Programación defensiva

Paradigmas de la Programación

FaMAF-UNC

2020

qué es programación defensiva?

Defend against the impossible, because the impossible will happen.

- principalmente, por el input de usuario
  - el usuario inocente
  - buffer overflows
  - code injection attacks
- también porque las condiciones cambian

cómo tratarlo?

detectar potenciales problemas

y después...

- graceful degradation (fault tolerance)
- offensive programming

no sólo cazar bugs

- cazar bugs = eliminar malfuncionamientos
- mejorar la comprensibilidad del código
- hacer código predecible ante cualquier circunstancia, especialmente las inesperadas
- reducir la superficie de ataque (no reducir el riesgo de fallos en operación normal)

detectar problemas

- errores **esperables** (el entorno los puede originar):
  - input de usuario inválido
  - se agotaron los recursos del sistema
  - falló el hardware
- errores **prevenibles** (producto del mal funcionamiento del software):
  - argumentos de función inválidos
  - valor fuera del rango
  - valor de retorno no documentado o excepción

algunas técnicas

- reuso de software inteligente
  - cuidado con el software legacy!
- canonicalización (mejor con librerías)

- y si todo falla... poner guardas!!

```
if (inesperado) { fail | excepción }
else { código del programa }
```

cuidado con el software legacy

Legacy problems are problems inherent when old designs are expected to work with today's requirements, especially when the old designs were not developed or tested with those requirements in mind.

- designed for ASCII input but now the input is UTF-8.
- when compiled on 64-bit architectures new arithmetic problems may occur (e.g. invalid signedness tests, invalid type casts, etc.).
- becomes vulnerable once network connectivity is added.
- code injection vulnerabilities, because most such problems were not widely understood at that time.

input y output seguro

- Validación
  - terminación o tratamiento con whitelist / blacklist
- Escapar los datos
- Traducir

canonicalización

Crackers are likely to invent new kinds of representations of incorrect data.

For example, if you checked if a requested file is not "/etc/passwd", a cracker might pass another variant of this file name, like "/etc/./passwd".

To avoid bugs due to non-canonical input, employ canonicalization libraries.

protegerse de lo desconocido

- se pueden aplicar estrategias más o menos conservadoras:
  - **blacklist**: defenderse de problemas conocidos
  - **whitelist**: defenderse de todo lo desconocido

buffer overflow

One of the most common problems is unchecked use of constant-size structures and functions for dynamic-size data (the buffer overflow problem). This is especially common for string data in C. C library functions like gets should never be used since the maximum size of the input buffer is not passed as an argument. C library functions like scanf can be used safely, but require the programmer to take care with the selection of safe format strings, by sanitising it before using it.

datos que vienen de la red

- Encrypt/authenticate all important data transmitted over networks.
- Do not attempt to implement your own encryption scheme, but use a proven one instead.

diseño por contrato

Design by contract uses preconditions, postconditions and invariants to ensure that provided data (and the state of the program as a whole) is sanitized. This allows code to document its assumptions and make them safely. This may involve checking arguments to a function or method for validity before executing the body of the function. After the body of a function, doing a check of object state (in object-oriented programming languages) or other held data and the return value before exits (break/return/throw/error code) is also wise.

aserciones

Within functions, you may want to check that you are not referencing something that is not valid (i.e., null) and that array lengths are valid before referencing elements, especially on all temporary/local instantiations. A good heuristic is to not trust the libraries you did not write either. So any time you call them, check what you get back from them. It often helps to create a small library of "asserting" and "checking" functions to do this along with a logger so you can trace your path and reduce the need for extensive debugging cycles in the first place. With the advent of logging libraries and aspect oriented programming, many of the tedious aspects of defensive programming are mitigated.

excepciones

- preferir excepciones a códigos de retorno

offensive programming

- el objetivo es descubrir todos los malfuncionamientos posibles, no salvarlos
- eliminar todas las estrategias del código que permiten “salvar errores”
  - no unnecessary checks (including asserts)
  - fallback code
  - shortcut code

offensive programming

- el objetivo es descubrir todos los malfuncionamientos posibles, no salvarlos
- eliminar todas las estrategias del código que permiten “salvar errores”
  - no unnecessary checks (including asserts)
  - fallback code
  - shortcut code

confiar en los datos

confiar en el software

ejemplo de programación “demasiado” defensiva

```
if (is_legacy_compatible(user_config)) {
    // Strategy: Don't trust that the new
code behaves the same
    old_code(user_config);
} else {
    // Fallback: Don't trust that the new
code handles the same cases
    if (new_code(user_config) != OK) {
        old_code(user_config);
    }
}
```

estrategia ofensiva

```
// Trust that the new code has no new bugs
new_code(user_config);
```

para seguir leyendo

https://en.wikipedia.org/wiki/Defensive_programming

https://en.wikipedia.org/wiki/Offensive_programming

http://wiki.c2.com/?DefensiveProgramming