### <https://github.com/thecyberhex/books>

### <http://www.securitytube.net/user/Audi>

### Burp Suite

### What is Security Testing?

Security Testing is defined as a type of Software Testing that ensures software systems and applications are free from any vulnerabilities, threats, risks that may cause a big loss. Security testing of any system is about finding all possible loopholes and weaknesses of the system which might result into a loss of information, revenue, repute at the hands of the employees or outsiders of the Organization.

The goal of security testing is to identify the threats in the system and measure its potential vulnerabilities, so the system does not stop functioning or is exploited. It also helps in detecting all possible security risks in the system and help developers in fixing these problems through coding.

**Types of Security Testing:**

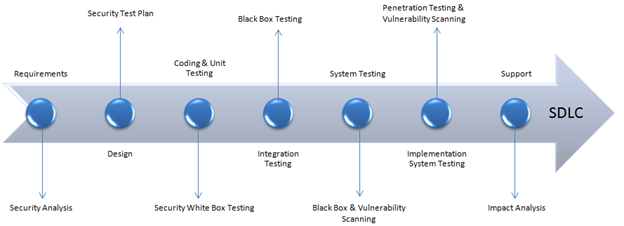
There are seven main types of security testing as per Open Source Security Testing methodology manual. They are explained as follows:

1. **Vulnerability Scanning**: This is done through automated software to scan a system against known vulnerability signatures.
2. **Security Scanning:** It involves identifying network and system weaknesses, and later provides solutions for reducing these risks. This scanning can be performed for both Manual and Automated scanning.
3. **Penetration testing**: This kind of testing simulates an attack from a malicious hacker. This testing involves analysis of a particular system to check for potential vulnerabilities to an external hacking attempt.
4. **Risk Assessment:** This testing involves analysis of security risks observed in the organization. Risks are classified as  Low, Medium and High. This testing recommends controls and measures to reduce the risk.
5. **Security Auditing:** This is an internal inspection of Applications and Operating systems for security flaws. An audit can also be done via line by line inspection of code
6. **Ethical hacking:** It's hacking an Organization Software systems. Unlike malicious hackers, who steal for their own gains, the intent is to expose security flaws in the system.
7. **Posture Assessment:** This combines Security scanning, [Ethical Hacking](https://www.guru99.com/ethical-hacking-tutorials.html) and Risk Assessments to show an overall security posture of an organization.

**How to do Security Testing**

It is always agreed, that cost will be more if we postpone security testing after software implementation phase or after deployment. So, it is necessary to involve security testing in the SDLC life cycle in the earlier phases.

Let's look into the corresponding Security processes to be adopted for every phase in SDLC

[](https://www.guru99.com/images/securityt3.png)

|  |  |
| --- | --- |
| **SDLC Phases** | **Security Processes** |
| **Requirements** | Security analysis for requirements and check abuse/misuse cases |
| **Design** | Security risks analysis for designing. Development of [Test Plan](https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html) including security tests |
| **Coding and Unit Testing** | Static and Dynamic Testing and Security [White Box Testing](https://www.guru99.com/white-box-testing.html) |
| **Integration Testing** | [Black Box Testing](https://www.guru99.com/black-box-testing.html) |
| **System Testing** | Black Box Testing and Vulnerability scanning |
| **Implementation** | [Penetration Testing](https://www.guru99.com/learn-penetration-testing.html), Vulnerability Scanning |
| **Support** | Impact analysis of Patches |

The test plan should include

* Security-related test cases or scenarios
* Test Data related to security testing
* Test Tools required for security testing
* Analysis of various tests outputs from different security tools

**Example Test Scenarios for Security Testing:**

Sample Test scenarios to give you a glimpse of security test cases -

* A password should be in encrypted format
* Application or System should not allow invalid users
* Check cookies and session time for application
* For financial sites, the Browser back button should not work.

**Methodologies/ Approach / Techniques for Security Testing**

In security testing, different methodologies are followed, and they are as follows:

* **Tiger Box**: This hacking is usually done on a laptop which has a collection of OSs and hacking tools. This testing helps penetration testers and security testers to conduct vulnerabilities assessment and attacks.
* [**Black Box**](https://www.guru99.com/black-box-testing.html): Tester is authorized to do testing on everything about the network topology and the technology.
* **Grey Box**: Partial information is given to the tester about the system, and it is a hybrid of white and black box models.

**Security Testing Roles**

* Hackers - Access computer system or network without authorization
* Crackers - Break into the systems to steal or destroy data
* Ethical Hacker - Performs most of the breaking activities but with permission from the owner
* Script Kiddies or packet monkeys - Inexperienced Hackers with programming language skill

**Myths and Facts of Security testing:**

Let's talk about an interesting topic on Myths and facts of security testing:

**Myth #1** We don't need a security policy as we have a small business

Fact: Everyone and every company need a security policy

**Myth #2**There is no return on investment in security testing

Fact: Security Testing can point out areas for improvement that can improve efficiency and reduce downtime, enabling maximum throughput.

**Myth #3**: Only way to secure is to unplug it.

Fact: The only and the best way to secure an organization is to find "Perfect Security". Perfect security can be achieved by performing a posture assessment and compare with business, legal and industry justifications.

**Myth #4**: The Internet isn't safe. I will purchase software or hardware to safeguard the system and save the business.

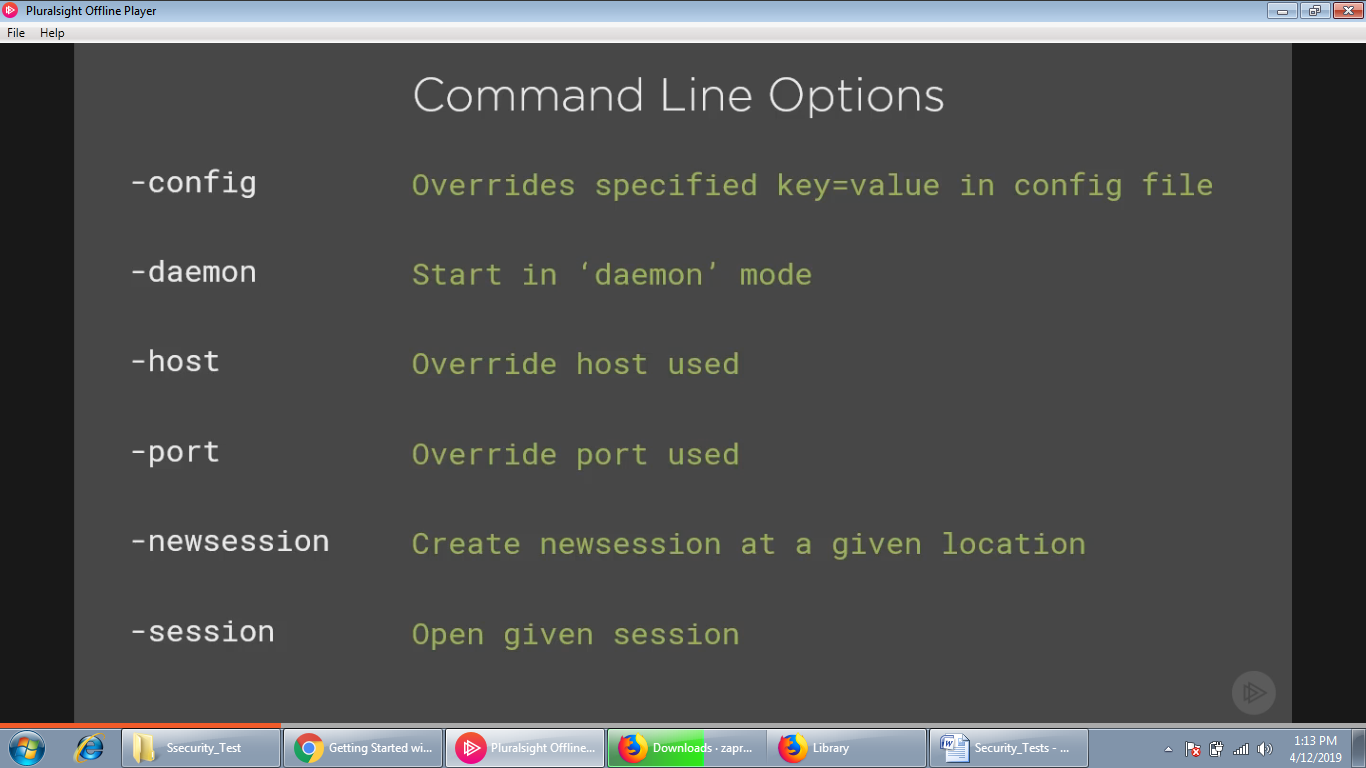
Fact: One of the biggest problems is to purchase software and hardware for security. Instead, the organization should understand security first and then apply it.

## ZAP

ZAP download Link –

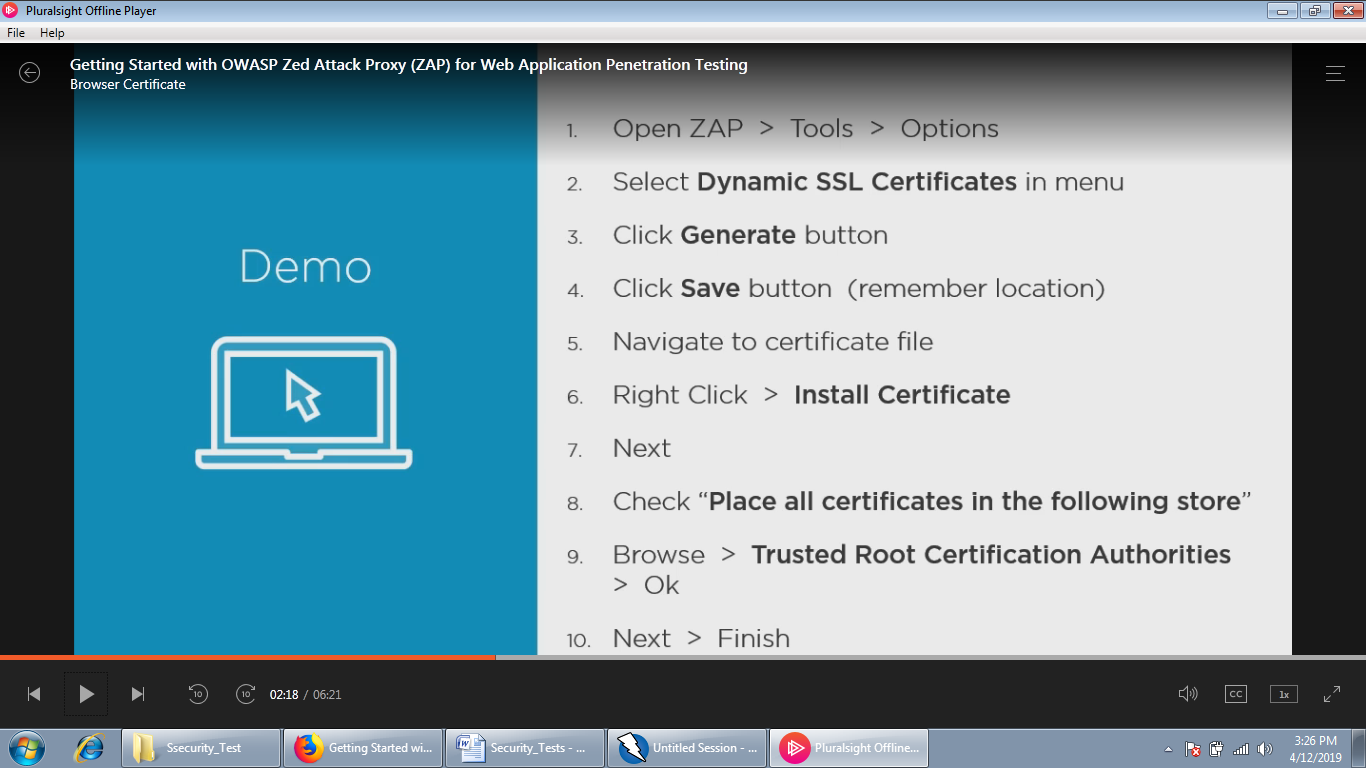
<https://github.com/zaproxy/zaproxy/wiki/downloads>

Simon Bennets – project lead of Zap project

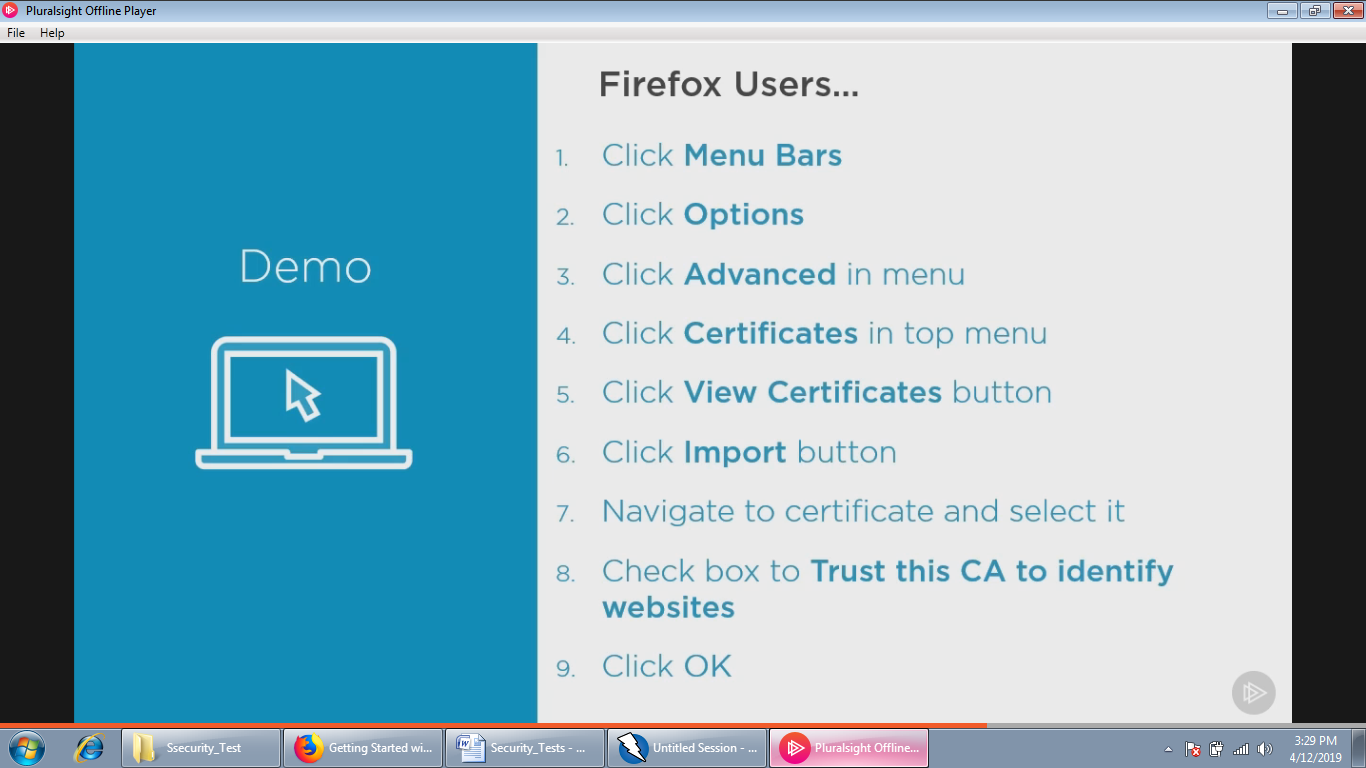


Proxy – Authority or power to act on behalf of or substitute for another.

**Install HTTPS Certificate**

****

**Import SSL Certificate in Firefox Browser**

****

Scope is the first thing to fix for penetration testing. What is the domain, do you have permission to do penetration test on domain.

**Virtual Box**

Download

<https://www.virtualbox.org/wiki/downloads>

**OWASP BWA (Broken Web Application)**

**Website**

<https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project>

**Download**

<https://sourceforge.net/projects/owaspbwa/files>

**ZAP Extensions**

<https://github.com/zaproxy/zap-extensions/wiki>

# Web Developer Security Checklist v1

Developing secure, robust web applications in the cloud is **hard, very hard**. If you think it is easy, you are either a higher form of life or you have a painful awakening ahead of you.

If you have drunk the [MVP](https://en.wikipedia.org/wiki/Minimum_viable_product) cool-aid and believe that you can create a product in one month that is both valuable and secure — think twice before you launch your “proto-product”. After you review the checklist below, acknowledge that you are skipping many of these critical security issues. At the very minimum, be honest with your potential users and let them know that you don’t have a complete product yet and are offering a prototype without full security.

This checklist is simple, and by no means complete. I’ve been developing secure web applications for over 14 years and this list contains some of the more important issues that I’ve painfully learned over this period. I hope you will consider them seriously when creating a web application.

Please comment if you have an item I can add to the list.

### ****Database****

* [ ] Use encryption for data identifying users and sensitive data like access tokens, email addresses or billing details if possible (this will restrict queries to exact match lookups).
* [ ] If your database supports low cost encryption at rest (like [AWS Aurora](https://aws.amazon.com/about-aws/whats-new/2015/12/amazon-aurora-now-supports-encryption-at-rest/)), then enable that to secure data on disk. Make sure all backups are stored encrypted as well.
* [ ] Use minimal privilege for the database access user account. Don’t use the database root account and check for unused accounts and accounts with bad passwords.
* [ ] Store and distribute secrets using a key store designed for the purpose such as [Vault](https://www.vaultproject.io/) or [AWS Secret Manager](https://aws.amazon.com/secrets-manager/). Don’t hard code secrets in your applications and NEVER check secrets into GitHub.
* [ ] Fully prevent SQL injection by only using SQL prepared statements. For example: if using NPM, don’t use npm-mysql, use npm-mysql2 which supports prepared statements.

### Development

* [ ] Ensure that all components of your software are scanned for vulnerabilities for every version pushed to production. This means O/S, libraries and packages. This should be automated into the [CI-CD](https://en.wikipedia.org/wiki/CI/CD) process.
* [ ] Secure development systems with equal vigilance to what you use for production systems. Build the software from secured, isolated development systems.

### Authentication

* [ ] Ensure all passwords are hashed using appropriate crypto such as [bcrypt](https://en.wikipedia.org/wiki/Bcrypt). Never write your own crypto and correctly initialize crypto with good random data.
* Use best-practices and proven components for login, forgot password and other password reset. Don’t invent your own — it is hard to get it right in all scenarios.
* [ ] Implement simple but adequate password rules that encourage users to have long, random passwords.
* [ ] Use multi-factor authentication for your logins to all your service providers.

### ****Denial of Service Protection****

* [ ] Make sure that DOS attacks on your APIs won’t cripple your site. At a minimum, have rate limiters on your slower API paths and authentication related APIs like login and token generation routines. Consider CAPTCHA on front-end APIs to protect back-end services against DOS.
* [ ] Enforce sanity limits on the size and structure of user submitted data and requests.
* [ ] Consider using [Distributed Denial of Service](https://en.wikipedia.org/wiki/Denial-of-service_attack) (DDOS) mitigation via a global caching proxy service like [CloudFlare](https://www.cloudflare.com/). This can be turned on if you suffer a DDOS attack and otherwise function as your DNS lookup.

### ****Web Traffic****

* [ ] Use TLS for the entire site, not just login forms and responses. Never use TLS for just the login form. Transitionally, use the strict-transport-security header to force HTTPS on all requests.
* [ ] Cookies must be httpOnly and secure and be scoped by path and domain.
* [ ] Use [CSP](https://en.wikipedia.org/wiki/Content_Security_Policy) without allowing unsafe-\* backdoors. It is a pain to configure, but worthwhile. Use CSP [Subresource Integrity](https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity) for CDN content.
* [ ] Use X-Frame-Option, X-XSS-Protection headers in client responses. Use <https://observatory.mozilla.org> to score your site.
* [ ] Use HSTS responses to force TLS only access. Redirect all HTTP request to HTTPS on the server as backup.
* [ ] Use CSRF tokens in all forms and use the new [SameSite Cookie](https://scotthelme.co.uk/csrf-is-dead/) response header which fixes CSRF once and for all newer browsers.

### ****APIs****

* [ ] Ensure that no resources are enumerable in your public APIs.
* [ ] Ensure that users are fully authenticated and authorized appropriately when using your APIs.
* [ ] Use canary checks in APIs to detect illegal or abnormal requests that indicate attacks.

### ****Validation and Encoding****

* [ ] Do client-side input validation for quick user feedback, but never trust it. Always validate and encode user input before displaying.
* [ ] Validate every last bit of user input using white lists on the server. Never directly inject user content into responses. Never use untrusted user input in SQL statements or other server-side logic.

### ****Cloud Configuration****

* [ ] Ensure all services have minimum ports open. While security through obscurity is no protection, using non-standard ports will make it a little bit harder for attackers.
* [ ] Host backend database and services on private VPCs that are not visible on any public network. Be very careful when configuring AWS security groups and peering VPCs which can inadvertently make services visible to the public.
* [ ] Isolate logical services in separate VPCs and peer VPCs to provide inter-service communication.
* [ ] Ensure all services only accept data from a minimal set of IP addresses.
* [ ] Restrict outgoing IP and port traffic to minimize APTs and “botification”.
* [ ] Always use AWS IAM users and roles and not root credentials. Invest in learning to use IAM effectively.
* [ ] Use minimal access privilege for all ops and developer staff. Give IAM users and roles the minimum capabilities required to complete the task.
* [ ] Regularly rotate passwords and access keys according to a schedule.

### ****Infrastructure****

* [ ] Ensure you can do upgrades without downtime. Ensure you can quickly update software in a fully automated manner.
* [ ] Create all infrastructure using a tool such as Terraform, and not via the cloud console. Infrastructure should be defined as “code” and be able to be recreated at the push of a button. Have zero tolerance for any resource created in the cloud by hand — Terraform can then audit your configuration.
* [ ] Use centralized logging for all services. You should never need SSH to access or retrieve logs.
* [ ] Don’t SSH into services except for one-off diagnosis. Using SSH regularly, typically means you have not automated an important task.
* [ ] Don’t keep port 22 open on any AWS service groups on a permanent basis. If you must use SSH, only use public key authentication and not passwords.
* [ ] Create [immutable hosts](http://chadfowler.com/2013/06/23/immutable-deployments.html) instead of long-lived servers that you patch and upgrade. (See [Immutable Infrastructure Can Be More Secure](https://simplesecurity.sensedeep.com/immutable-infrastructure-can-be-dramatically-more-secure-238f297eca49)).
* [ ] Use an [Intrusion Detection System](https://en.wikipedia.org/wiki/Intrusion_detection_system) to minimize [APTs](https://en.wikipedia.org/wiki/Advanced_persistent_threat).

### ****Operation****

* [ ] Power off unused services and servers. The most secure server is one that is powered down. Schedule dev servers to be [powered down](https://www.powerdown.io) after hours when not required.

### Test

* [ ] Audit your design and implementation.
* [ ] Do penetration testing — hack yourself, but also have someone other than you pen testing as well.

### ****Train****

* [ ] Train staff (especially senior staff) as to the dangers and techniques used in security social engineering.

### ****Finally, have a plan****

* [ ] Have a threat model that describes what you are defending against. It should list and prioritize the possible threats and actors.
* [ ] Have a practiced security incident plan. One day, you will need it.

# Web Development Security Checklist v2

# Web Developer Security Checklist V2



Developing secure, robust web applications in the cloud is **hard**, very hard. If you think it is easy, you are either a higher form of life or you have a painful awakening ahead of you.

If you have drunk the [MVP](https://en.wikipedia.org/wiki/Minimum_viable_product) cool-aid and believe that you can create a product in one month that is both valuable and secure — think twice before you launch your “proto-product”.

After you review the checklist below, acknowledge that you are skipping many of these critical security issues. At the very minimum, be honest with your potential users and let them know that you don’t have a complete product yet and are offering a prototype without full security.

This checklist is simple, and by no means complete. I’ve been developing secure web applications for over 14 years and this list contains some of the more important issues that I’ve painfully learned over this period. I hope you will consider them seriously when creating a web application.

This is version 2 of the checklist. It has been re-organized from [Version 1](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist-v1.html) and has a few new items by public demand (Thank you). While I try to keep the list tight and focused, please comment if you have an item that you think I should add to the list.

## Contents

* [Credentials and Secrets](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#credentials-and-secrets)
* [Authentication](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#authentication)
* [Database](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#database)
* [Apps](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#apps)
* [APIs](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#apis)
* [Network Traffic](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#network-traffic)
* [Cloud Configuration](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#cloud-configuration)
* [Infrastructure](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#infrastructure)
* [Denial of Service Protection](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#denial-of-service-protection)
* [Hack Yourself](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#hack-yourself)
* [Incident Response](https://www.powerdown.io/blog/posts/stories/web-developer-security-checklist.html#incident-response)

## ****Credentials and Secrets****

* 

Store and distribute secrets using a key store designed for the purpose. Don’t hard code secrets in your applications and definitely don't store in GitHub!. For CMS fans, don't store your credentials in a file in the document directory.

* 

Use a team-based password manager such as [1Password](https://1password.com/business/) for all service passwords and credentials. NEVER email passwords or credentials to team members.

* 

Use multi-factor authentication for all your logins to service providers.

## ****Authentication****

* 

Ensure all passwords are hashed using appropriate crypto such as [bcrypt](https://en.wikipedia.org/wiki/Bcrypt). Never write your own crypto and correctly initialize crypto with good random data. Consider using an authentication service like [Auth0](https://auth0.com/) or [AWS Cognito](https://aws.amazon.com/cognito/).

* 

Use best-practices and proven components for login, forgot password and other password reset. Don’t invent your own — it is hard to get it right in all scenarios.

* 

Implement simple but adequate password rules that encourage users to have long, random passwords.

* 

Never, EVER have any undocumented and unpublicized means of access to the device including back-door accounts (like "field-service").

* 

Run applications and containers with minimal privilege and never as root (Note: Docker runs apps as root by default).

## ****Database****

* 

Don't store sensitive data unless you truly need it. This means email addresses, personally identifying information and other personal information in general. Treat sensitive data like radioactive waste — i.e. there is an real, large and ongoing cost to securing it, and one day it can hurt you.

* 

Keep a complete list of all the places you store sensitive information: databases, file systems, Dropbox, GitHub, Vault, Office docs and even the paper folder. This is useful to manage, required by GDPR and essential if hacked. You need to be able to locate all sensitive information.

* 

If subject to GDPR, make sure you really understand the requirements and design it in from the start. For some, it will represent a major change in design and thinking. See [Privacy Cheatsheet](https://blog.varonis.com/privacy-design-cheat-sheet/) and [Intro to GDPR](https://www.cmswire.com/customer-experience/an-introduction-to-the-gdpr/).

* 

Use encryption for data identifying users and sensitive data like access tokens, email addresses or billing details if possible (this will restrict queries to exact match lookups).

* 

If your database supports low cost encryption at rest (like [AWS Aurora](https://aws.amazon.com/about-aws/whats-new/2015/12/amazon-aurora-now-supports-encryption-at-rest/)), then enable that to secure data on disk. Make sure all backups are stored encrypted as well.

* 

Use minimal privilege for the database access user account. Don’t use the database root account and check for unused accounts and accounts with bad passwords.

* 

Fully prevent SQL injection by only using SQL prepared statements. For example: if using NPM, don’t use npm-mysql, use npm-mysql2 which supports prepared statements.

## ****Apps****

* 

Secure development systems with equal vigilance to what you use for production systems. Build the software from secured, isolated development systems.

* 

Ensure that all components of your software are scanned for vulnerabilities for every version pushed to production. This means O/S, libraries and packages. This should be automated into the [CI-CD](https://en.wikipedia.org/wiki/CI/CD) process.

* 

Do client-side input validation for quick user feedback, but never trust it. Always validate and encode user input before displaying.

* 

Validate every last bit of user input using white lists on the server. Consider generating validation code from API specifications using a tool like [Swagger](https://swagger.io), it is more reliable than hand-generated code.

* 

Never directly inject user content into responses. Never use untrusted user input in SQL statements or other server-side logic.

* 

Use centralized logging for all apps, servers and services. You should never need SSH to access or retrieve logs.

* 

Log with sufficient detail to diagnose all operational and security issues and NEVER log sensitive or personal information. Consider creating logs in JSON with [high cardinality](https://www.honeycomb.io/blog/2018/03/observability-a-manifesto/) fields rather than flat text lines.

* 

Don't emit revealing error details or stack traces to users and don't deploy your apps to production with DEBUG enabled.

## ****APIs****

* 

Ensure that users are fully authenticated and authorized appropriately when using your APIs.

* 

Ensure that no resources are enumerable in your public APIs. For IDs, consider using [RFC 4122 compliant UUIDs](http://www.ietf.org/rfc/rfc4122.txt) instead of integers. For node, see [NPM uuid](https://www.npmjs.com/package/uuid).

* 

Use canary checks in APIs to detect illegal or abnormal requests that indicate attacks.

## ****Network Traffic****

* 

Segment your network and protect sensitive services. Use firewalls, virtual private networks and cloud Security Groups to restrict and control inbound and outbound traffic to/from appropriate destinations. [AWS](https://aws.amazon.com/elasticloadbalancing/features/#Details_for_Elastic_Load_Balancing_Products) and [CloudFlare](https://www.cloudflare.com/waf/) both have excellent offerings.

* 

Use TLS for the entire site, not just login forms and responses. Never use TLS for just the login form. Transitionally, use the strict-transport-security header to force HTTPS on all requests.

* 

Cookies must be httpOnly and secure and be scoped by path and domain.

* 

Use [CSP](https://en.wikipedia.org/wiki/Content_Security_Policy) without allowing unsafe-\* backdoors. It is a pain to configure, but worthwhile. Use CSP [Subresource Integrity](https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity) for CDN content.

* 

Use X-Frame-Option, X-XSS-Protection headers in client responses. Use <https://observatory.mozilla.org> to score your site.

* 

Use HSTS responses to force TLS only access. Redirect all HTTP request to HTTPS on the server as backup.

* 

Use CSRF tokens in all forms and use the new [SameSite Cookie](https://scotthelme.co.uk/csrf-is-dead/) response header which fixes CSRF once and for all newer browsers.

* 

Remove other identifying headers that can make a hackers job easier of identifying your stack and software versions.

* 

Don't use GET requests with sensitive data or tokens in the URL as these will be logged on servers and proxies.

## ****Cloud Configuration****

* 

Ensure all services have minimum ports open. While security through obscurity is no protection, using non-standard ports will make it a little bit harder for attackers.

* 

Host backend database and services on private VPCs that are not visible on any public network. Be very careful when configuring AWS security groups and peering VPCs which can inadvertently make services visible to the public.

* 

Create test and staging resources in a separate AWS account to that used by production resources.

* 

Isolate logical services in separate VPCs and peer VPCs to provide inter-service communication.

* 

Ensure all services only accept data from a minimal set of IP addresses.

* 

Restrict outgoing IP and port traffic to minimize APTs and “botification”.

* 

Always use AWS IAM roles and not root credentials.

* 

Use minimal access privilege for all ops and developer staff.

* 

Regularly rotate passwords and access keys according to a schedule.

## ****Infrastructure****

* 

Ensure you can do upgrades without downtime. Ensure you can quickly update software in a fully automated manner.

* 

Create all infrastructure using a tool such as [Terraform](https://www.terraform.io), and not via the cloud console. Infrastructure should be defined as “code” and be able to be recreated at the push of a button. Have zero tolerance for any resource created in the cloud by hand — Terraform can then audit your configuration.

* 

Don’t SSH into services except for one-off diagnosis. Using SSH regularly, typically means you have not automated an important task.

* 

Don’t keep port 22 open on any AWS service groups on a permanent basis. If you must use SSH, only use public key authentication and not passwords.

* 

Create [immutable hosts](http://chadfowler.com/2013/06/23/immutable-deployments.html) instead of long-lived servers that you patch and upgrade. (See [Immutable Infrastructure Can Be More Secure](https://www.powerdown.io/blog/posts/stories/immutable-infrastructure-can-be-dramatically-more-secure.html)).

* 

If not using Immutable Infrastructure (bad), ensure you have an automated system to patch and update all servers and regularly update your AMIs and rotate your servers to prevent long-lived APTs.

* 

Power off unused services and servers. The most secure server is one that is powered down. Schedule dev servers to be [powered down](https://www.powerdown.io) after hours when not required.

* 

Use an [Intrusion Detection System](https://en.wikipedia.org/wiki/Intrusion_detection_system) to minimize [APTs](https://en.wikipedia.org/wiki/Advanced_persistent_threat).

## ****Denial of Service Protection****

* 

Make sure that DOS attacks on your APIs won’t cripple your site. At a minimum, have rate limiters on your slower API paths and authentication related APIs like login and token generation routines. Consider CAPTCHA on front-end APIs to protect back-end services against DOS.

* 

Enforce sanity limits on the size and structure of user submitted data and requests.

* 

Perform [Chaos testing](https://boyter.org/2016/07/chaos-testing-engineering/) to determine how your service behaves under stress.

* 

Consider using [Distributed Denial of Service](https://en.wikipedia.org/wiki/Denial-of-service_attack) (DDOS) mitigation via a global caching proxy service like [CloudFlare](https://www.cloudflare.com/). This can be turned on if you suffer a DDOS attack and otherwise function as your DNS lookup.

## ****Hack Yourself****

* 

Audit your design and implementation.

* 

Do penetration testing — hack yourself, but also have someone other than you do pen testing as well.

* 

Proactively test your app beyond normal use. Consider the [OWASP test checklist](https://github.com/0xRadi/OWASP-Web-Checklist) to guide your test hacking.

## ****Incident Response****

* 

Train staff (especially senior staff) as to the dangers and techniques used in security social engineering.

* 

Have a threat model that describes what you are defending against. It should list and prioritize the possible threats and actors.

* 

Setup a standard email account and web page dedicated for users to report security issues ([security@example.com](mailto:security@example.com) and /security).

* 

Have a practiced security incident plan. One day, you will need it.

## Security is a Journey

Most of all, remember that security is a journey and cannot be "baked-in" to the product just before shipping. I hope this checklist will prompt you through your entire development lifecycle to improve the security of your services.

**Penetration Testing**

**Common Security Protocols**