

Ciberseguridad

Grado en Ingeniería Informática

M5 - Desarrollo seguro de aplicaciones

Seguridad en la implementación

Oscar Delgado
oscar.delgado@uam.es

Álvaro Ortigosa (Coord.)
alvaro.ortigosa@uam.es

Programación Segura

C / C++

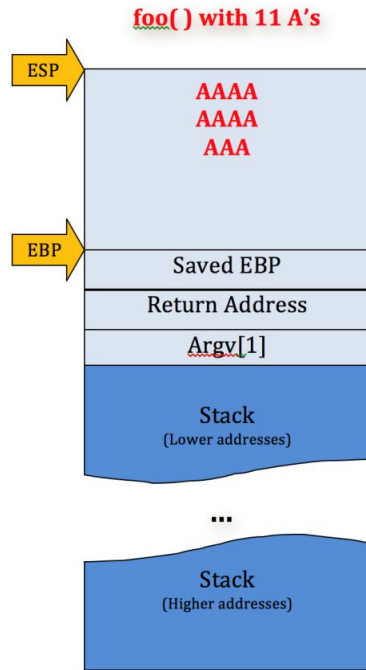
Test

```
#include <string.h>

void f(char* s)
{
    char buffer[10];
    strcpy(buffer, s);
}

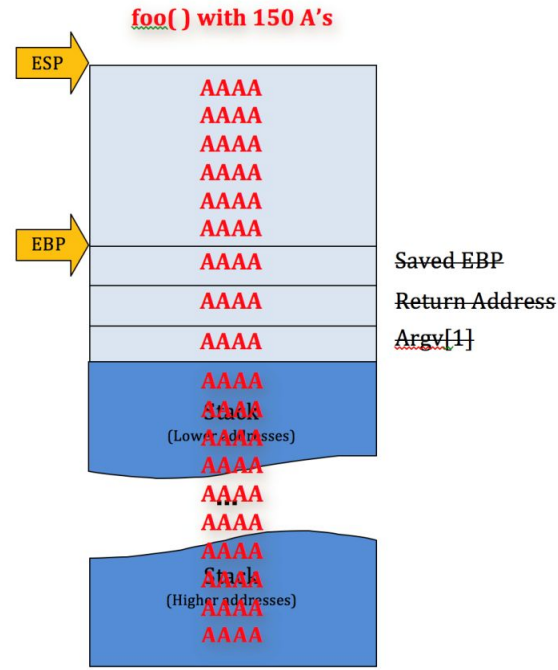
void main(void)
{
    f("01234567890123456789");
}
```

Desbordamiento de búfer



[6] CALL strcpy()

Copies value of Argv[1]
into space reserved for
local variable c



[6] CALL strcpy()

Copies value of Argv[1]
into space reserved for
local variable c

Contramedidas

- Recordar que los compiladores de lenguajes como C/C++ generan directamente el código indicado por el programador.
 - En C# o Java, por ejemplo, accesos incorrectos suelen derivar en excepciones en tiempo de ejecución.
 - En C++ pueden derivar en corrupción del contenido de la memoria.
- En lo posible, utilizar lenguajes fuertemente tipados, sin acceso directo a la memoria (C/C++ y ensamblador)
- Utilizar librerías de protección específicas (StackGuard)
- En general, validación de entrada.

Validación/depuración de entrada

- La entrada de datos, de cualquier fuente, es el mayor causante de fallos de software.
- Ninguna aplicación debería confiar en ningún dato de entrada que no haya sido validado.
- Validación: ¿buscar los patrones erróneos o los correctos?
 - En general, es preferible filtrar sólo los datos correctos, y rechazar cualquier otra cosa.
 - No intentar gestionar excepciones

Depurar las entradas

- Limpiar fuentes externas (x ej. entradas del usuario, scrapping de una web o consultas a bb.dd.).
- Hacerlo apenas los datos sean adquiridos:
 - Disminuye riesgo de manipulación de datos no depurados

Validación de entrada

```
bool IsBadExtension(char *szFilename) {
    bool fIsBad = false;
    if (szFilename) {
        size_t cFilename = strlen(szFilename);
        if (cFilename >= 3) {
            char *szBadExt[] = {".exe", ".com", ".bat", ".cmd"};
            char *szLCase = _strlwr(_strdup(szFilename));

            for (int i=0; i < sizeof(szBadExt)/sizeof(szBadExt[0]); i++){
                if (szLCase[cFilename-1] == szBadExt[i][3] &&
                    szLCase[cFilename-2] == szBadExt[i][2] &&
                    szLCase[cFilename-3] == szBadExt[i][1] &&
                    szLCase[cFilename-4] == szBadExt[i][0])
                    fIsBad = true;
            }
        }
        return fIsBad;
    }
}

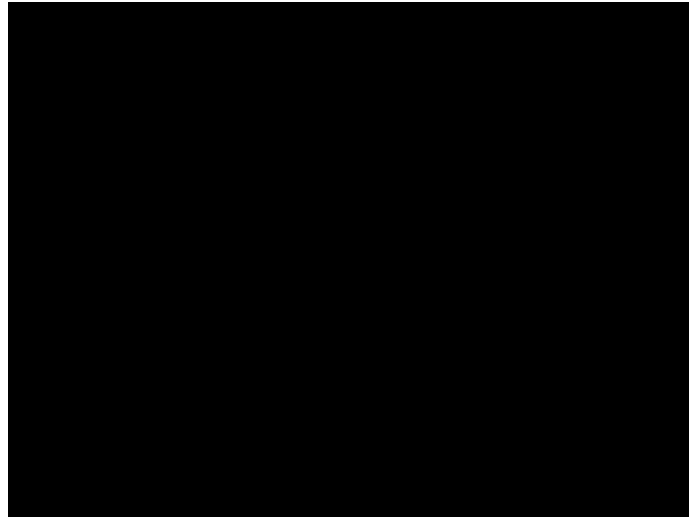
bool CheckFileExtension(char *szFilename) {
    if (!IsBadExtension(szFilename))
        if (UploadUserFile(szFilename))
            NotifyUserUploadOK(szFilename);
}
```


Uso correcto de las APIs

- No confiar en comportamiento sin documentar
 - El comportamiento real puede no ser el supuesto, o puede cambiar con el tiempo.
- No utilizar APIs que se sepan vulnerables
 - Por ejemplo strcpy, sprintf o system.
 - No es que sean siempre inseguras, pero un uso descuidado puede provocar abusos.
 - Sistemas de análisis estático de código ayudan a evitarlo.

Cuidado con overflows y underflows

```
1  #include <iostream>
2  int main()
3  {
4      for (size_t i = 5; i >= 0; --i)
5          std::cout << i << ' ';
6  }
```



Underflow!!

Cuidadosa gestión de errores y excepciones

- Anticipar posibles errores y forzar una gestión robusta.
- No filtrar información sensible: pilas de ejecución, volcados de BB.DD, códigos internos de error, ID o cualquier información personal de los usuarios.
- No permitir autenticación por el disparo de excepciones.
- No recoger los errores y excepciones sin gestionarlos (“tragarlos”); el sistema puede continuar operando en un estado inconsistente (y vulnerable)

No obsesionarse con pequeñas ganancias de eficiencia

- Si utilizamos C/C++ normalmente es porque se necesita eficiencia.
- En esos escenarios no es raro poner en peligro la seguridad para lograr mayor eficiencia.
 - “No hay razón para asumir que esta variable tendrá un valor no válido, así que quitemos la comprobación correspondiente”
 - Crear una variable sin inicializar, porque será inicializada más adelante.

Implementar correctamente la seguridad

- Los mecanismos de seguridad deben ser implementados de forma correcta.
 - Por ejemplo, no implementar seguridad por oscuridad. Con tiempo y dinero puede ser hackeado.
 - No tener contraseñas inmersas en el código.
- No implementar funciones criptográficas caseras, a menos que seas un investigador experto en el tema; utilizar cuidadosamente las existentes.
 - Por ejemplo, cuidado con la generación de números aleatorios.

Buenas prácticas

- Del mismo modo que se deben seguir guías de estilo y buenas prácticas para un buen código, se debería seguir guías para código seguro:
 - Por ejemplo, SEI Cert C++
 - <https://wiki.sei.cmu.edu/confluence/display/cplusplus/>
- Utilizar herramientas de soporte:
 - Prestar atención a los mensajes del compilador, incluyendo los warnings.
 - Utilizar las opciones del compilar. Por ejemplo GCC permite usar la bandera `-fsanitize`.
 - Utilizar herramientas de análisis estático de código

Programación Segura

Python

Principios de seguridad: Python

1. Depurar siempre las entradas.
2. Revisar el código.
3. Cuidado con la descarga de paquetes.
4. Revisar las licencias de las dependencias.
5. No usar la versión preinstalada.
6. Utilizar entornos virtuales.
7. Configurar `DEBUG=False` en producción.
8. Cuidado con el formateo de strings.
9. Mucha precaución al (de)serializar.
10. Utilizar las anotaciones de tipos.

Depurar las entradas

- Una vez más, es lo más importante para evitar ataques.
- En el caso particular de Python, algunas bibliotecas útiles:
 - Schema: “a library for validating Python data structures, such as those obtained from config-files, forms, external services or command-line parsing, converted from JSON/YAML (or something else) to Python data-types”

Ejemplo de uso de Schema

```
>>> from schema import Schema, And, Use, Optional, SchemaError

>>> schema = Schema([{'name': And(str, len),
...                     'age': And(Use(int), lambda n: 18 <= n <= 99),
...                     Optional('gender'): And(str, Use(str.lower),
...                                             lambda s: s in ('squid', 'kid'))}]]

>>> data = [{'name': 'Sue', 'age': '28', 'gender': 'Squid'},
...          {'name': 'Sam', 'age': '42'},
...          {'name': 'Sacha', 'age': '20', 'gender': 'KID'}]

>>> validated = schema.validate(data)

>>> assert validated == [{'name': 'Sue', 'age': 28, 'gender': 'squid'},
...                       {'name': 'Sam', 'age': 42},
...                       {'name': 'Sacha', 'age': 20, 'gender': 'kid'}]
```

Biblioteca bleach

- “An allowed-list-based HTML sanitizing library that escapes or strips markup and attributes.”

```
>>> import bleach

>>> bleach.clean('an <script>evil()</script> example')
u'an &lt;script&gt;evil()&lt;/script&gt; example'

>>> bleach.linkify('an http://example.com url')
u'an <a href="http://example.com" rel="nofollow">http://example.com</a> url'
```

Validando en Python

- Frameworks populares como Flask y Django incorporan sus propias herramientas.

`flask.escape()`

Replace the characters `&`, `<`, `>`, `'`, and `"` in the string with HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML.

If the object has an `__html__` method, it is called and the return value is assumed to already be safe for HTML.

Parameters: `s` – An object to be converted to a string and escaped.

Returns: A Markup string with the escaped text.

Validando con Django

- `mark_safe(string)` como herramienta básica

```
>>> mystr = "<b>Hello World</b>  "
>>> mystr = mark_safe(mystr)
>>> type(mystr)
<class 'django.utils.safestring.SafeString'>

>>> mystr = mystr.strip()  # removing whitespace
>>> type(mystr)
<type 'str'>
```

Validando con Django

- Como es un framework especializado provee herramientas específicas:

`escape(text)`

Returns the given text with ampersands, quotes and angle brackets encoded for use in HTML. The input is first coerced to a string and the output has **`mark_safe()`** applied.

```
format_html(  
    "{} <b>{}</b> {}",  
    mark_safe(some_html),  
    some_text,  
    some_other_text,  
)
```


Específico de dominios

- El problema de estas bibliotecas es que al ser especializadas no sirven para todo.
- Si quiero limpiar SQL:

```
# Instead of this ...
```

```
cursor.execute(f"SELECT admin FROM users WHERE username = '{username}'");
```

```
# ...do this...
```

```
cursor.execute("SELECT admin FROM users WHERE username = %(username)s", {'username': username});
```

O mejor (sqlalchemy):

```
query = session.query(User).filter(User.name.like('%{username}'))
```

Revisar el código

- Existen numerosas herramientas para hacer análisis estático de código con fines de seguridad, a distintos niveles.
- A nivel de estilo (linter), adherencia a PEP8.
 - Ejemplo: pep8, pylint, flake8.
- A nivel sintáctico:
 - Bandit: transforma código en árboles sintácticos para buscar problemas de seguridad.



Bandit

```
[ > bandit examples/yaml_load.py
[main] INFO    profile include tests: None
[main] INFO    profile exclude tests: None
[main] INFO    cli include tests: None
[main] INFO    cli exclude tests: None
[main] INFO    running on Python 3.8.2
Run started:2022-02-15 19:18:52.689821
```

Test results:

```
>> Issue: [B506:yaml_load] Use of unsafe yaml load. Allows instantiation of arbitrary objects. Consider yaml.safe_load().
Severity: Medium    Confidence: High
Location: examples/yaml_load.py:7:8
More Info: https://bandit.readthedocs.io/en/latest/plugins/b506\_yaml\_load.html

6      ystr = yaml.dump({'a': 1, 'b': 2, 'c': 3})
7      y = yaml.load(ystr)
8      yaml.dump(y)
```

Code scanned:

```
Total lines of code: 12
Total lines skipped (#nosec): 0
```

Run metrics:

```
Total issues (by severity):
    Undefined: 0.0
    Low: 0.0
    Medium: 1.0
    High: 0.0
Total issues (by confidence):
    Undefined: 0.0
    Low: 0.0
    Medium: 0.0
    High: 1.0
```

Files skipped (0):

```
> █
```

Descarga de paquetes

- Es muy fácil incluir bibliotecas, pero también introducir vulnerabilidades.
- El instalador estándar (pip) utiliza el *Python Pack Index (PyPI)*.
- No es realista esperar que los voluntarios que gestionan PyPI revisen los módulos que se incorporan → es razonable esperar que haya algunos módulos maliciosos

Prevención contra módulos maliciosos

- Un poco de investigación previa en el módulo.
- Vigilar que no haya *misspelling*

Python Package Index nukes 3,653 malicious libraries uploaded soon after security shortcoming highlighted

Unauthorized versions of CuPy and other projects flood PyPI

Prevención contra módulos maliciosos

- Confiar en terceras partes:

snykAdvisor

PyPI Search packages

All Packages Code Examples Categories Developer Tools Sign Up

Find the best package for your next project.

Search and compare over 1 million open source packages.

PyPI Search packages

Examples: pandas, numpy, django, flask, boto3, simplejson, ipython

Django v3.1.7

A high-level Python Web framework that encourages rapid development and clean, pragmatic design.

PyPI README GitHub Website BSD-3-Clause Latest version published 19 days ago

```
pip install django
```

Explore Similar Packages

Flask 99/100 Rails N/A react N/A

Package Health Score

97 / 100

POPULARITY KEY ECOSYSTEM PROJECT
MAINTENANCE HEALTHY
SECURITY NO KNOWN SECURITY ISSUES
COMMUNITY ACTIVE

Make sure the open source you're using is safe to use

SECURE MY PROJECT

Popularity KEY ECOSYSTEM PROJECT

Popularity by version

TOTAL WEEKLY DOWNLOADS (1,833,178)

| Version | Downloads |
|---------|-----------|
| 1.9.13 | ~50K |
| 2.1.7 | ~50K |
| 1.11.29 | ~50K |
| 3.1.5 | ~100K |
| 2.2.18 | ~100K |
| 3.1.6 | ~150K |
| 2.2.19 | ~100K |
| 3.1.7 | ~250K |

DEPENDENTS 9.57K GITHUB STARS 56.08K FORKS 24.04K CONTRIBUTORS 400

Maintenance HEALTHY

COMMIT FREQUENCY

| Version | Commits |
|---------|---------|
| 1.9.13 | ~50 |
| 2.1.7 | ~100 |
| 1.11.29 | ~150 |
| 3.1.5 | ~100 |
| 2.2.18 | ~100 |
| 3.1.6 | ~150 |
| 2.2.19 | ~100 |
| 3.1.7 | ~200 |

OPEN ISSUES 0 MERGED PR 6.31K OPEN PR 156 LAST COMMIT 18 hours ago

Revisar las licencias de las dependencias

- El software open source se puede usar libremente, pero pueden haber condiciones y restricciones.
 - Cómo se puede usar el software, si se pueden hacer cambios, etc.
- Es muy importante conocer los términos de las licencias de los paquetes que usamos para no entrar en conflictos legales.

No usar versión preinstalada del intérprete

- La mayoría de los sistemas POSIX incluye una versión del intérprete Python.
- Problema: no suele estar actualizada.
- Asegurarse de usar siempre la última versión disponible y mantenerla actualizada.
 - Puede ser un problema con proyectos en producción, debe implementarse un proceso de verificación antes de actualizar.

Utilizar entornos virtuales

- Un entorno virtual aísla la versión de intérprete, bibliotecas y scripts utilizados en el proyecto.
- La mayor parte de los IDEs, CLIs y paneles de control (como *Anaconda Navigator*) incorporan funciones para cambiar de un entorno virtual a otro.
- Conviene usar `pip freeze` o equivalente para registrar cambios en el entorno (evita problemas con bibliotecas)

Configurar `DEBUG=False` en producción





- Hay que prevenir cualquier filtración de información que pueda ser útil a un atacante.
- **Cuidado:** por omisión la mayoría de los frameworks tiene esto configurado a `True`.
- Es recomendable tener un sistema (script) de *deployment* que tenga automatizada esta verificación al pasar de desarrollo a producción.

Formateo de cadenas de texto

- Aunque un principio de Python es tener sólo una forma de hacer cada cosa, en realidad hay 4 formas distintas de dar formato a los strings.
- El formateo es cada vez más flexible y potente, pero con estas características también se incrementa el potencial para uso mal intencionado.
 - Los `fstrings` son muy flexibles y por lo tanto peligrosos.
 - El módulo `string` incluya una clase `Template` que es menos flexible con los tipos y no evalúa comandos Python → más seguro
- Por supuesto, especial precaución al formatear *inputs* del usuario.

Mucha precaución al (de)serializar

- Python provee mecanismo estándar para serializar/deserializar (Pickle).
- Se sabe que es **muy inseguro** y sólo debería ser usado con fuentes confiables y con mucha precaución.
- Por ejemplo YAML (alternativa a JSON): Biblioteca estándar PyYAML

| VULNERABILITY | | VULNERABLE VERSION |
|---|--------------------------|--------------------|
|  | Arbitrary Code Execution | [0,5.4) |
|  | Arbitrary Code Execution | [0,5.3.1) |
|  | Improper Access Control | [5.1,5.2) |
|  | Arbitrary Code Execution | (,4.2b1) |

ACTIVIDAD



Atacando la deserialización en Python

Como hemos visto, la serialización/deserialización de objetos debe ser realizada como mucho cuidado, y solo desde fuentes confiables.

Veamos cómo se lleva a cabo un ataque de este tipo. Imaginemos un código sencillo como el siguiente:

```
#!/usr/bin/env python3
import pickle, time

class Song:
    def __init__(self, title, length_in_seconds, singer):
        self.title = title
        self.length_in_seconds = length_in_seconds
        self.singer = singer

track1 = Song("Happy Birthday", "37", "Everyone")

# Exportamos los metadatos de la canción
pickle.dump(track1, open('track_file', 'wb'))
print(f"Exportada la canción [ {track1.title}]")

time.sleep(3)

# Los importamos de nuevo
loaded_track = pickle.load(open('track_file', 'rb'))
print(f"Importada la canción [ {loaded_track.title}]")
```

ACTIVIDAD



Atacando la deserialización en Python (II)

Un atacante genera ahora el siguiente código, que contiene una llamada al SO para listar los archivos del directorio actual

```
#!/usr/bin/env python3

import pickle, os

class SerializedPickle(object):
    def __reduce__(self):
        return(os.system, ("ls -la",))

pickle.dump(SerializedPickle(), open('malicious_pickle', 'wb'))
```


ACTIVIDAD



Atacando la deserialización en Python (III)

El ataque ya está montado. Si el atacante sustituye la canción serializada por su versión modificada, el player lo deserializará y ejecutará, sin querer, el código que contiene.

```
Exportada la canción [Happy Birthday]
total 20
drwxrwxr-x 2 oscar oscar 4096 abr 17 12:53 .
drwxrwxrwx 4 oscar oscar 4096 abr 17 12:36 ..
-rw-rw-r-- 1 oscar oscar  198 abr 17 12:51 evil_pickle.py
-rw-rw-r-- 1 oscar oscar  554 abr 17 12:51 music_player.py
-rw-rw-r-- 1 oscar oscar   44 abr 17 12:53 track_file
Traceback (most recent call last):

File "/[...]/music_player.py", line 20, in <module>
    print(f"Importada la canción [{loaded_track.title}])")
                                   ^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'int' object has no attribute 'title'
```

Posible defensa

- PyYAML provee 2 métodos: `yalm.Loader()` y `yalm.SafeLoader()`.
- El segundo impide la carga de clases personalizadas, pero sí permite tipos estándar como hashes y arrays.
-
- Un caso similar es la decodificación XML.
 - Cuidado con los ataques de DOS o expansión de entidades externas.

Adicional: usar las anotaciones de tipos

- Introducidas a partir de 3.5, no son forzadas por el intérprete → uso limitado para seguridad.
- Sin embargo, sí pueden ser de utilidad para verificadores estáticos:

```
1  MODE = Literal['r', 'rb', 'w', 'wb']
2  def open_helper(file: str, mode: MODE) -> str:
3      ...
4  open_helper('/some/path', 'r') # Passes type check
5  open_helper('/other/path', 'typo') # Error in type checker
```

Programación Segura

Java

Reglas seguridad

1. Visibilidad de funciones y atributos

No utilices funciones o atributos públicos, a menos que tengas una buena razón. Decláralos como privados y proporciona acceso a ellos solo a través de *getters* y *setters*. Todo lo que pueda ser privado, hazlo privado!

2. Atributos estáticos

Evita utilizar atributos estáticos, porque “pertenecen” a la clase (y no a los objetos) y pueden ser accedidos por cualquier otra clase.

Reglas seguridad

3. Evita la serialización

Oracle tiene planes para eliminar la serialización del lenguaje Java en el medio plazo. Mark Reinhold, arquitecto jefe de Java en Oracle, ha declarado que un tercio o más de todas las vulnerabilidades de Java están relacionadas con la serialización.

Soluciones:

- Utilizar formatos no nativos, como JSON.
 - Ojo, se aplica lo dicho para Python
- Restringir la serialización desde el código

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

Reglas seguridad

4. No almacenes secretos

- No guardes secretos (claves criptográficas, contraseñas) en el código o en los datos, porque pueden ser obtenidos fácilmente desensamblando el código.
- La ofuscación tampoco ayuda demasiado
- Si hay que almacenar contraseñas, no guardes el texto plano sino el hash:
 - Preferiblemente usando un *salted hash* y un algoritmo recomendado, como SHA-2.

Reglas seguridad

5. Utilizar sólo bibliotecas confiables y probadas

- Al igual que en Python, las bibliotecas públicas pueden contener vulnerabilidades o incluso código malicioso.
- Utilizar bibliotecas probadas
- Mantenerlas actualizadas
- Comprobar periódicamente si se han reportado vulnerabilidades o si necesitan algún arreglo (fix) de seguridad.

Reglas seguridad

6. Filtrar información sensible

- Es importante recordar que información sensible, que puede ser utilizada para un ataque, se puede fugar a través del texto del mensaje o del propio tipo de una excepción.
- Por ejemplo el mensaje `FileNotFoundException`. Estos mensajes pueden contener información sobre la organización del sistema de archivos y el tipo de excepción revelar el tipo de fichero faltante.
- De la misma forma, es más seguro utilizar ventanas de error genéricas para los usuario, para no dar demasiados detalles.

Reglas seguridad

7. No registrar (log) información sensible

- En línea con lo anterior, tampoco es conveniente que información sensible quede registrada en el propio sistema y que pueda ser objeto de un robo de datos.
- Por ejemplo, números de tarjetas de crédito y débito, cuentas bancarias, pasaportes y claves son muy sensibles y valiosos para los criminales.
- Si fuera necesario, pensar en grabar por ejemplo sólo los últimos 4 dígitos, y hacer de forma cifrada con una biblioteca probada
 - ¡No escribas tus propias funciones de cifrado!
-

Reglas seguridad

8. Escribe código simple

- En general, cuando más simple el código más seguro.
 - Esto es cierto no sólo para Java.
 - Y esto es bueno no sólo por cuestiones de seguridad.
- Utilizar verificadores de código; las vulnerabilidades son más fáciles de eliminar en fases iniciales, y no cuando el software está en producción.
- Encapsular el código tanto como sea posible. Ocultar los detalles de implementación es bueno tanto para la seguridad como para la mantenibilidad del código.
- API e interfaces tan pequeñas como sea posible.

Reglas seguridad

9. Prevenir las inyecciones de código

- Limpiar las entradas (ya lo hemos dicho, pero nunca será suficiente).
 - Validación con listas blancas.
 - Escapar los caracteres reservados.
- SQL: usar sentencias preparadas y procedimientos almacenados.
- En aplicaciones Web, limpiar las entradas tanto del lado servidor como del lado cliente.