Defensive programming

From Wikipedia, the free encyclopedia

Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances. Defensive programming practices are often used where high availability, safety or security is needed.

Defensive programming is an approach to improve software and source code, in terms of:

- General quality – reducing the number of software bugs and problems.
- Making the source code comprehensible – the source code should be readable and understandable so it is approved in a code audit.
- Making the software behave in a predictable manner despite unexpected inputs or user actions.

Overly defensive programming however introduces unnecessary code for errors impossible to even happen, thus wasting runtime and maintenance costs. There is also the risk that the code traps or prevents too many exceptions, potentially resulting in unnoticed, incorrect results.

Contents

Secure programming

Secure programming is the subset of defensive programming concerned with computer security. That is to say, security is the concern, not necessarily safety or availability (the software may be allowed to fail in certain ways). As with all kinds of defensive programming, avoiding bugs is a primary objective, however the motivation is not as much to reduce the likelihood of failure in normal operation (as if safety was the concern), but to reduce the attack surface – the programmer must assume that the software might be misused actively to reveal bugs, and that bugs could be exploited maliciously.

```
int risky_programming(char *input){
  char str[1000+1];     // one more for the null character
  // ...
  strcpy(str, input);   // copy input
  // ...
}
```

The function will crash when the input is over 1000 characters. Some novice programmers may not feel that this is a problem, supposing that no user will enter such a long input. This particular bug demonstrates a vulnerability which enables buffer overflow exploits. Here is a solution to this example:

```
int secure_programming(char *input){
  char str[1000];
  // ...
  strncpy(str, input, sizeof(str)); // copy input without exceeding the length of the destinatic
  str[sizeof(str) - 1] = '\0'; // if strlen(input) >= sizeof(str) then strncpy won't NUL terminc
  // ...
}
```

Offensive programming

Offensive programming can be considered a category of defensive programming, with the added emphasis that certain errors should *not* be handled defensively. In this practice, only errors from outside the program's control are to be handled (such as user input); the software itself, as well as data from within the program's line of defense, are to be trusted in this methodology.

Trusting internal data validity

Overly defensive programming

```
const char* trafficlight_colorname(enum trafficlight_color c) {
    switch (c) {
        case TRAFFICLIGHT_RED:    return "red";
        case TRAFFICLIGHT_YELLOW: return "yellow";
        case TRAFFICLIGHT_GREEN:  return "green";
    }
    return "black"; // To be handled as a dead traffic light.
}
```

Offensive programming

```
const char* trafficlight_colorname(enum trafficlight_color c) {
    switch (c) {
        case TRAFFICLIGHT_RED:    return "red";
        case TRAFFICLIGHT_YELLOW: return "yellow";
        case TRAFFICLIGHT_GREEN:  return "green";
    }
    assert(0); // Assert that this section is unreachable.
}
```

Trusting software components

Overly defensive programming

```
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);
    }
}
```

Offensive programming

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

Techniques

Here are some defensive programming techniques:

Intelligent source code reuse

If existing code is tested and known to work, reusing it may reduce the chance of bugs being introduced.

However, reusing code is not *always* a good practice, because it also amplifies the damages of a potential attack on the initial code. Reuse in this case may cause serious business process bugs.

Legacy problems

Before reusing old source code, libraries, APIs, configurations and so forth, it must be considered if the old work is valid for reuse, or if it is likely to be prone to legacy problems.

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.

Many software products have experienced problems with old legacy source code, for example:

- Legacy code may not have been designed under a defensive programming initiative, and might therefore be of much lower quality than newly designed source code.
- Legacy code may have been written and tested under conditions which no longer apply. The old quality assurance tests may have no validity any more.
  - **Example 1**: legacy code may have been designed for ASCII input but now the input is UTF-8.
  - **Example 2**: legacy code may have been compiled and tested on 32-bit architectures, but when compiled on 64-bit architectures new arithmetic problems may occur (e.g. invalid signedness tests, invalid type casts, etc.).
  - **Example 3**: legacy code may have been targeted for offline machines, but becomes vulnerable once network connectivity is added.
- Legacy code is not written with new problems in mind. For example, source code written about 1990 is likely to be prone to many code injection vulnerabilities, because most such problems were not widely understood at that time.

Notable examples of the legacy problem:

- BIND 9, presented by Paul Vixie and David Conrad as "BINDv9 is a complete rewrite", "Security was a key consideration in design" * (http://impressive.net/archives/fogo/20001005080818.O15286@impressive.net), naming security, robustness, scalability and new protocols as key concerns for rewriting old legacy code.
- Microsoft Windows suffered from "the" Windows Metafile vulnerability and other exploits related to the WMF format. Microsoft Security Response Center describes the WMF-features as *"Around 1990, WMF support was added... This was a different time in the security landscape... were all completely trusted"* * (http://blogs.technet.com/msrc/archive/2006/01/13/417431.aspx), not being developed under the security initiatives at Microsoft.
- Oracle is combating legacy problems, such as old source code written without addressing concerns of SQL injection and privilege escalation, resulting in many security vulnerabilities which has taken time to fix and also generated incomplete fixes. This has given rise to heavy criticism from security experts such as David Litchfield, Alexander Kornbrust, Cesar Cerrudo (1 (http://seclists.org/lists/bugtraq/2006/May/0039.html), 2 (http://seclists.org/lists/bugtraq/2006/May/0045.html), 3 (http://seclists.org/lists/bugtraq/2006/May/0083.html)). An additional criticism is that default installations (largely a legacy from old versions) are not aligned with their own security recommendations, such as Oracle Database Security Checklist (http://www.oracle.com/technology/deploy/security/database-security/pdf/twp_security_checklist_database.pdf), which is hard to amend as many applications require the less secure legacy settings to function correctly.

Secure input and output handling

Canonicalization

Malicious users are likely to invent new kinds of representations of incorrect data. For example, if a program checks if the requested file is not "/etc/passwd", a cracker might pass another variant of this file name, like "/etc/./passwd". Canonicalization libraries can be employed to avoid bugs due to non-canonical input.

Low tolerance against "potential" bugs

Assume that code constructs that appear to be problem prone (similar to known vulnerabilities, etc.) are bugs and potential security flaws. The basic rule of thumb is: "I'm not aware of all types of security exploits. I must protect against those I *do* know of and then I must be proactive!".

Other techniques

- 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.
- Encrypt/authenticate all important data transmitted over networks. Do not attempt to implement your own encryption scheme, but use a proven one instead.
- All data is important until proven otherwise.
- All data is tainted until proven otherwise.
- All code is insecure until proven otherwise.
  - You cannot prove the security of any code in userland, or, more canonically: *"never trust the client"*.
- If data are to be checked for correctness, verify that they are correct, not that they are incorrect.
- Design by contract
  - 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.
- Assertions
  - 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.
- Prefer exceptions to return codes
    - Generally speaking, it is preferable to throw intelligible exception messages that enforce part of your API contract and guide the client programmer instead of returning values that a client programmer is likely to be unprepared for and hence minimize their complaints and increase robustness and security of your software.

See also

- Computer security
- Immunity-aware programming

External links

- CERT Secure Coding Standards (https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards)
- "Secure Programming for Linux and Unix HOWTO" (http://www.dwheeler.com/secure-programs) by David A. Wheeler

Categories: Programming paradigms | Programming principles