Defensive programming [closed]

Asked 16 years, 4 months ago Modified 11 years, 1 month ago

Viewed 3k times

11

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 9 years ago.

Improve this question

When writing code do you consciously program defensively to ensure high program quality and to avoid the possibility of your code being exploited maliciously, e.g. through buffer overflow exploits or code injection ?

What's the "minimum" level of quality you'll always apply to your code ?

security defensive-programming

Share

Improve this question

edited Nov 14, 2013 at 22:00

user1228

Follow

asked Aug 9, 2008 at 17:02

David

14.2k ● 25 ● 82 ● 102

14 Answers

Sorted by: Highest score (default)

12

In my line of work, our code has to be top quality.
So, we focus on two main things:

1. Testing
2. Code reviews

Those bring home the money.

Share Improve this answer

Follow

edited Aug 31, 2011 at 23:00

EvilTeach

28.8k ● 21 ● 88 ● 144

answered Aug 9, 2008 at 18:00

abyx

72.6k ● 19 ● 97 ● 121

4

Similar to abyx, in the team I am on developers always use unit testing and code reviews. In addition to that, I also aim to make sure that I don't incorporate code that people *may* use - I tend to write code only for the basic set of methods required for the object at hand to function as has been spec'd out. I've found that incorporating methods that may never be used, but provide

functionality can unintentionally introduce a "backdoor" or unintended/unanticipated use into the system.

It's much easier to go back later and introduce methods, attributes, and properties for which are asked versus anticipating something that may never come.

Share Improve this answer

Follow

edited Nov 23, 2009 at 23:51

abyx

72.6k ● 19 ● 97 ● 121

answered Aug 9, 2008 at 18:05

Tom

15.9k ● 5 ● 50 ● 63

2 What you describe is also known as YAGNI - en.wikipedia.org/wiki/YAGNI – abyx Nov 23, 2009 at 23:54

1

I'd recommend being defensive for data that enter a "component" or framework. Within a "component" or framework one should think that the data is "correct".

Thinking like this. It is up to the caller to supply correct parameters otherwise ALL functions and methods have to check every incomming parameter. But if the check is only done for the caller the check is only needed once. So, a parameter should be "correct" and thus can be passed through to lower levels.

1. Always check data from external sources, users etc

2. A "component" or framework should always check incomming calls.

If there is a bug and a wrong value is used in a call. What is really the right thing todo? One only have an indication that the "data" the program is working on is wrong and some like ASSERTS but others want to use advanced error reporting and possible error recovery. In any case the data is found to be faulty and in few cases it's good to continue working on it. (note it's good if servers don't die at least)

An image sent from a satellite might be a case to try advanced error recovery on...an image downloaded from the internet to put up an error icon for...

Share Improve this answer

Follow

answered Aug 10, 2008 at 9:26

epatel

46k ● 17 ● 111 ● 144

1

I recommend people write code that is fascist in the development environment and benevolent in production.

During development you want to catch bad data/logic/code as early as possible to prevent problems either going unnoticed or resulting in later problems where the root cause is hard to track.

In production handle problems as gracefully as possible. If something really is a non-recoverable error then handle it and present that information to the user.

As an example here's our code to Normalize a vector. If you feed it bad data in development it will scream, in production it returns a safety value.

```
inline const Vector3 Normalize( Vector3arg vec )
{
    const float len = Length(vec);
    ASSERTMSG(len > 0.0f "Invalid Normalization");
    return len == 0.0f ? vec : vec / len;
}
```

Share Improve this answer

Follow

answered Aug 13, 2008 at 18:15

Andrew Grant

58.8k ● 22 ● 131 ● 144

We use a similar way to "increase" the problem detection on legacy code. The macro is easy to add, on its only effect is a debug messagebox while testing. This is not a miracle solution, but it is better than the "return FALSE ;" no one ever checks – paercebal Sep 18, 2008 at 13:11

0

I always work to prevent things like injection attacks. However, when you work on an internal intranet site, most of the security features feel like wasted effort. I still do them, maybe just not as well.

Share Improve this answer

Follow

answered Aug 9, 2008 at 17:09

EndangeredMassa

17.5k ● 8 ● 56 ● 80

0

Well, there is a certain set of best practices for security. At a minimum, for database applications, you need to watch out for SQL Injection.

Other stuff like hashing passwords, encrypting connection strings, etc. are also a standard.

From here on, it depends on the actual application.

Luckily, if you are working with frameworks such as .Net, a lot of security protection comes built-in.

Share Improve this answer

Follow

answered Aug 9, 2008 at 17:47

Vaibhav

11.4k ● 11 ● 53 ● 71

0

You have to always program defensively I would say even for internal apps, simply because users could just through sheer luck write something that breaks your app. Granted you probably don't have to worry about trying to cheat you out of money but still. Always program defensively and assume the app will fail.

Share Improve this answer

Follow

answered Aug 9, 2008 at 17:54

Justin Yost

2,340 ● 19 ● 31

Using Test Driven Development certainly helps. You write a single component at a time and then enumerate all of the potential cases for inputs (via tests) before writing the

0

code. This ensures that you've covered all bases and haven't written any ***cool*** code that no-one will use but might break.

Although I don't do anything formal I generally spend some time looking at each class and ensuring that:

1. if they are in a valid state that they stay in a valid state
2. there is no way to construct them in an invalid state
3. Under exceptional circumstances they will fail as gracefully as possible (frequently this is a cleanup and throw)

Share Improve this answer

Follow

answered Aug 10, 2008 at 0:53

Mike Minutillo

54.8k ● 14 ● 48 ● 41

0

It depends.

If I am genuinely hacking something up for my own use then I will write the best code that I don't have to think about. Let the compiler be my friend for warnings etc. but I won't automatically create types for the hell of it.

The more likely the code is to be used, even occasionally, I ramp up the level of checks.

- minimal magic numbers
- better variable names

- fully checked & defined array/string lengths
- programming by contract assertions
- null value checks
- exceptions (depending upon context of the code)
- basic explanatory comments
- accessible usage documentation (if perl etc.)

Share Improve this answer

Follow

answered Aug 10, 2008 at 8:13

itj

1,165 ● 1 ● 11 ● 16

0

I'll take a different definition of defensive programming, as the one that's advocated by *Effective Java* by Josh Bloch. In the book, he talks about how to handle mutable objects that callers pass to your code (e.g., in setters), and mutable objects that you pass to callers (e.g., in getters).

- For setters, make sure to clone any mutable objects, and store the clone. This way, callers cannot change the passed-in object after the fact to break your program's invariants.
- For getters, either return an immutable view of your internal data, if the interface allows it; or else return a clone of the internal data.
- When calling user-supplied callbacks with internal data, send in an immutable view or clone, as appropriate, unless you intend the callback to alter

the data, in which case you have to validate it after the fact.

The take-home message is to make sure no outside code can hold an alias to any mutable objects that you use internally, so that you can maintain your invariants.

Share Improve this answer

Follow

answered Aug 10, 2008 at 8:26

C. K. Young

223k ● 47 ● 390 ● 443

0

I am very much of the opinion that correct programming will protect against these risks. Things like avoiding deprecated functions, which (in the Microsoft C++ libraries at least) are commonly deprecated because of security vulnerabilities, and validating everything that crosses an external boundary.

Functions that are only called from your code should not require excessive parameter validation because you control the caller, that is, no external boundary is crossed. Functions called by other people's code should assume that the incoming parameters will be invalid and/or malicious at some point.

My approach to dealing with exposed functions is to simply crash out, with a helpful message if possible. If the caller can't get the parameters right then the problem is in their code and they should fix it, not you. (Obviously you have provided documentation for your function, since it is exposed.)

Code injection is only an issue if your application is able to elevate the current user. If a process can inject code into your application then it could easily write the code to memory and execute it anyway. Without being able to gain full access to the system code injection attacks are pointless. (This is why applications used by administrators should not be writeable by lesser users.)

Share Improve this answer

Follow

answered Aug 10, 2008 at 11:27

Zooba

11.4k ● 3 ● 38 ● 40

0

In my experience, positively employing defensive programming does not necessarily mean that you end up improving the quality of your code. Don't get me wrong, you need to defensively program to catch the kinds of problems that users will come across - users don't like it when your program crashes on them - but this is unlikely to make the code any easier to maintain, test, etc.

Several years ago, we made it policy to use assertions at all levels of our software and this - along with unit testing, code reviews, etc. plus our existing application test suites - had a significant, positive effect on the quality of our code.

Share Improve this answer

Follow

answered Aug 13, 2008 at 17:36

Nick

761 ● 4 ● 9

0

Java, Signed JARs and JAAS.

Java to prevent buffer overflow and pointer/stack whacking exploits.

Don't use JNI. (Java Native Interface) it exposes you to DLL/Shared libraries.

Signed JAR's to stop class loading being a security problem.

JAAS can let your application not trust anyone, even itself.

J2EE has (admittedly limited) built-in support for Role based security.

There is some overhead for some of this but the security holes go away.

Share Improve this answer

Follow

answered Sep 5, 2008 at 5:31

Tim Williscroft

3,735 ● 25 ● 37

0

Simple answer: **It depends**. Too much defensive coding **can** cause major performance issues.

Share Improve this answer

Follow

answered Sep 7, 2009 at 11:28

Locked