WEB APPLICATION SECURITY STANDARD

The capitalized terms used herein are defined in the Web Application Security policy.

The requirements outlined in this document represent minimum baseline standards for the secure development, testing, and scanning of, and for established criticality and risk ratings for, University Web Applications.

1. Inventory – Risk, Criticality, Data Classification

1.1. Inventory. Web developers and responsible business units must maintain a current inventory of Web Applications. The inventory must include Web Application descriptions and data classification. The inventory may include Web Application authentication mechanisms, availability rating, overall criticality rating, and URLs for all the application environments (e.g., Dev, Test, Prod).

1.2. Criticality Classification. Classification is performed initially by web developers in coordination with the business unit responsible for the application. An initial evaluation based on Data Classification and availability requirements will result in an overall criticality rating for the Web Application. If consensus between web developers and the business unit on a criticality rating cannot be reached, the Chief Information Officer ("CIO"), or their designee, and/or the IT and Data Governance Trustees should be consulted. The following criticality table represents the model to follow:

Criticality	Data Classification*		Availability Rating
Very High Risk	Level 4 – Highly Sensitive Data	or	Tier 1 – Mission Critical
High Risk	Level 3 – Sensitive Data	or	Tier 2 – Enterprise Applications
Low Risk	Level 2 – Internal Data	or	Tier 3 – Internal Only
Very Low Risk	Level 1 – Public Data	or	Tier 4 – All other

*The University's data classifications are set forth in the Data Classification and Handling policy

Availability ratings are a business-based classification determined by the importance of an application's position in business continuity. "Tier 1 – mission critical" Web Applications represent core functions that if unavailable would result in the University being unable to conduct business (*e.g.*, enterprise learning systems, payroll systems, student administration systems, and authentication systems that support other systems). "Tier 1 – mission critical" Web Applications additionally represent those applications that handle Highly Sensitive Data.

2. Secure Web Development

Use of secure development guidelines (*e.g.*, the Open Web Application Security Project also known as "OWASP") is essential to a secure Software Development Life Cycle ("SDLC"). Consider the following principles during application threat modeling and secure application development.

2.1. **Defense in depth**. Using layered security mechanisms increases security of the system as a whole. Security improves by ensuring additional controls are in place when controls in earlier stages of application security fail. Example: a web server is compromised by a previously undisclosed vulnerability. The web server attempts to connect to another near-by web server with the same vulnerability. However, a firewall blocks the outbound request.

2.2. **Use a positive security model**. When possible and practical, an "allow list" should be used. Unknown data formats or requests are checked against the allow list; if present, the request or data is processed as normal. Data or requests not on the allow list are denied. Positive security models apply to network architecture as well. Hosts or applications that employ allow lists are less susceptible to attack due to attack surface reduction.. Example: using input validation to ensure a parameter is an integer is establishing a positive security model. Non-integer values are rejected.

2.3. **Fail securely.** In general, applications should be able to handle undefined scenarios in a secure fashion. If an exception or software failure is experienced, request processing should complete in a fashion consistent with a deny action (*i.e.*, a safe code path should be present and taken). Example: a request is received to promote user x to administrator privilege. During verification of user x with an identity provider, an exception occurs. The request should not be completed since user x was not successfully verified.

2.4. **Run with least privilege.** The principle of least privilege recommends that accounts (including application accounts) have the least privilege required to perform their business processes. This encompasses user rights, resource permissions such as CPU limits, memory, network, and file system permissions. Example: a Web Application connects to a database to retrieve information. The Web Application has no business need to write to the database. The database account used by the web server should only have select access to the database, tables, and columns it requires to perform its work.

2.5. **Open design.** Secure components are resilient to direct analysis. Components designed with transparency in mind provide the strongest protections to threats by being known and verifiably secure. "Security through obscurity" is the opposite of an Open Design and implies that external threats are unaware of internal controls. Choosing an Open Design helps ensure that security is strong by being clear and apparent. Example: leaving a house key beneath the doormat is an example of security through obscurity. Anyone who lifts the mat will find the key. An open design may be to leave a house key in a clearly visible combination lock box. Authorized parties are given the combination to the lock box in order to recover the key.

2.6. **Keep security simple.** Related to open design, try to keep controls as simple as possible while still being effective. More code and complexity in a system increases the likelihood of bugs and vulnerabilities being present in the system. Using clear and concise controls reduces risk by being simple to implement and to utilize. Example: uninstalling unused services and software reduces the attack surface of a server and is a simple mechanism to achieve higher security.

2.7. **Detect intrusions.** Log all security relevant events. Establish monitoring procedures for these logs. Establish procedures to identify intrusions and respond appropriately. Example: an application that requires authentication has received 10,000 failed authentication attempts in a five-minute period from a single Internet Protocol address. This may be an attack or a misconfigured device. An alert should be generated, and an analyst should perform analysis to determine if the attack was successful or if support should be informed of the misconfiguration.

2.8. **Do not trust infrastructure.** By authenticating and authorizing requests, even from "trusted" infrastructure, trust relationships are not abused. Example: compromising a Real-time Transport Protocol ("RTP") server where minimum privilege is used should not allow a threat to "pivot" to a nearby web server that may have an implicit trust relationship with that RTP server. This web server should not trust the RTP server.

2.9. **Do not trust services.** A compromised service cannot be trusted. Data presented by a compromised service may be malicious. Example: when a database service is compromised, the data it provides to other services may contain malicious components that allow these data consuming services to be

compromised. These other services should not trust the database. Verifying data received from services ensures trust relationships are not abused.

2.10. Establish secure defaults. Related to using a positive security model, providing secure defaults ensures that security is considered. If a control allows users to "opt-out," this decision may be weighed by the user and chosen as their needs require. Initial configuration should always be as secure as possible.

2.11. Do not trust third-party source code. Using third-party code, such as via open-source projects and package/dependency management tools, introduces a potential attack vector that needs to be audited/monitored. If you are using package tools, use their built-in alerting/warning systems for compromised code and take the necessary precautions. Additionally, consider using monitoring services to receive warnings on packages during development and as a scheduled periodic check over your code base. When possible, code review third-party code.

3. Authentication and Authorization.

Ensuring appropriate access to Web Applications is a critical security component. Authentication and authorization help ensure the right user or client has access to the right resource at the right time.

3.1. Authentication. Web Applications must properly authenticate users through ITS supported central authentication systems. If supported Authentication systems cannot be used, an exception request must be made that details why the application cannot use existing systems (*e.g.,* a grant application that has predominantly non-University users).

3.2. Authorization. When possible, establish Authorizations for Applications by Affiliation, group membership, or employment status, rather than by individual assignment. If individual Authorizations are used, these should expire and require renewal on a periodic (at least annually) basis. Use central Authorization and group membership sources where possible, as opposed to groups that only exist within the application. This allows user access across the enterprise to be discovered and managed outside of the specific application. Centrally managed groups can be assigned roles within the application. If additional functionality is needed, coordinate with ITS. Document clear rules and processes for vetting and approving Authorizations. On at least an annual basis, review and promptly remove all Authorizations for individuals who have left the University, transferred to another department, or assumed new job duties within the department.

4. Vulnerability Testing. All Web Applications should undergo vulnerability testing. Vulnerability testing uses automated scanning tools and manual testing to identify known vulnerabilities, such as outdated software versions, misconfigurations, weak passwords, improper access controls, and improper inputs.

5. Web Application Security: A Web Application's Developer has primary responsibility for the security of the Web Application, including timely response and reporting of reasonably suspected or reported security issues and testing results. Remediation steps, and follow-up testing results, will be documented and, when necessary, attached to an associated change management record.

5.1. Exceptions Review. It may be necessary to postpone mitigation of known security vulnerabilities. If security mitigation cannot follow the set schedule, an exception request must be made that details why postponement or deferral is necessary. The CIO. or their designee, will promptly act on all exception request approvals, which must be submitted through the ServiceNow system. Exceptions may include:

- Production system freeze or semester start-up periods
- Conflicts with other critical changes scheduled during the same period
- Security testing is believed to break functionality or cause excessive system load
- Systems, applications, or devices where appropriate risk-mitigation controls are put in place, documented and validated
- Resources are unavailable to perform assessments
- In all cases of exception requests, the implementation of Web Application firewall should be considered as a mitigating control to potential threats

6. **Training.** In addition to the University's Information Security Awareness Training requirements, web developers should pursue or receive web development and secure coding training to ensure a baseline set of skills and knowledge for securing Web Applications. This training should include annual review of the OWASP guidelines and taking part in peer code reviews.