INTRODUCCIÓN

¿Por qué las puertas de acceso a una vivienda tienen las bisagras hacia adentro?

¿Qué notáis de raro en esta imagen?

¿Por qué las puertas de acceso a una vivienda tienen las bisagras **hacia adentro**?

- El pestillo asociado al pomo de una puerta es una medida de seguridad
- Las bisagras no son una medida de seguridad, pero están implicadas en la seguridad

medidas de seguridad $\neq$ medidas seguras

En el **software**, este tipo de medidas no se tiene siempre en cuenta...

Ejemplo: extracto de la documentación de BEA WebLogic (2004):

> Since most security for Web applications can be implemented by a system administrator, application developers *need not pay attention to the details of securing the application unless there are special considerations that must be addressed in the code*. For programming custom security into an application, WebLogic Server application developers can take advantage of BEA-supplied Application Programming Interfaces (APIs) for obtaining information about subjects and principals (identifying information for users) that are used by WebLogic Server. The APIs are found in the `weblogic.security` package.

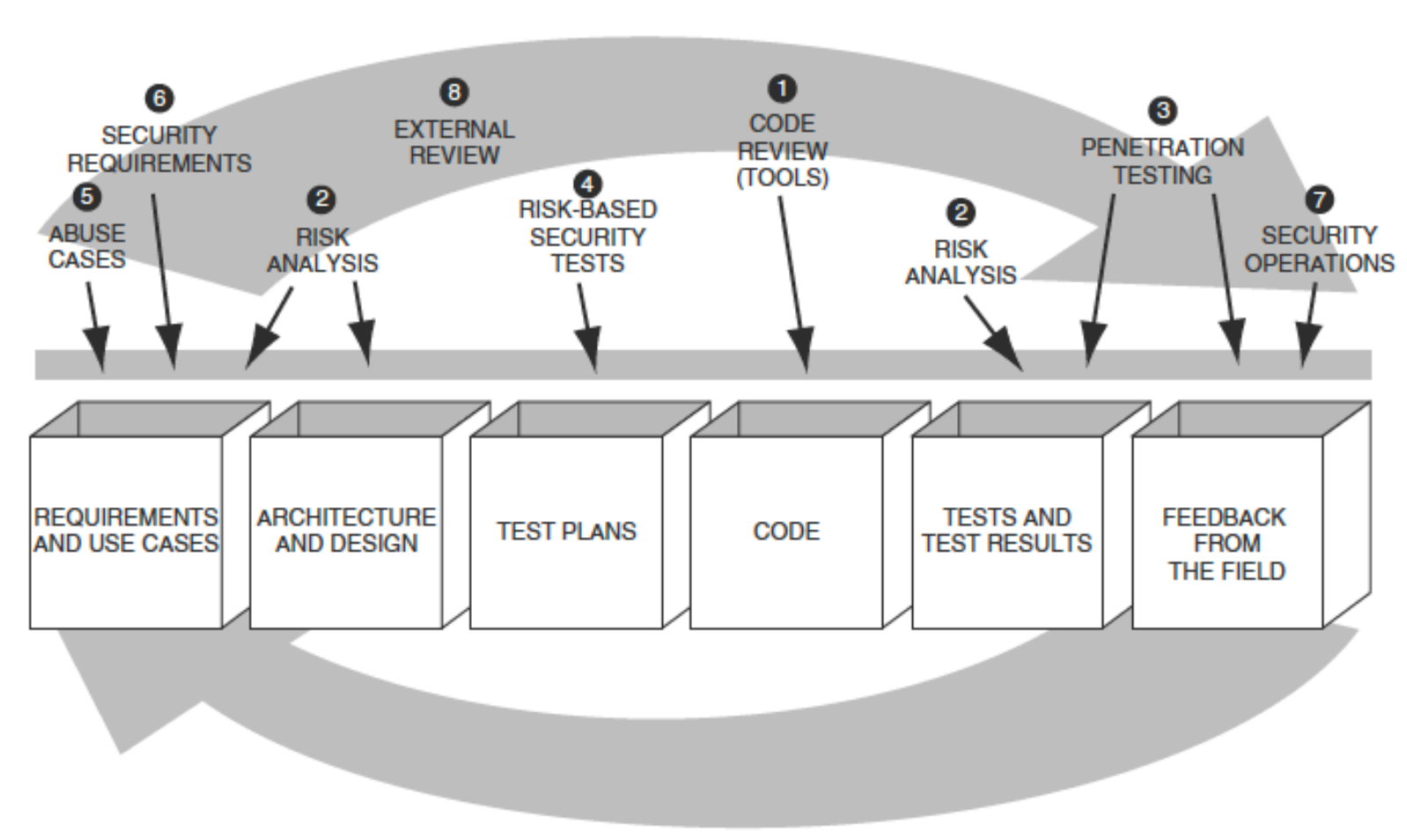
Muchos sistemas de software no se **construyen** con la seguridad en mente.

Las tecnologías **reactivas** como los cortafuegos pueden aliviar algunos ataques a los servidores, pero no abordan el problema principal: **software mal hecho**

Para poner en práctica la seguridad del software hay que:

- Adoptar prácticas de ingeniería
- Considerar la seguridad como parte del **Software Development Lifecycle** (SDL)

Software security points in the SDL



¿Cuáles son los puntos más importantes de cara a la seguridad?

Esta asignatura trata sobre la seguridad en los siguientes puntos...

- Arquitectura y diseño
- Construcción (no sólo implementación/código)
- Operaciones

CÓDIGO Y DISEÑO

Ejemplo: Entrada no validada

Vulnerabilidad provocada por no validar la entrada del usuario:

Implementación en Python:

```
def greet_user(name):
    print(f"Hola {name}!")

# Supongamos que la entrada proviene de un parámetro web
user_input = "<script>alert('XSS')</script>"
greet_user(user_input)
```

Solución más segura:

```
import html

def greet_user(name):
    safe_name = html.escape(name)
    print(f"Hola {safe_name}!")
```

Ejemplo: Buffer overflow

Provocado por ofrecer a los atacantes la posibilidad de escribir arbitrariamente en memoria.

Implementación en C:

```c
void trouble() {
  int a = 32; /* integer */
  char line[128]; /* character array */
  gets(line); /* read a line from stdin */
}
```

Ejemplo: Buffer overflow

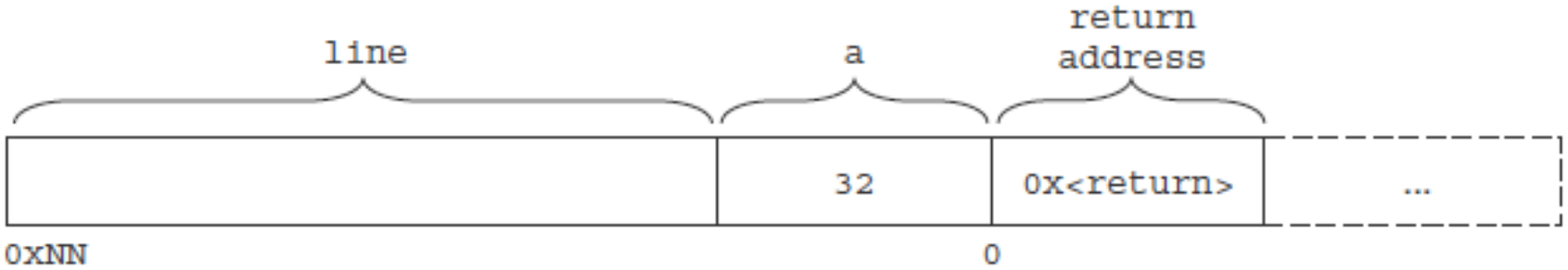
Solución más segura:

```
void trouble() {
  int a = 32; /* integer */
  char line[128]; /* character array */
  fgets(line, sizeof(line), stdin);  /* read a line from stdin */
}
```

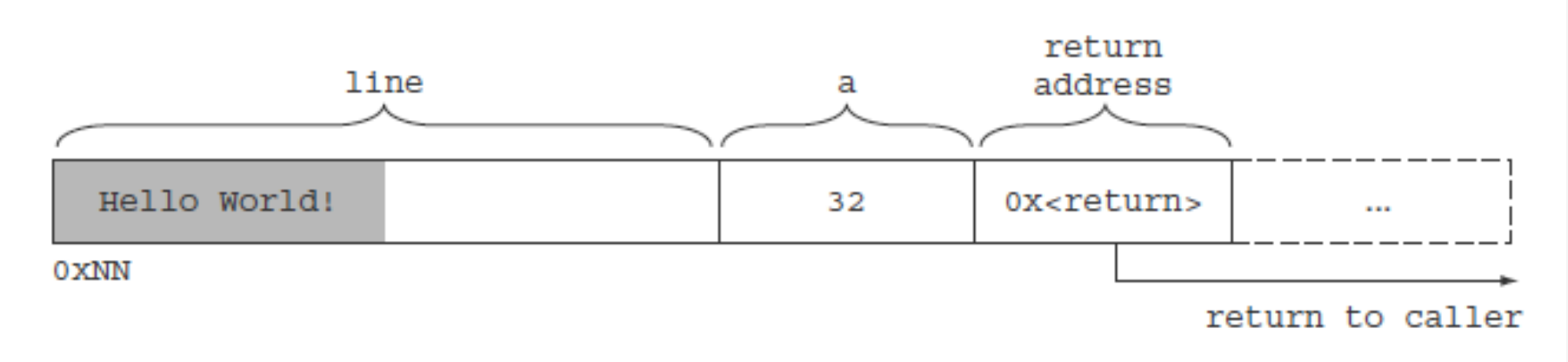
Estructura en memoria:

- `0xNN` es la dirección de comienzo de la pila (*stack*)
- Marco de pila tras la llamada a `trouble()`...

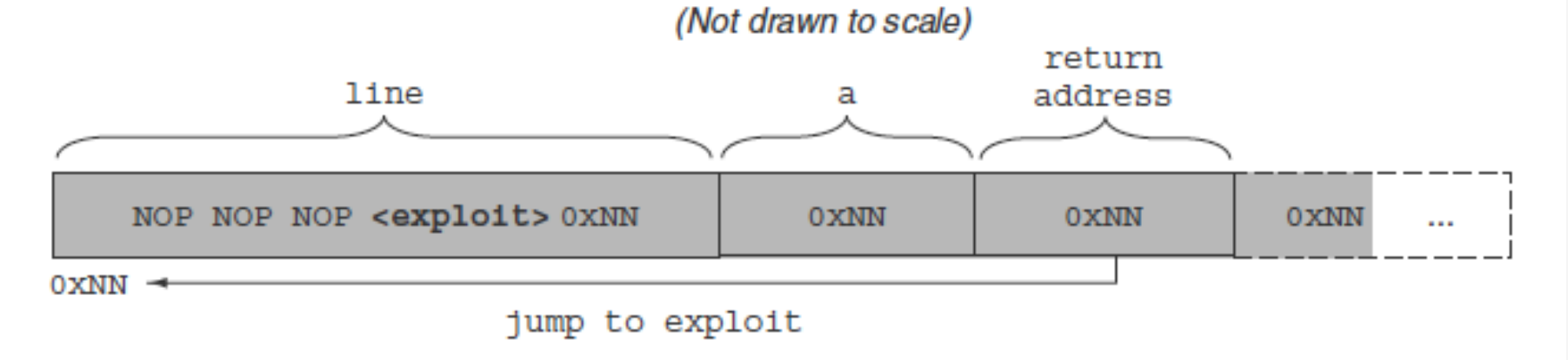
Marco de pila **antes de la ejecución:**



Marco de pila **después de ejecución normal:**



Marco de pila **tras un ataque por buffer overflow:**



Implementation bugs

¿A qué se debe este *buffer overflow*?

Buffer overflow provocado por el (mal) uso de APIs vulnerables:

`gets()`, `strcpy()`, etc.

¿Cómo evitar el **mal uso** de APIs vulnerables?

- Code review (manual)
- Análisis estático (automatizado):
  - **Static Application Security Testing (SAST)**
- Análisis de dependencias (integrado en el SDLC):
  - **Software Composition Analysis (SCA)**
- Análisis dinámico
  - **Dynamic Application Security Testing (DAST)**

¿Todos los problemas de seguridad se pueden resolver con análisis de código?

Menos del 50%

— según Chest & West: *Secure Programming with Static Analysis*, 2007 —

¿Qué otras actividades relacionadas con el desarrollo tienen implicaciones en la seguridad?

Arquitectura y diseño

Ejemplo: Arquitectura insegura

Un servicio web que ejecuta consultas SQL directamente desde parámetros HTTP:

Implementación en Python:

```
@app.route('/user')
def user_info():
    name = request.args['name']
    query = f"SELECT * FROM users WHERE name='{name}'"
    return db.execute(query)
```

Principios de diseño seguro

1. **Defensa en profundidad**: varias capas de seguridad.
2. **Mínimo privilegio**: cada módulo o usuario solo debe tener los permisos mínimos necesarios.
3. **Separación de responsabilidades**: la lógica de negocio no debe ejecutar tareas del sistema.
4. ***Fail safe defaults***: por defecto, negar acceso o fallar de forma controlada.

¿Por qué el **diseño** es clave para la seguridad?

El diseño asegura la **mantenibilidad**

Mantenibilidad

- Hallar *code smells* o problemas de mantenibilidad
- *Refactoring*: cómo cambiar el diseño/implementación sin cambiar la funcionalidad
- *Linting*: verificar la calidad y estilo del código
- *Coding standards*: **Common Weakness Enumeration (CWE)**, CERT, CVE, NVD, ...

> Ann Campbell: For secure code, maintainability matters

Coding standards

El 60% de las CWE no son exactamente vulnerabilidades...

La vista CWE-699: Software Development contiene debilidades de categorías diversas...

- Problemas de complejidad
- Errores numéricos
- Malas prácticas de codificación

Malas prácticas de codificación

- CWE-478: Missing default case
- CWE-581: Object Model Violation: Just One of Equals and Hashcode Defined

Tendencias 2019-2024

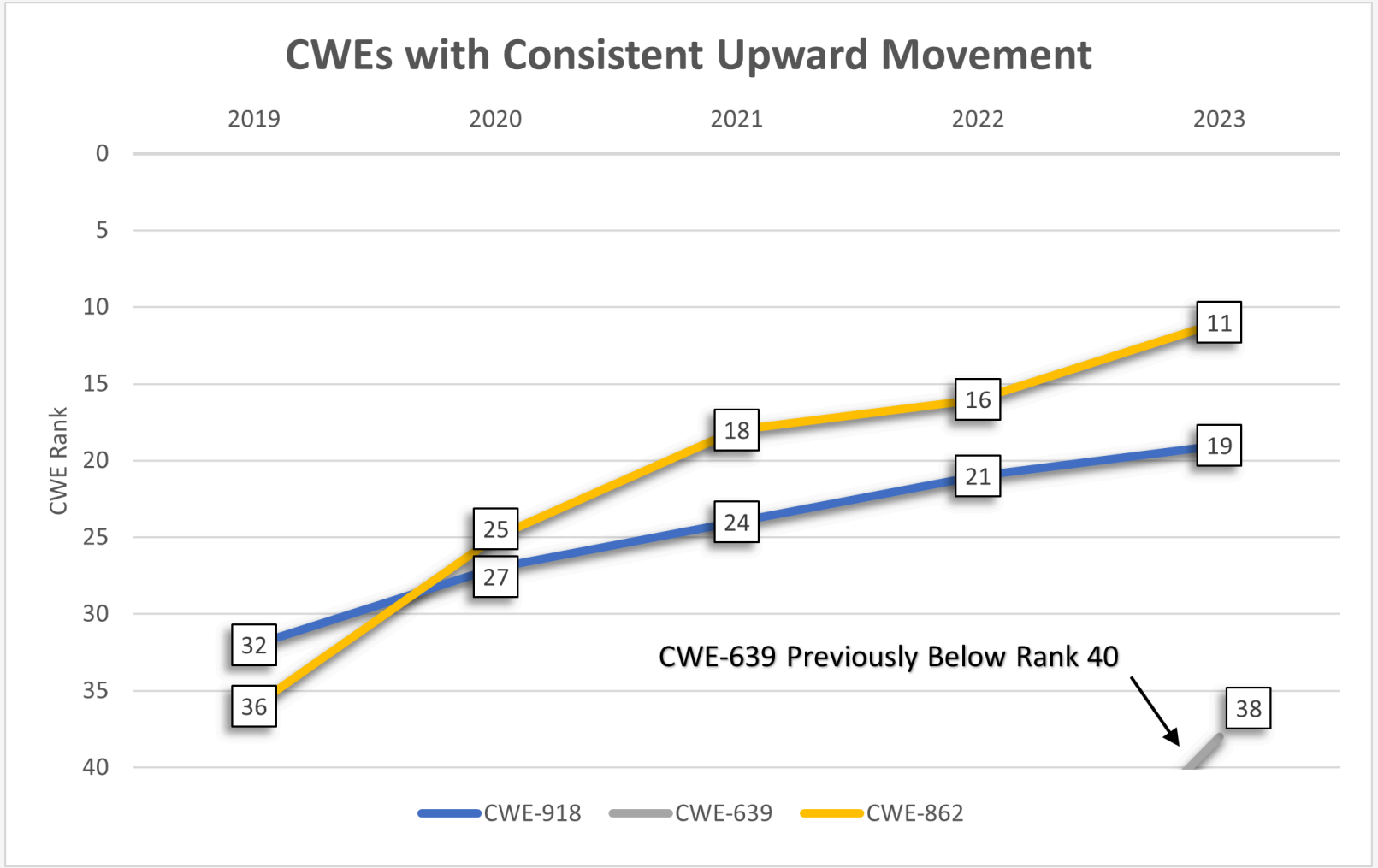
CWE Top 25 Most Dangerous Software Weaknesses

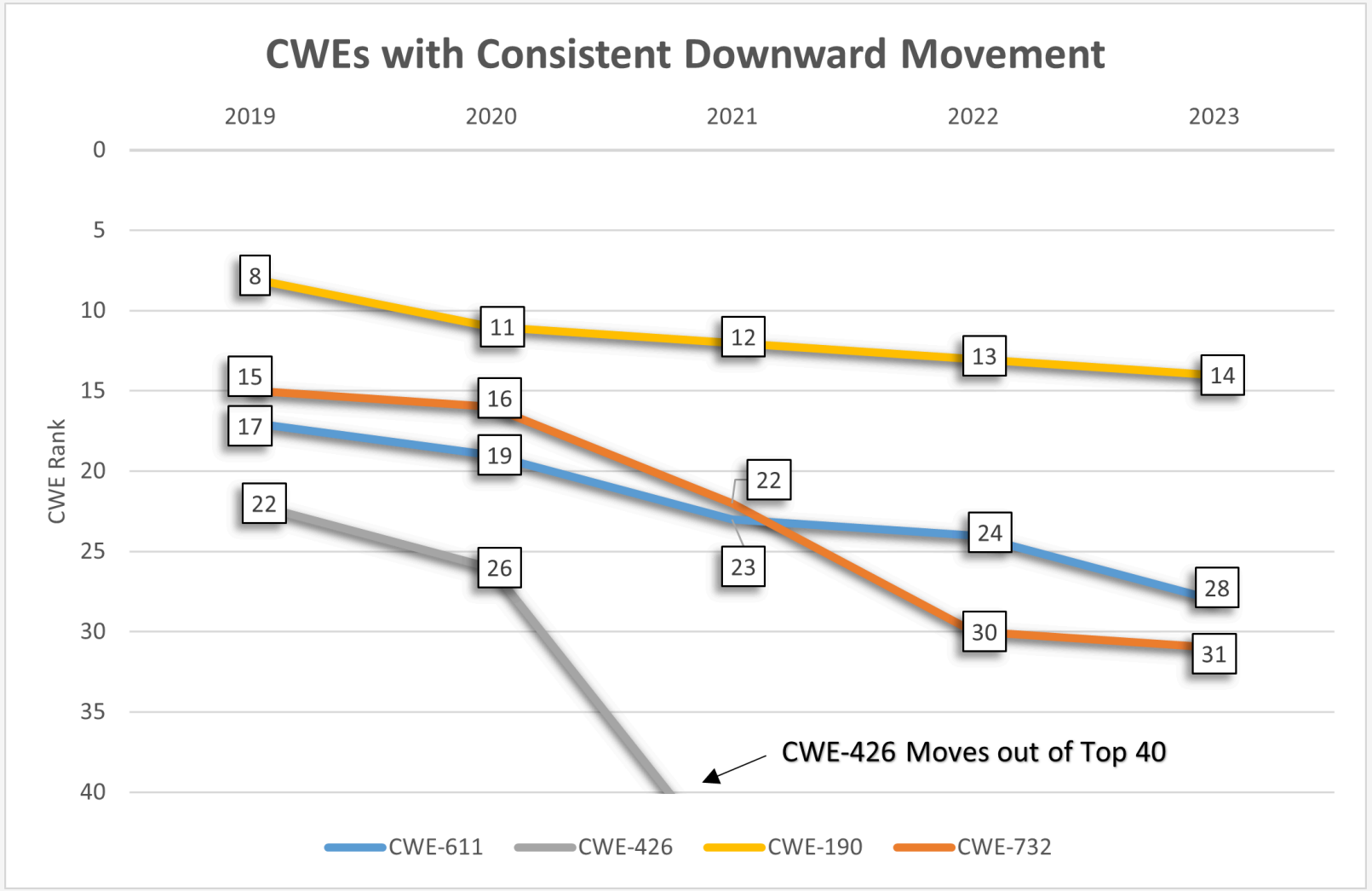
Al alza:

- CWE-862: Missing Authorization
- CWE-918: Server-Side Request Forgery (SSRF)
- CWE-639: Authorization Bypass Through User-Controlled Key

A la baja:

- CWE-190: Integer Overflow or Wraparound
- CWE-732: Incorrect Permission Assignment for Critical Resource
- CWE-611: Improper Restriction of XML External Entity Reference
- CWE-426: Untrusted Search Path


CWEs with Consistent Upward Movement
2019
2020
2021
2022
2023
0
5
10
15
20
25
30
35
40
CWE Rank
32
36
27
25
24
18
21
16
19
11
38
CWE-639 Previously Below Rank 40
CWE-918
CWE-639
CWE-862


CWEs with Consistent Downward Movement
2019
2020
2021
2022
2023
0
5
10
15
20
25
30
35
40
CWE Rank
8
11
12
13
14
15
16
17
19
22
24
28
22
26
23
30
31
CWE-426 Moves out of Top 40
CWE-611
CWE-426
CWE-190
CWE-732

¿Qué requisito de seguridad puede verse incumplido si no se cumple la CWE-581?

Disponibilidad

ARQUITECTURA

> Robert C. Martin: La arquitectura de un sistema software es la forma dada al sistema por aquellos que lo construyen. La forma es la división del sistema en componentes, la disposición de esos componentes y la manera en que se comunican entre sí.

Para un enfoque completo a la seguridad del software hay que combinar revisiones de código y análisis arquitectónico

¿Qué tiene que ver la **arquitectura** en todo esto de la seguridad?

La **arquitectura software** decide los componentes que conforman un sistema software y sus interacciones

¿Son importantes otras actividades relacionadas con el desarrollo de software?

Mucho, pero no las trataremos aquí

No nos interesan (tanto)...

- Técnicas criptográficas avanzadas
- Gestión de identidades
- Control de acceso
- Etc.

Sí nos interesan aquellos aspectos del desarrollo de software que, en principio, no parecen relacionados con la seguridad, pero que la ponen en riesgo si no se hacen bien.

¿Así que hay que saber **programación** y técnicas de **desarrollo de software** para poder garantizar la seguridad?

¿No hay un respiro para quienes no les gusta programar?

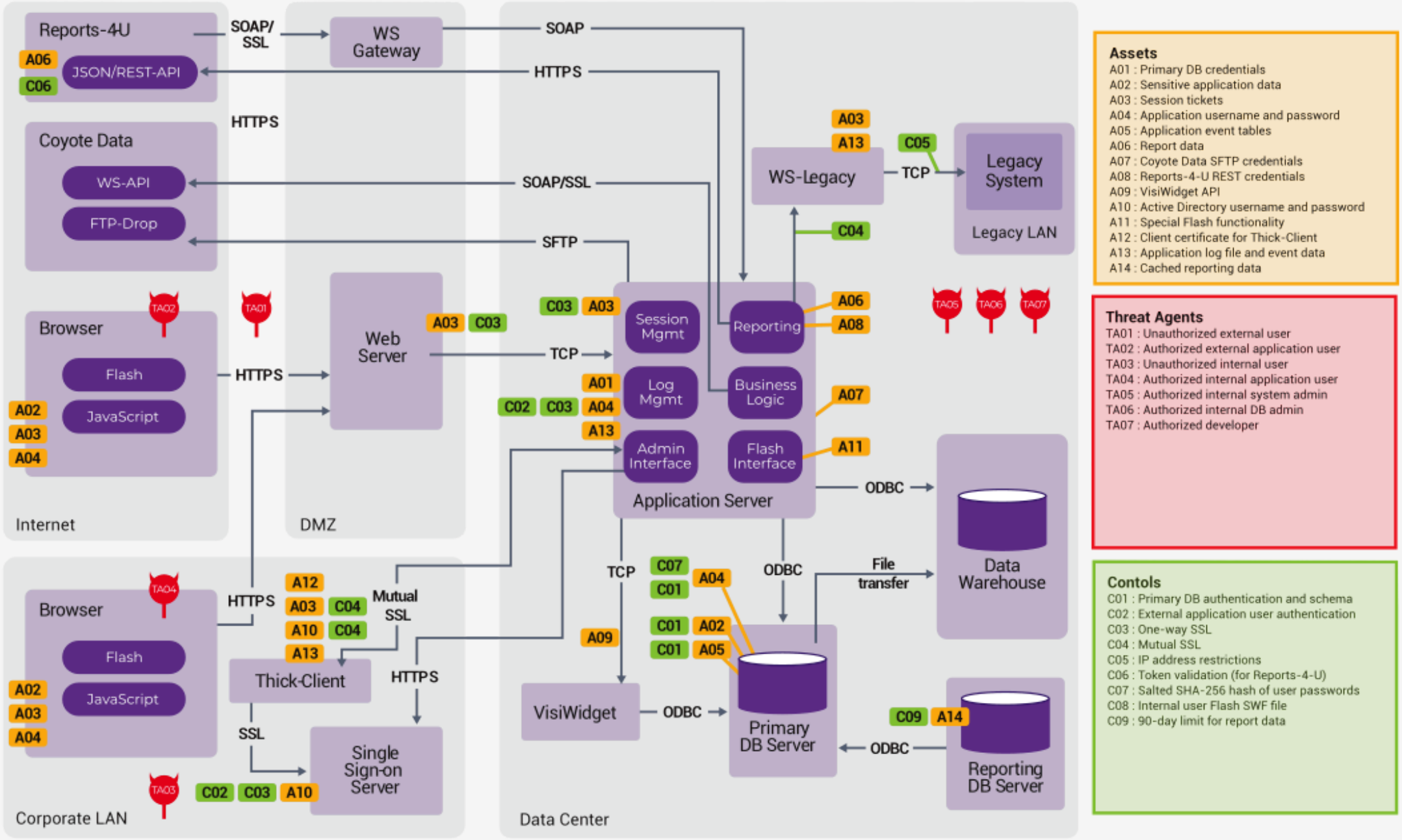
Threat modeling

😅 **Threat modeling**: diagramas y diseño de la arquitectura

- Identificar requisitos de seguridad
- Señalar amenazas y vulnerabilidades
- Cuantificar la criticidad de amenazas y vulnerabilidades
- Priorizar métodos de reparación

Métodos de threat modeling en arquitectura software:

- STRIDE: Microsoft SDL
- LINDDUN: Para mitigar amenazas a la privacidad
- SYNOPSIS



Assets

A01 : Primary DB credentials
A02 : Sensitive application data
A03 : Session tickets
A04 : Application username and password
A05 : Application event tables
A06 : Report data
A07 : Coyote Data SFTP credentials
A08 : Reports-4-U REST credentials
A09 : VisiWidget API
A10 : Active Directory username and password
A11 : Special Flash functionality
A12 : Client certificate for Thick-Client
A13 : Application log file and event data
A14 : Cached reporting data

Threat Agents

TA01 : Unauthorized external user
TA02 : Authorized external application user
TA03 : Unauthorized internal user
TA04 : Authorized internal application user
TA05 : Authorized internal system admin
TA06 : Authorized internal DB admin
TA07 : Authorized developer

Contols

C01 : Primary DB authentication and schema
C02 : External application user authentication
C03 : One-way SSL
C04 : Mutual SSL
C05 : IP address restrictions
C06 : Token validation (for Reports-4-U)
C07 : Salted SHA-256 hash of user passwords
C08 : Internal user Flash SWF file
C09 : 90-day limit for report data

¿Conocéis **Windsurf**? ¿Github **Copilot**?

¿Es otro respiro para quienes no les gusta programar?

Ejemplo con Windsurf/Copilot

1. Abrir código fuente Python con VSCode
   - `server-side-request.py`
   - `temporary-file-creation.py`
   - `cryptographic-key-generation.py`
2. Comprobar las reglas SounarSource:
   - RSPEC-5144: Server-side requests should not be vulnerable to forging attacks
   - RSPEC-5445: Insecure temporary file creation methods should not be used
   - RSPEC-4426: Cryptographic key generation should be based on strong parameters
3. Probar Windsurf o Github Copilot...
4. Probar otros IDE: Cursor, Copilot Agent mode, etc.

PROGRAMACIÓN DEFENSIVA

Garantizar el comportamiento de todo elemento de una aplicación ante cualquier situación de uso por incorrecta o imprevisible que ésta pueda parecer

¿La programación defensiva garantiza un software seguro?

No lo garantiza, porque al hablar de seguridad se supone la existencia de un **adversario**

La seguridad del software significa crear programas que se comportan correctamente incluso en presencia de un comportamiento malicioso.

Ejemplo en C

```c
void printMsg(FILE* file, char* msg) {
  fprintf(file, msg);
}
```

```c
void printMsg(FILE* file, char* msg) {
  fprintf(file, msg);
}
```

¿Cómo arreglamos el hecho de que algún argumento de la función pueda ser `null`?

Aplicar programación defensiva...

Aplicando programación defensiva:

```
void printMsg(FILE* file, char* msg) {
  if (file == NULL) {
    logError("attempt to print message to null file");
  } else if (msg == NULL) {
    logError("attempt to print null message");
  } else {
    fprintf(file, msg);
  }
}
```

> NOTA: Hay alternativas mejores para el tratamiento de null, como los Optionals

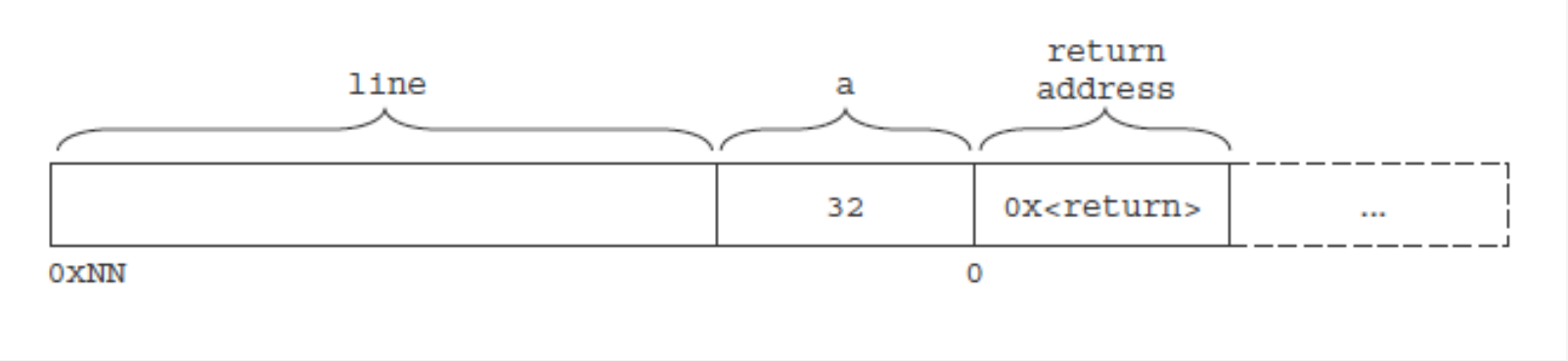
¿Desde la perspectiva de la seguridad, esta solución que aplica programación defensiva es suficiente?

Desde la perspectiva de la seguridad, esta solución no es suficiente.

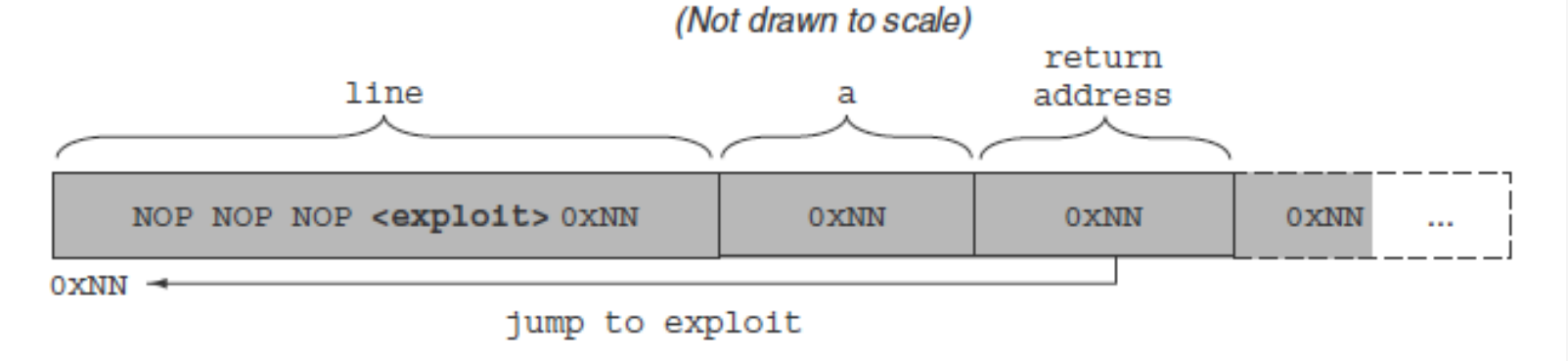
Un atacante podría especificar un string en un formato maligo, pensado para provocar un buffer overflow.

Recordemos el marco de pila de una llamada a función...

...antes de la ejecución:



...tras un ataque por buffer overflow:



LA FALACIA DE LA CALIDAD

Las organizaciones que desarrollan software tienen grupos de **ingenieros de calidad** - *Quality Assurance (QA)* dedicados a escribir y ejecutar tests y evaluar sus resultados.

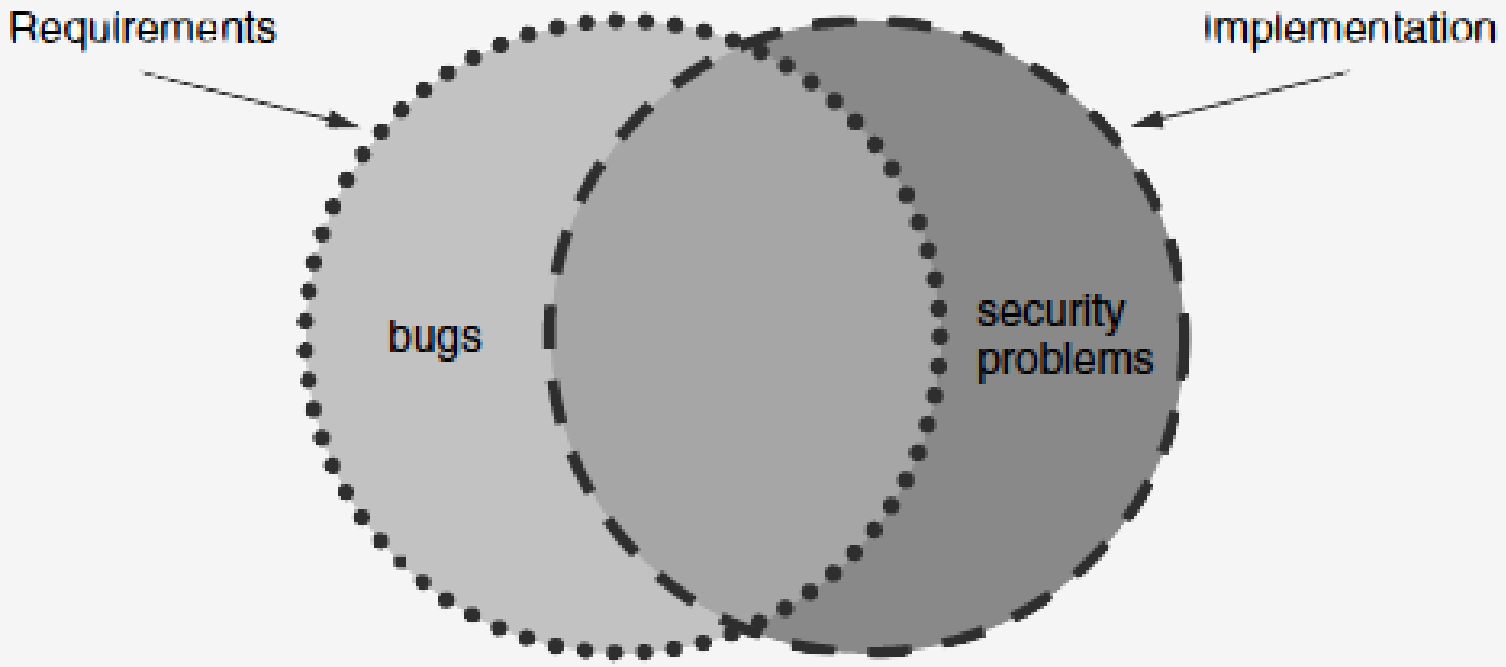
Es casi imposible mejorar la seguridad del software solo mejorando la QA

- Los **tests de funcionalidad** sirven para asegurarse de que los usuarios típicos con las necesidades típicas estén contentos, pero no sirven para encontrar defectos de seguridad que no estén relacionados con características específicas de seguridad.
- La mayor parte de las **pruebas software** sirven para comparar la implementación con los requisitos, enfoque no adecuado para encontrar problemas de seguridad.
- Los problemas de seguridad no suelen ser violaciones de los requisitos, sino **funcionalidades no deliberadas** que hacen que un programa sea inseguro.

Software fiable vs seguro

El software **fiable** hace lo que se supone que tiene que hacer.

El software **seguro** hace lo que se supone que tiene que hacer, y nada más.



Ejemplo

Implementación con JSP

```
<c:if test="${param.sayHello}">
  <!-- Let's welcome the user ${param.name} -->
  Hello ${param.name}!
</c:if>
```

- Puede que cumpla con requisitos, pero...

¿alguna vulnerabilidad?

Es vulnerable a ataques de *Cross-Site Scripting* (XSS)

Ninguna cantidad de pruebas de **caja negra** llega a ser suficiente para revelar este problema.

- Las pruebas de caja negra solo pueden comenzar cuando el sistema está terminado. Tras desplegarlo, los atacantes y los defensores están en situación de igualdad. Los atacantes pueden ahora estudiar el software.
- La suma de horas de todos los posibles atacantes puede superar con creces la suma de horas de los defensores haciendo pruebas. Los atacantes pueden encontrar más vulnerabilidades.
- Encontrar una prueba fallida es un signo de problemas, pero pasar una prueba con éxito no significa mucho.