permissões excessivas** de uso do sistema
  - Ex: instalar programas ou arquivos em diretórios que possam ser lidos ou escritos por todos; configurações *default* para concessão de permissões podem culminar em acesso não autorizado ou falsificação de endereço/identidade

Outras Características de Segurança

- Tratamento adequado a dados sigilosos ou de caráter privado:
  - Todo o sistema deve ser planejado para que não haja vazamento de dados críticos, como nº de cartão de crédito, dados de folha de pagamento, registros médicos, relações familiares/sociais, endereço de email, telefone, mensagens como SMS e email, etc
  - Se há política de privacidade sobre tais dados, é preciso garantir o não vazamento

Outras Características de Segurança

- Nunca invente seus próprios algoritmos criptográficos
  - use apenas os padronizados (que foram exaustivamente testados por especialistas) e sempre seguindo as especificações
  - princípio de Kerckhoffs (a força do algoritmo deve estar na sua chave)
- Extensa lista, por exemplo, em:

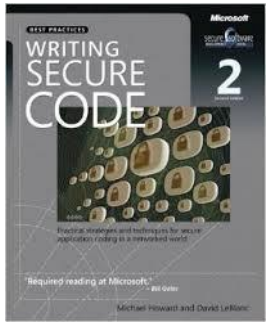
http://cwe.mitre.org/data/definitions/254.html

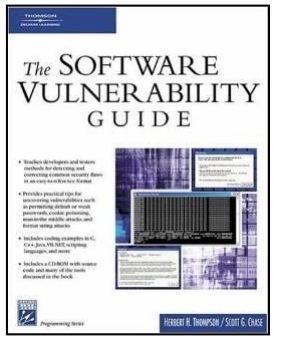
ESTUDO INDIVIDUAL

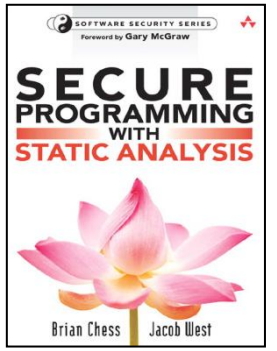
Leitura Recomendada

Além dos links indicados ao longo dos slides:

- Arteau. “**Bug Patterns**”, http://h3xstream.github.io/find-sec-bugs/bugs.htm
- HOWARD, M.; LEBLANC, D.: "**Writing Secure Code**" Microsoft Press, 2a edição, 2002.
- THOMPSON, H.; CHASE, SCOTT G.: "**The Software Vulnerability Guide**" Charles River Media, 2005
- WEELER, D.: "**Secure Programming for Linux and Unix HOWTO – Creating Secure Software**". Ebook disponível em http://www.dwheeler.com/secure-programs/
- CHESS, WEST. “**Secure Programming with Static Analysis**”. Addison-Wesley Professional, 2007.







Exercício

1) Em ambiente **Windows**, há uma série de recomendações para o desenvolvimento de software com proteção de dados secretos. Leia o capítulo 9 de

- HOWARD, M.; LEBLANC, D.: "**Writing Secure Code**" Microsoft Press, 2a edição, 2002.

para saber mais.