Securing Web Application Technologies (SWAT) CHECKLIST

The SWAT Checklist provides an easy-to-reference set of best practices that raise awareness and help development teams create more secure applications. It's a first step toward building a base of security knowledge around web application security. Use this checklist to identify the minimum standard that is required to neutralize vulnerabilities in your critical applications.

ERROR HANDLING AND LOGGING

BEST PRACTICE	DESCRIPTION	CWE ID
Display generic error messages	Error messages should not reveal details about the internal state of the application. For example, file system path and stack information should not be exposed to the user through error messages.	CWE-209
No unhandled exceptions	Given the languages and frameworks in use for web application development, never allow an unhandled exception to occur. Error handlers should be configured to handle unexpected errors and gracefully return controlled output to the user.	CWE-391
Suppress framework-generated errors	Your development framework or platform may generate default error messages. These should be suppressed or replaced with customized error messages, as framework-generated messages may reveal sensitive information to the user.	CWE-209
Log all authentication and validation activities	Log any authentication and session management activities along with all input validation failures. Any security-related events should be logged. These may be used to detect past or in-progress attacks.	CWE-778
Log all privilege changes	Any activities or occasions where the user's privilege level changes should be logged.	CWE-778
Log administrative activities	Any administrative activities on the application or any of its components should be logged.	CWE-778
Log access to sensitive data	Any access to sensitive data should be logged. This is particularly important for corporations that have to meet regulatory requirements like HIPAA, PCI, or SOX.	CWE-778
Do not log inappropriate data	While logging errors and auditing access are important, sensitive data should never be logged in an unencrypted form. For example, under HIPAA and PCI, it would be a violation to log sensitive data into the log itself unless the log is encrypted on the disk. Additionally, it can create a serious exposure point should the web application itself become compromised.	CWE-532
Store logs securely	Logs should be stored and maintained appropriately to avoid information loss or tampering by intruders. Log retention should also follow the retention policy set forth by the organization to meet regulatory requirements and provide enough information for forensic and incident response activities.	CWE-533

DATA PROTECTION

BEST PRACTICE	DESCRIPTION	CWE ID
Use HTTPS everywhere	Ideally, HTTPS should be used for your entire application. If you have to limit where it's used, then HTTPS must be applied to any authentication pages as well as to all pages after the user is authenticated. If sensitive information (e.g., personal information) can be submitted before authentication, those features must also be sent over HTTPS. Always link to the HTTPS version of URL if available. Relying on redirection from HTTP to HTTPS increases the opportunity for an attacker to insert a man-in-the-middle attack without raising the user's suspicion. **EXAMPLE:** sslstrip	CWE-311 CWE-319 CWE-523
Disable HTTP access for all protected resources	For all pages requiring protection by HTTPS, the same URL should not be accessible via the insecure HTTP channel.	CWE-319
Use strong TLS configurations	Weak ciphers must be disabled on all servers. For example, SSL v2, SSL v3, and TLS protocols prior to 1.2 have known weaknesses and are not considered secure. Additionally, disable the NULL, RC4, DES, and MD5 cipher suites. Ensure all key lengths are greater than 128 bits, use secure renegotiation, and disable compression. **EXAMPLE:** Qualys SSL Labs	
Use the Strict-Transport-Security header	The Strict-Transport-Security header ensures that the browser does not talk to the server over HTTP. This helps reduce the risk of HTTP downgrade attacks as implemented by the sslsniff tool.	
Store user passwords using a strong, iterative, salted hash	User passwords must be stored using secure hashing techniques with strong algorithms like PBKDF2, bcrypt, or SHA-512. Simply hashing the password a single time does not sufficiently protect the password. Use adaptive hashing (a work factor), combined with a randomly generated salt for each user to make the hash strong. **EXAMPLE:** https://haveibeenpwned.com	CWE-257
Securely exchange encryption keys	If encryption keys are exchanged or pre-set in your application, then any key establishment or exchange must be performed over a secure channel.	
Set up secure key management processes	When keys are stored in your system they must be properly secured and only accessible to the appropriate staff on a need-to-know basis. **EXAMPLE:** AWS Key Management Service (KMS), Azure Key Vault, AWS CloudHSM	CWE-320
Use valid HTTPS certificates from a reputable certificate authority	HTTPS certificates should be signed by a reputable certificate authority. The name on the certificate should match the FQDN of the website. The certificate itself should be valid and not expired. **EXAMPLE:** Let's Encrypt https://letsencrypt.org	
Disable data caching using cache control headers and autocomplete	Browser data caching should be disabled using the cache control HTTP headers or meta tags within the HTML page. Additionally, sensitive input fields, such as the login form, should have the autocomplete attribute set to off in the HTML form to instruct the browser not to cache the credentials.	CWE-524
Encrypt sensitive data at rest	Encrypt sensitive or critical data before storage.	CWE-311 CWE-312
Limit the use and storage of sensitive data	Conduct an evaluation to ensure that sensitive data elements are not being unnecessarily transported or stored. Where possible, use tokenization to reduce data exposure risks.	

CONFIGURATION AND OPERATIONS

BEST PRACTICE	DESCRIPTION	CWE ID
Automate application deployment	Automating the deployment of your application, using Continuous Integration and Continuous Deployment, helps to ensure that changes are made in a consistent, repeatable manner in all environments.	
Establish a rigorous change management process	A rigorous change management process must be maintained during operations. For example, new releases should only be deployed after proper testing and associated documentation has been completed. **EXAMPLE:** DevOps Audit Defense Toolkit https://itrevolution.com/devops-audit-defense-toolkit	CWE-439
Define security requirements	Engage the business owner to define security requirements for the application. This includes items that range from the whitelist validation rules all the way to nonfunctional requirements like the performance of the login function. Defining these requirements up front ensures that security is baked into the system.	
Conduct a design review	Integrating security into the design phase saves money and time. Conduct a risk review with security professionals and threat model the application to identify key risks. This helps you integrate appropriate countermeasures into the design and architecture of the application.	CWE-701 CWE-656
Perform code reviews	Security-focused code reviews can be one of the most effective ways to find security bugs. Regularly review your code looking for common issues like SQL Injection and Cross-Site Scripting. Leverage automated tools to maximize breadth of coverage and consistency.	CWE-702
Perform security testing	Conduct security testing both during and after development to ensure the application meets security standards. Testing should also be conducted after major releases to ensure vulnerabilities did not get introduced during the update process. Leverage automation by including security tests into the CI/CD pipeline.	
Harden the infrastructure	All components of infrastructure that support the application should be configured according to security best practices and hardening guidelines. In a typical web application this can include routers, firewalls, network switches, operating systems, web servers, application servers, databases, and application frameworks.	CWE-15 CWE-656
Define an incident handling plan	An incident handling plan should be drafted and tested on a regular basis. The contact list of people to involve in a security incident related to the application should be well defined and kept up to date.	
Educate the team on security	Training helps define a common language that the team can use to improve the security of the application. Education should not be confined solely to software developers, testers, and architects. Anyone associated with the development process, such as business analysts and project managers, should all have periodic software security awareness training.	

AUTHENTICATION

BEST PRACTICE	DESCRIPTION	CWE ID
Don't hardcode credentials	Never allow credentials to be stored directly within the application code. While it can be convenient to test application code with hardcoded credentials during development, this significantly increases risk and should be avoided. **EXAMPLE:** Hard-coded passwords in networking devices https://www.us-cert.gov/control_systems/pdf/ICSA-12-243-01.pdf	CWE-798
Develop a strong password reset system	Password reset systems are often the weakest link in an application. These systems are often based on users answering personal questions to establish their identity and in turn reset the password. The system needs to be based on questions that are both hard to guess and brute force. Additionally, any password reset option must not reveal whether or not an account is valid, preventing username harvesting. **EXAMPLE:** Sara Palin password hack http://en.wikipedia.org/wiki/Sarah_Palin_email_hack	CWE-640
Implement a strong password policy	A password policy should be created and implemented so that passwords meet specific strength criteria. **EXAMPLE:** https://pages.nist.gov/800-63-3/sp800-63-3.html	CWE-521
Implement account lockout against brute-force attacks	Account lockout needs to be implemented to prevent brute-force attacks against both the authentication and password reset functionality. After several tries on a specific user account, the account should be locked for a period of time or until it is manually unlocked. Additionally, it is best to continue the same failure message indicating that the credentials are incorrect or the account is locked to prevent an attacker from harvesting usernames.	CWE-307
Don't disclose too much information in error messages	Messages for authentication errors must be clear and, at the same time, be written so that sensitive information about the system is not disclosed. For example, error messages that reveal that the user id is valid but that the corresponding password is incorrect confirm to an attacker that the account does exist on the system.	
Store database credentials securely	Modern web applications usually consist of multiple layers. The business logic tier (processing of information) often connects to the data tier (database). Connecting to the database, of course, requires authentication. The authentication credentials in the business logic tier must be stored in a centralized location that is locked down. Scattering credentials throughout the source code is not acceptable. Some development frameworks provide a centralized secure location for storing credentials to the backend database. These encrypted stores should be leveraged when possible.	CWE-257
Applications and middleware should run with minimal privileges	If an application becomes compromised it is important that the application itself and any middleware services be configured to run with minimal privileges. For instance, while the application layer or business layer need the ability to read and write data to the underlying database, administrative credentials that grant access to other databases or tables should not be provided.	CWE-250

SESSION MANAGEMENT

BEST PRACTICE	DESCRIPTION	CWE ID
Ensure that session identifiers are sufficiently random	Session tokens must be generated by secure random functions and must be of sufficient length to withstand analysis and prediction.	CWE-6
Regenerate session tokens	Session tokens should be regenerated when the user authenticates to the application and when the user privilege level changes. Additionally, should the encryption status change, the session token should always be regenerated.	CWE-384
Implement an idle session timeout	When a user is not active, the application should automatically log the user out. Be aware that Ajax applications may make recurring calls to the application, effectively resetting the timeout counter automatically.	CWE-613
Implement an absolute session timeout	Users should be logged out after an extensive amount of time (e.g., 4-8 hours) has passed since they logged in. This helps mitigate the risk of an attacker using a hijacked session.	CWE-613
Destroy sessions at any sign of tampering	Unless the application requires multiple simultaneous sessions for a single user, implement features to detect session cloning attempts. Should any sign of session cloning be detected, the session should be destroyed, forcing the real user to reauthenticate.	
Invalidate the session after logout	When the user logs out of the application, the session and corresponding data on the server must be destroyed. This ensures that the session cannot be accidentially revived.	CWE-613
Place a logout button on every page	The logout button or logout link should be easily accessible to users on every page after they have authenticated.	
Use secure cookie attributes	The session cookie should have the HttpOnly, Secure, and SameSite flags set. This ensures that the session id will not be accessible to client-side scripts, will only be transmitted over HTTPS, and will only be sent with requests from the same site (mitigates CSRF).	CWE-79 CWE-614
Set the cookie domain and path correctly	The cookie domain and path scope should be set to the most restrictive settings for your application. Any wildcard domain scoped cookie must have a good justification for its existence.	
Use non-persistent cookies	If a cookie has the "Max-Age" or "Expires" attributes the browser treats it as a persistent cookie and stores it to disk until the expiration time. Do not do this for session cookies.	

INPUT AND OUTPUT HANDLING

BEST PRACTICE	DESCRIPTION	CWE ID
Conduct contextual output encoding	All output functions must contextually encode data before sending the data to the user. Depending on where the output will end up in the HTML page, the output must be encoded differently. For example, data placed in the URL context must be encoded differently than data placed in a JavaScript context within the HTML page. **EXAMPLE:** Resource: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet	CWE-79
Prefer whitelists over blacklists	For each user input field, there should be validation on the input content. Whitelisting input is the preferred approach. Only accept data that meet a certain criteria. For input that needs more flexibility, blacklisting can also be applied where known bad input patterns or characters are blocked.	CWE-159 CWE-144
Use parameterized SQL queries	SQL queries should be crafted with user content passed into a bind variable. Queries written this way are safe against SQL injection attacks. SQL queries should not be created dynamically using string concatenation. Similarly, the SQL query string used in a bound or parameterized query should never be dynamically built from user input. **EXAMPLE:** Sony SQL injection hack http://www.infosecurity-magazine.com/view/27930/lulzsec-sony-pictures-hackers-were-school-chums	CWE-89 CWE-564
Prevent Insecure Deserialization	Do not accept serialized objects from untrusted sources, define known good data types when deserializing data, and implement integrity checks on serialized objects.	CWE-502
Use tokens to prevent forged requests	In order to prevent Cross-Site Request Forgery attacks, you must embed a random value that is not known to third parties into the HTML form. This CSRF protection token must be unique to each request. This prevents a forged CSRF request from being submitted because the attacker does not know the value of the token.	CWE-352
Set the encoding for your application	For every page in your application, set the encoding using HTTP headers or meta tags within HTML. This ensures that the encoding of the page is always defined and that the browser will not have to determine the encoding on its own. Setting a consistent encoding like UTF-8 for your application reduces the overall risk of issues like Cross-Site Scripting.	CWE-172
Validate uploaded files	When accepting file uploads from the user make sure to validate the size of the file, the file type, and the file contents, and ensure that it is not possible to override the destination path for the file.	CWE-434 CWE-616 CWE-22
Use the nosniff header for uploaded content	When hosting user uploaded content that can be viewed by other users, use the X-Content-Type-Options: nosniff header so that browsers do not try to guess the data type. Sometimes the browser can be tricked into displaying the data type incorrectly (e.g., showing a GIF file as HTML). Always let the server or application determine the data type.	CWE-430
Prevent tabnabbing	Use the "rel" anchor tag attribute with values of "noopener" or "noreferrer" to prevent an opened tab from tampering with the calling tabs location in the browser. In JavaScript this can be prevented by setting window.opener to null.	CWE-1022
Validate the source of input	The source of the input must be validated. For example, if input is expected from a POST request, do not accept the input variable from a GET request.	CWE-20 CWE-346
X-Frame-Options or CSP headers	Use the X-Frame-Options header or Content-Security-Policy (CSP) header frame-ancestors directive to prevent content from being loaded by a foreign site in a frame. This mitigates Clickjacking attacks. For older browsers that do not support this header add framebusting Javascript code to mitigate Clickjacking (although this method is not foolproof and can be circumvented).	CAPEC-103 CWE-693
Use secure HTTP response headers	The Content Security Policy (CSP), X-XSS-Protection, and Public-Key-Pins headers help defend against Cross-Site Scripting (XSS) and Man-in-the-Middle (MITM) attacks. **EXAMPLE:** OWASP Secure Headers Project https://www.owasp.org/index.php/OWASP_Secure_Headers_Project	CWE-79 CWE-692

ACCESS CONTROL

BEST PRACTICE	DESCRIPTION	CWE ID
Apply access control checks consistently	Always apply the principle of complete mediation, forcing all requests through a common security "gate keeper." This ensures that access control checks are triggered whether or not the user is authenticated.	CWE-284
Apply the principle of least privilege	Use a Mandatory Access Control system. All access decisions will be based on the principle of least privilege. If not explicitly allowed then access should be denied. Additionally, after an account is created, rights must be specifically added to that account to grant access to resources.	CWE-272 CWE-250
Don't use direct object references for access control checks	Do not allow direct references to files or parameters that can be manipulated to grant excessive access. Access control decisions must be based on the authenticated user identity and trusted server-side information.	CWE-284
Don't use unvalidated forwards or redirects	An unvalidated forward can allow an attacker to access private content without authentication. Unvalidated redirects allow an attacker to lure victims into visiting malicious sites. Prevent this from occurring by conducting the appropriate access control checks before sending the user to the given location.	CWE-601

SANS Secure DevOps

POSTER

Securing Web Application Technologies (SWAT)

CHECKLIST

Version 1.6

AND

Secure DevOps Practices

Ingraining security into the mind of every developer.

software-security.sans.org

APSPS_SEC540_v1.6_1-19

SANS Secure DevOps

CURRICULUM

APPLICATION SECURITY

- **DEV522** Defending Web Applications Security Essentials GWEB
- **DEV531** Defending Mobile Applications Security Essentials
- **DEV541** Secure Coding in Java/JEE GSSP-JAVA
- **DEV544** Secure Coding in .NET GSSP-NET

SECURE DEVOPS AND CLOUD

- **SEC540** Cloud and DevOps Security Automation
- **SEC545** Cloud Security Architecture and Operations
- **SEC524** Cloud Security and Risk Fundamentals
- **SEC534** Secure DevOps: A Practical Introduction

AWARENESS & TESTING

- **SSA.DEVELOPER** Application Security Awareness Modules
- **SEC542** Web App Penetration Testing and Ethical Hacking GWAPT
- **SEC642** Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques
- AppSec CyberTalent Assessment sans.org/appsec-assessment

Website
software-security.sans.org
Free resources, white papers, webcasts, and more

Twitter
@sansappsec
Latest news, promos, and other information

Blog
software-security.sans.org/blog

Cloud Security Top Ten

1 Insecure use of Developer Credentials
Developer credentials allow your team and integrations access to your account. They should be stored and used securely to ensure that only authorized individuals and use-cases have access. When possible considering tracking and auto-expiring credentials after a set period of time or inactivity.

2 Publicly Accessible Storage
Cloud providers have several different methods of storing objects and data. Regularly review your configurations to ensure that only the intended components are publicly accessible.

3 Improper use of Default Configurations
Cloud providers pre-configure common access control policies. These can be convenient, but often introduce risk as provider's service offerings change. Pre-configured rules often change to introduce access to new services outside the context of what is actually needed or being used.

4 Broken Access Control
Principles of least privilege should be followed when architecting access to cloud services. Consider the granularity of access to services, systems, and the network. Regularly or automatically review this access to ensure that least privilege is being followed.

5 Misconfigured Network Constructs
Most cloud providers have sophisticated methods to control network access beyond simple IP address based rules. Consider using these constructs for controlling access at a granular level. Consider using cloud provider based network components to segment traffic thoughtfully.

6 Inadequate Monitoring and Logging
Turn on and regularly monitor API access logging. Consider a risk based logging strategy for services which are not logged by way of these core logging services.

7 Lack of Inventory Management
API based access solves a lot of inventory management problems. Consider strategies to enrich your environment with additional information around ownership, use-case, and sensitivity.

8 Domain Hijacking
Transitive-trust often exists between cloud services and DNS entries. Regularly review your DNS and cloud configurations to prevent take-over situations.

9 Lack of a Disaster Recovery Plan
Cloud environments do not automatically solve DR concerns. Consider what level of investment is appropriate for catastrophic events within your cloud environment. Design a DR program to recover from outside accounts, providers, or locales.

10 Manual Account Configuration
Doing things by hand limits your ability to scale and leverage cloud-native security tools and controls. Consider "security-as-code" and automation as your best friends within cloud environments.

Contributors: Ben Hagen | Mark Hillick | Will Bengston | Steve Woodrow | Thomas Vachon

Serverless Security Top Ten

1 Function Event Data Injection
Serverless architectures provide a multitude of event sources, which can trigger the execution of a serverless function. These functions can consume input from each type of event source, and such event input might include different message formats, depending on the type of event and its source. The various parts of these event messages can contain attacker-controlled or otherwise dangerous inputs.

2 Broken Authentication
Serverless architectures promote a microservices-oriented system design and are composed of functions that are weaved together and orchestrated to form the overall system logic. Some serverless functions may expose public web APIs, while others may consume events of different source types, such as cloud storage events, NoSQL database events, IoT device telemetry signals or even SMS message notifications. Apply robust authentication schemes, which provide access control and protection, to all relevant functions, event types and triggers.

3 Insecure Serverless Deployment Configuration
Cloud services in general, and serverless architectures in particular offer many customizations and configuration settings in order to adapt them for each specific need, task or surrounding environment. Some of these configuration settings have critical implications on the overall security posture of the application and should be given attention. Do not rely on the default settings provided by serverless architecture vendors.

4 Over-Privileged Function Permissions and Roles
Serverless applications should always follow the principle of "least privilege" . This means that a serverless function should be given only those privileges, which are essential in order to perform its intended logic. In a system where all functions share the same set of over-privileged permissions, a vulnerability in a single function can eventually escalate into a system-wide security catastrophe.

5 Inadequate Function Monitoring and Logging
Augment basic or out-of-the-box logging configurations to provide a full security event audit trail. This should includes items such as successful/failed API access key use, attempts to invoke serverless functions with inadequate permissions, successful/failed deployment of new serverless functions or configurations, changes to function permissions or execution roles, anomalous interaction or irregular flow between serverless functions, outbound connections initiated by serverless functions, and execution of serverless functions or access to data from an external third-party account not related to the main account.

6 Insecure Third-Party Dependencies
Define a process for maintaining an inventory list of software packages and other dependencies and their versions, scanning software for known vulnerable dependencies, removing unnecessary dependencies, and upgrading deprecated package versions to the latest versions and applying all relevant software patches.

7 Insecure Application Secrets Storage
Store all application secrets in secure encrypted storage and ensure that encryption keys are maintained via a centralized encryption key management infrastructure or service. Such services are offered by most serverless architecture and cloud vendors, who also provide developers with secure APIs that can easily and seamlessly integrate into serverless environments.

8 Denial of Service and Financial Resource Exhaustion
Serverless architecture vendors define default limits on the execution of serverless functions. Depending on the type of limit and activity, poorly designed or configured applications may be abused in such a way that will eventually cause latency to become unacceptable or even render it unusable for other users. Additionally, an attacker may push the serverless application to "over-execute" for long periods of time, essentially inflating the monthly bill and inflicting a financial loss for the target organization.

9 Serverless Function Execution Flow Manipulation
Manipulation of application flow may help attackers to subvert application logic. Using this technique, an attacker may sometimes bypass access controls, elevate user privileges or even mount a Denial of Service attack. In a system where multiple functions exist, and each function may invoke another function, the order of invocation might be critical for achieving the desired logic. Moreover, the design might assume that certain functions are only invoked under specific scenarios and only by authorized invokers. Make sure that proper access controls and permissions are set for each function, and where applicable, use a robust application state management facility.

10 Improper Exception Handling and Verbose Error Messages
Options for performing line-by-line debugging of serverless based applications is rather limited and more complex compared to the debugging capabilities that are available when developing standard applications. This forces some developers to adopt the use of verbose error messages, enable debugging environment variables and eventually forget to clean the code when moving it to the production environment. Verbose error messages such as stack traces or syntax errors, which are exposed to end users, may reveal details about the internal logic of the serverless function, and in turn reveal potential weaknesses, flaws or even leak sensitive data. If your serverless environment supports defining custom error responses, such as those provided by API gateways, create simple error messages that do not reveal any details about the internal implementation or any environment variables.

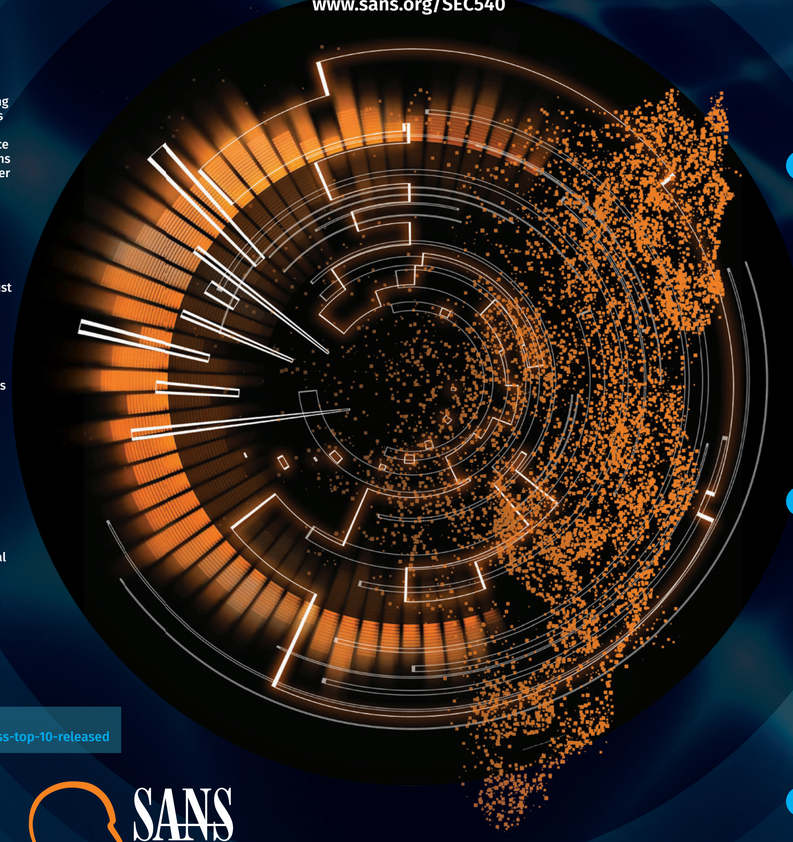
Contributed by Ory Segal and PureSec
https://www.puresec.io/blog/serverless-top-10-released

Secure DevOps Practices

Learn to build, deliver, and deploy modern applications using secure DevOps and cloud principles, practices, and tools.

SEC540: Cloud and DevOps Security Automation

www.sans.org/SEC540



Building a DevSecOps Program (CALMS)

Culture
Break down barriers between Development, Security, and Operations through education and outreach

Automation
Embed self-service automated security scanning and testing in continuous delivery

Lean
Value stream analysis on security and compliance processes to optimize flow

Measurement
Use metrics to shape design and drive decisions

Sharing
Share threats, risks, and vulnerabilities by adding them to engineering backlogs

Start Your DevOps Metrics Program

- Number of high-severity vulnerabilities and how long they are open
- Build and deployment cycle time
- Automated test frequency and coverage
- Scanning frequency and coverage
- Number of attacks (and attackers) hitting your application

First Steps in Automation

- Build a security smoke test (e.g., ZAP Baseline Scan)
- Conduct negative unit testing to get off of the happy path
- Attack your system before somebody else does (e.g., Gauntlt)
- Add hardening steps into configuration recipes (e.g., dev-sec.io)
- Harden and test your CI/CD pipelines and do not rely on developer-friendly defaults

SANS Secure DevOps CURRICULUM

APPLICATION SECURITY

DEV522 Defending Web Applications Security Essentials GWEB

DEV531 Defending Mobile Applications Security Essentials

DEV541 Secure Coding in Java/JEE GSSP-JAVA

DEV544 Secure Coding in .NET GSSP-NET

SECURE DEVOPS AND CLOUD

SEC540 Cloud and DevOps Security Automation

SEC545 Cloud Security Architecture and Operations

SEC524 Cloud Security and Risk Fundamentals

SEC534 Secure DevOps: A Practical Introduction

AWARENESS & TESTING

SSA.DEVELOPER Application Security Awareness Modules

SEC542 Web App Penetration Testing and Ethical Hacking GWAPT

SEC642 Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques

AppSec CyberTalent Assessment
sans.org/appsec-assessment

Website
software-security.sans.org
Free resources, white papers, webcasts, and more

Twitter
@sansappsec
Latest news, promos, and other information

Blog
software-security.sans.org/blog

Secure DevOps Toolchain

Pre-Commit
Security activities before code is checked in to version control

Threat Modeling/Attack Mapping:
- Attacker personas
- Evil user stories
- Raindance
- Mozilla Rapid Risk Assessment
- OWASP ThreatDragon
- SAFECode Tactical Threat Modeling
- Slack goSDL
- ThreatPlaybook

Security and Privacy Stories:
- OWASP ASVS
- SAFECode Security Stories

Pre-Commit Security Hooks:
- git-hound
- git-secrets
- Repo-supervisor
- ThoughtWorks Talisman

IDE Security Plugins:
- DevSkim
- FindSecurityBugs
- Puma Scan
- SonarLint

Manual and Peer Reviews:
- Gerrit
- GitHub pull request
- GitLab merge request
- Review Board

Secure Coding Standards:
- CERT Secure Coding Standards
- OWASP Proactive Controls
- SAFECode Fundamental Practices for Secure Software Development

Commit (Continuous Integration)
Fast, automated security checks during the build and Continuous Integration steps

Static Code Analysis (SCA):
- Brakeman
- ESLint
- FindSecurityBugs
- NodeJsScan
- Phan

Security Unit Tests:
- JUnit
- Mocha
- xUnit

Infrastructure as Code Analysis:
- ansible-lint
- cfn_nag
- cookstyle
- Foodcritic
- puppet-lint

Container Hardening:
- Bane
- CIS Benchmarks
- grsecurity

Dependency Management:
- Bundler-Audit
- Github security alerts
- Node Security Platform
- PHP Security Checker
- Retire.JS
- OWASP Dependency Check
- Terrascan

Container Security:
- Actuary
- Anchore
- Clair
- Dagda
- Docker Bench
- kube-bench
- kube-hunter
- Falco

Acceptance (Continuous Delivery)
Automated security acceptance, functional testing, and deep out-of-band scanning during Continuous Delivery

Infrastructure as Code:
- Ansible
- Chef
- Puppet
- SaltStack
- Terraform
- Vagrant

Immutable Infrastructure:
- Docker
- rkt

Security Scanning:
- Arachni
- nmap
- sqlmap
- ssh_scan
- sslyze
- ZAP

Cloud Configuration Management:
- AWS CloudFormation
- Azure Resource Manager
- Google Cloud Deployment Manager

Security Acceptance Testing:
- BDD-Security
- Gauntlt
- Mittn

Infrastructure Tests:
- CIS
- Serverspec
- Test Kitchen

Infrastructure Compliance Checks:
- HubbleStack
- InSpec

Production (Continuous Deployment)
Security checks before, during, and after code is deployed to production

Security Smoke Tests:
- ZAP Baseline Scan
- nmap
- ssllabs-scan

Cloud Secrets Management:
- AWS KMS
- AWS Secrets Manager
- Azure Key Vault
- Google Cloud KMS

Configuration Safety Checks:
- AWS Config
- AWS Trusted Advisor
- Microsoft Azure Advisor
- Security Monkey
- OSQuery

Cloud Security Testing:
- CloudSploit
- Nimbostratus

Secrets Management:
- Ansible Vault
- Blackbox
- Chef Vault
- CyberArk Conjur
- Docker Secrets
- Hashicorp Vault
- Pinterest Knox

Serverless Protection:
- FunctionShield

Server Hardening:
- CIS
- dev-sec.io
- SIMP

Host Intrusion Detection System (HIDS):
- fail2ban
- OSSEC
- Samhain
- Wazuh

Operations
Continuous security monitoring, testing, audit, and compliance checks

Fault Injection:
- Chaos Kong
- Chaos Monkey
- Infection Monkey
- pumba

Cyber Simulations:
- Game day exercises
- Tabletop scenarios

Continuous Scanning:
- Netflix Aardvark
- OpenSCAP
- OpenVAS
- Prowler
- Scout2
- vuls

Penetration Testing:
- Attack-driven defense
- Bug Bounties
- Red team exercises

Threat Intelligence:
- Diamond Model
- Kill Chain
- STIX
- TAXII

Cloud Compliance:
- Cloud Custodian
- Forseti Security
- Netflix Repokid
- CIS AWS Benchmark
- CIS Azure Benchmark

Blameless Postmortems:
- Etsy Morgue

Continuous Monitoring:
- ElastAlert
- grafana
- graphite
- prometheus
- seyren
- sof-elk
- statsd
- 411

Cloud Monitoring:
- CloudWatch
- CloudTrail
- Reddalert
- Azure Security Center

Poster contributors:
- Ben Allen
- Will Bengston
- Jim Bird
- David Deatherage
- Mark Geeslin
- Ben Hagen
- Mark Hillick
- Eric Johnson
- Frank Kim
- Jason Lam
- Gregory Leonard
- Ory Segal and PureSec
- Dr. Johannes Ullrich
- Thomas Vachon
- Steve Woodrow

SANS