Security Principles

March 2, 2026

Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Security Principles

Software Architecture

March 2, 2026

Brae Webb

1 Introduction

One quality attribute which developers often overlook is security. Security can be the cause of a great deal of frustration for developers; there are no comfortable architectures, nor command-line tools to magically make an application secure. While the world depends on technology more than ever, and, at the same time the impacts of cyber attacks become more devastating, it has become crystal clear that security is everyone's responsibility. As users of technology, as developers, and as architects, we all need to ensure we take security seriously.

Learning, and for that matter, teaching, how to make software secure is no easy task. Every application has different flaws and risks, every attack vector and exploit is unique; managing to keep up with it all is daunting. None of us will ever be able to build a completely secure system but that is no reason to stop trying. As developers and architects, security should be an on-going process in the back of your minds. A nagging voice which constantly asks 'what if?'.

We introduce security first to set the example. As we go through this course, the principle of security will recur again and again. With each architecture introduced, we will stop and ask ourselves 'what if?'. In your future careers, you should endeavour to continue this same practice. Each feature, pipeline, access control change, or code review, ask yourself, 'what are the security implications?'.

With that said, there are some useful principles, and a handful of best practices which we will explore. But again, even if you follow these practices and embody the principles, your applications will still be hopelessly insecure, unless, you constantly reflect on the security implications of your each and every decision.

2 You

Before we even fantasise about keeping our applications secure, let's review if you are secure right now. As developers we often have heightened privileges and access, at times above that of even company CEOs. If you are not secure, then nor is anything on which you work. Let's review some of the basics.

Keep your software up to date. Are you running the latest version of your operating system? The latest Chrome, or Firefox, or god-forbid, Edge? If not, then there is a good chance you are currently at risk. Software updates, while annoying, provide vital patches to discovered exploits. You must keep your software up to date.

Use multi-factor authentication. This may be hard to explain to your grandmother but this should be obvious to software developers. One million passwords are stolen every week [1]. If you do not have some sort of multi-factor authentication enabled, hackers can access your account immediately after stealing your password.

Use a password manager. Following from the startling statistic of a million stolen passwords per week, we must seriously consider how we use passwords. Our practices should be guided by the fact that at least one service we use likely stores our password insecurely. We should assume that our password will be compromised. What can we do about this? The first thing is to avoid password reuse, one password per service. Of course, humans have very limited memory for remembering good passwords. So go through and update your passwords with randomly generated secure passwords, and then store them in a password manager.

3 Principles of Securing Software

Okay, now that we are not a security risk ourselves, we can start considering how to secure the software we develop. Before looking at pragmatic practices, we will develop a set of guiding principles. These principles are far from comprehensive but they provide a useful foundation to enable discussion of our security practices. The principles presented in this course are derived from Saltzer and Schroeder [2], Gasser [3], and Viega and McGraw [4]. Some principles have been renamed for consistency and clarity. Refer to Appendix A for the comprehensive list of principles from these sources.

3.1 Principle of Least Privilege

> Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.
>
> — Jerry Saltzer [5]

The principle of least privilege was identified in 1974 by Jerry Saltzer [5]. This principle is mostly common sense and the benefits should be apparent. If you maintain the principle of least privilege then you minimise your attack surface by limiting the amount of damage any one user can do. This protects software from intentionally malicious actors while also minimising the damage of bugs which occur unintentionally in the software.

Example Consider a web application which lists COVID close contact locations for a state. We will assume that the locations are maintained within an SQL database. Assume that each time the tracing page is loaded, an SQL query is sent to the database to find all current close contact locations. If the developers follow the principle of least privilege, then the account used to query that data would only be able to list the current locations.

For this example, the tracing website had to be developed and rolled out quickly, as such, the developers created only one SQL user which they used for both the tracing website and the portal where the government can log new locations. This user account would have the ability to create new close contact locations, and if done poorly enough, the account might even have access to delete existing locations.

Since the developers have violated the principle of least privilege, their software is now at risk. If a malicious actor is able to gain database access via SQL injection, or, just as likely, if the software has a typo in an SQL query, the integrity of the tracing data could be jeopardised. This could be mitigated by using the principle of least privilege and creating separate user accounts for modifying and subsequently viewing the data.

Exemplar One of the primary examples of a good application of this principle is within Unix operating systems. In the Unix operating system, a sudoer (a user account which can use the `sudo` command) has a lot of destructive power. Commands running at the sudo level can do essentially anything they wish, including wiping the computer. However, a sudoer is not typically using these privileges. The user has to specify that they intend to run a command with elevated privileges, which helps avoid accidental destruction of the computer.

Fail-safe defaults The principle of fail-safe defaults is often presented on its own. Fail-safe defaults means that the default for access to a resource should be denied until explicit access is provided. We include fail-safe defaults as a property of the principle of least privilege given the strong connection.

3.2 Principle of Failing Securely

> Death, taxes, and computer system failure are all inevitable to some degree. Plan for the event.
>
> — Howard and LeBlanc [6]

Computer systems fail. As we will see in this course, the more complicated your software, the more often and dramatically it can be expected to fail. The principle of failing securely asks us to stash away our optimism and become a realist for a moment. When designing an architecture or a piece of software, plan for the ways your software will fail. And when your software does fail, it should not create a security vulnerability [7].

Example An interesting example of failing securely comes from Facebook's October 2021 outage which we discussed previously. As you may be aware, one cause of the outage was a DNS resolution issue triggered by Facebook's data centres going offline [8]. The DNS resolution issue meant that the internal Facebook development tools and communication channels went down as well. As you might expect, losing your tools and access to your team members makes resolving an outage far more difficult.

Early reports of the incident indicated that the outage of Facebook's internal infrastructure also meant employees were locked out of the data centres. While it seems that developers were locked out of their buildings, the data centres were not affected. Nevertheless, it is interesting to consider whether an internal outage should, somewhat understandably, lock access to the data centres.

This example highlights the key difference between a system which *fails safely*[1] and a system which *fails securely*. In a fail *safe* system, an internal outage would allow physical access to the data centre to enable maintenance to fix the problem. Whereas in a fail *secure* system, the outage would cause the data centre to lock and prevent access to maintain the security of the data within. There isn't one correct way to deal with failure. While in this case it would have helped Facebook resolve the issue quicker, if a data breach occurred through an intentional outage there would be equal criticism.

Regardless of the security policy you choose, it is always important to prepare for failure and weigh the pros and cons of each policy.

3.3 Principle of KISS

> Simplicity is the ultimate sophistication
>
> — Leonardo Da Vinci[2]

We will keep this principle simple. The principle of Keep it Simple Stupid (KISS) is needed as complicated software or processes are, more often than not, insecure. Simple means less can go wrong.

3.4 Principle of Open Design

> One ought to design systems under the assumption that the enemy will immediately gain full familiarity with them.
>
> — C. E. Shannon [9]

The principle of open design, also known as Kerckhoffs' principle, stresses that security through obscurity, or security through secrecy, does not work. If the security of your software relies on keeping certain implementation details secret then your system is not secure. Of course, there are some acceptable secrets such as private keys, however, these should still be considered a vulnerability. Always assume that if an implementation detail can be discovered, it will be. There is software constantly scanning the internet for open ports, unpublished IP addresses, and routers secured by the default username and password.

[1] No relation to fail-safe defaults.
[2] maybe

Example An example which immediately springs to mind is our first and second year assignment marking tools. To date, I am not aware of the tools being exploited, however they are currently vulnerable. The tools currently rely on students not knowing how they work. There are ways to create 'assignment' submissions which pass all the functionality tests for any given semester. Fortunately, the threat of academic misconduct is enough of a deterrent that it has yet to be a problem.

The example does illustrate why the principle of open design is so frequently violated. In the short-term security through obscurity can work, and work well, but it is a long-term disaster waiting to happen. It is also common place to violate the principle slightly by trying to build systems which do not rely on secrecy but keeping the systems secret 'just in case'. In many situations this is appropriate, however, a truly secure system should be open for community scrutiny.

3.5 Principle of Usability

The final principle we should explore is the principle of usability, also known as 'psychological acceptability'. This principle asks that we have realistic expectations of our users. If the user is made to jump through numerous hoops to securely use your application, they will find a way around it. The idea is that the security systems put in place should, as much as possible, avoid making it more difficult to access resources.

Example The example for this principle includes a confession. The university has implemented a multi-factor authentication mechanism for staff and students. Unfortunately, there was a bug in the single sign-on which means that MFA is not remembered causing the system to re-prompt me at every *single* log in. A direct consequence of this inconvenience is that I have installed software on all my devices which automatically authenticates the MFA, bypassing, in part, the intended additional security.

The university through violating the principle of usability has made it more difficult for users to be secure than insecure. As predicted by the principle, the inconvenience leads straight to bypassing. Violation of this principle often has relatively minimal *visible* impacts, which results in the principle not being considered as often. The long-term implications are that what was an annoyance circumvented by your users, may become the cause of a major security breach long after the security feature was implemented.

4 Practices of Secure Software

With some of the guiding principles of secure software now covered, there are a few useful practices which can be employed to help secure our systems.

4.1 Encryption

In the arms race between hackers and security practitioners *encryption* was one of the first weapons wielded. Encryption is the act of encoding data in a way which is difficult to decode. The most notable early example of encryption was the Enigma Machine in the 1930s, if you trace the history of cryptography back far enough you will eventually end up in World War II Germany [10].

While encryption is outside of the scope of this course, it is well worth having knowledge of available encryption techniques. You should not attempt to design your own encryption algorithms, unless you are an expert in security and encryption. Rather, you should use existing well-known and well-tested encryption algorithms. In the design of a secured software system encryption can play many roles.

4.2 Sanitisation

Another practice which you should endeavour to utilise is sanitisation. If we assume that you cannot tell the difference between your user and a potential attacker, then you cannot trust your user. If you cannot

trust your user, then you should not trust the data they give you. One of the oldest and most common forms of attacks is user input injection. In such an injection attack, a user intentionally or unintentionally provides input which damages your system.

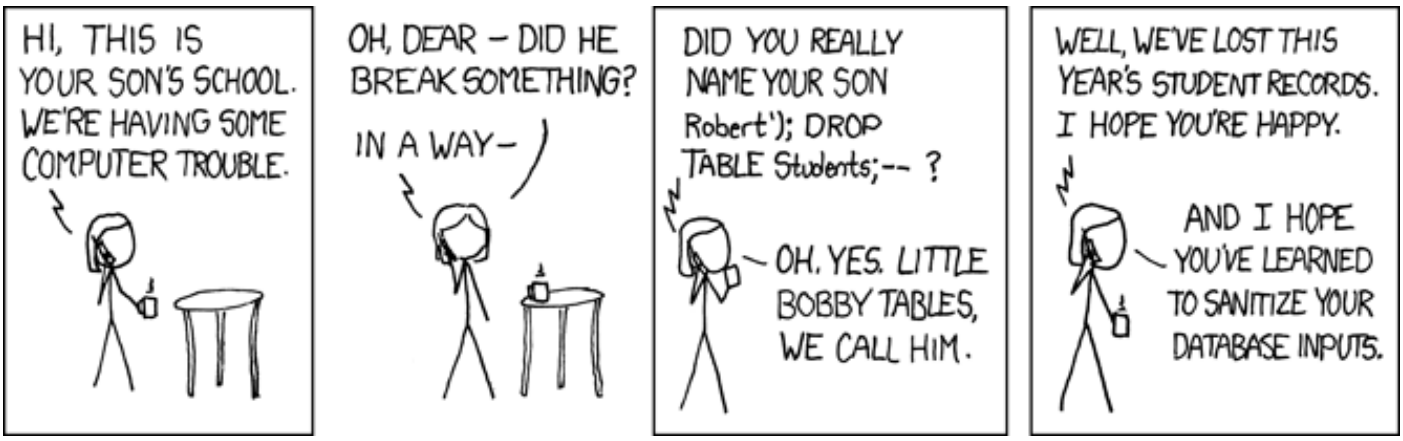

Figure 1: https://xkcd.com/327/

When done maliciously, attackers enter characters or sequences of characters which are commonly used to escape a string. The input which follows would then be run as code on the victim's system. The method for preventing against injection attacks is called *sanitisation*, which is simply a process of removing dangerous or unnecessary characters from a user's input.

4.3 Firewalls

A firewall is a piece of networking software which is designed to protect a computer system against unwanted traffic. Firewalls scan incoming and outgoing network packets and are able to drop the packet if it is deemed unwanted. Firewalls can be configured by a set of rules defining which traffic is unwanted. A firewall rule could specify that a computer system drop all packets destined for port 80 or all packets not destined for port 80. They may also be configured to support more advanced packet filtering.

From our perspective the primary advantage of a firewall is to prevent traffic which we did not intend. If we are hosting a web server, only allowing traffic on port 80 and port 443 is desirable, and prevents users accessing a database on the server which we forgot to password protect. The principle for firewalls is to block by default and allow when required.

4.4 Dependency Management

Modern software has a lot of dependencies. Each new dependency a software adopts is a new potential attack surface. Relying on software which is automatically updated is dangerous, the authors can at any point inject malicious code into your system. There is a particularly interesting recent example of dependency injection which is worth reading about [11].

One simple practice to help prevent against dependency injection is by using lock files. Most package managers generate lock files when installing software. Lock files keep track of the exact version of a library that is installed. If you ask a package manager to install version >=2.4.0 of a library and it actually downloads version 2.5.6, this will be tracked in the lock file. Additionally, lock files track the hash of the dependency so that if someone attempts a dependency injection, the code change will be noticed by the package manager. Lock files should be tracked in version control and updated periodically when a safe and tested version of a dependency is released.

5 Conclusion

We have looked at a few guiding principles for designing and developing secure software. This list of principles is incomplete, security requires constant consideration. Software security is an ongoing arms race for which we are all currently unqualified to enlist. Secure yourself as best you can and, when you are in doubt, consult with or hire an expert.

A Original Security Design Principles

Saltzer and Schroeder

1. Economy of mechanism ..... Principle of KISS
2. Fail-safe defaults ..... Principle of Least Privilege
3. Complete mediation ..... Not covered
   Access to resources must *always* be authorised.
4. Open design ..... Principle of Open Design
5. Separation of privilege ..... Not covered
   No one user account or role should hold too much power.
   Consider multi-role authentication where appropriate.
6. Least privilege ..... Principle of Least Privilege
7. Least common mechanism ..... Not covered
   Minimise the amount of resources shared between users.
8. Psychological acceptability ..... Principle of Usability

Gasser

1. Consider Security from the Start ..... Implicit
2. Anticipate Future Security Requirements
3. Minimise and Isolate Security Controls
4. Enforce Least Privilege ..... Principle of Least Privilege
5. Structure the Security-Relevant Functions
6. Make Security Friendly ..... Principle of Usability
7. Do Not Depend on Secrecy for Security ..... Principle of Open Design

Viega and McGeaw

1. Secure the Weakest Link
2. Practice Defense in Depth
3. Fail Securely ..... Principle of Failing Securely
4. Follow the Principle of Least Privilege ..... Principle of Least Privilege
5. Compartmentalise
6. Keep It Simple ..... Principle of KISS
7. Promote Privacy
8. Remember That Hiding Secrets is Hard
9. Be Reluctant to Trust
10. Use Your Community Resources

References

[1] S. Manjarres, "2021 world password day: How many will be stolen this year?." https://www.secplicity.org/2021/05/04/2021-world-password-day-how-many-will-be-stolen-this-year/, May 2021.

[2] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.

[3] M. Gasser, *Building a Secure Computer System*, pp. 35–44. Van Nostrand Reinhold Company, January 1988.

[4] J. Viega and G. R. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, pp. 91–113. Addison-Wesley Professional, September 2001.

[5] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, pp. 388–402, July 1974.

[6] M. Howard and D. LeBlanc, *Security Principles To Live By*, p. 64. Microsoft Press Redmond, Wash., December 2002.

[7] M. Gegick and S. Barnum, "Failing securely." https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/failing-securely, December 2005.

[8] S. Janardhan, "More details about the October 4 outage." https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/, October 2021.

[9] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[10] A. Hoffman, *Web Application Security: Exploration and Countermeasures for Modern Web Applications*. O'Reilly Media, Inc., 2020.

[11] E. Roth, "Open source developer corrupts widely-used libraries, affecting tons of projects." https://www.theverge.com/2022/1/9/22874949/developer-corrupts-open-source-libraries-projects-affected, January 2022.