

STATE OF ALABAMA

Information Technology Guideline

GUIDELINE 661G1-01: APPLICATION SECURITY

Enforcing appropriate security standards, mitigating known vulnerabilities, and testing applications for security flaws before deployment is vital to ensuring the confidentiality, integrity, availability of State of Alabama production applications, application components, and data. The following guidelines provide best practice guidance regarding the controls used to ensure the defense of State of Alabama applications against common vulnerabilities, errors, and attack types including (but not limited to) SQL injection, input validation, and error handling, in addition to database security and controls on Mobile Code (e.g. JavaScript, ActiveX, Flash, etc).

OBJECTIVE:

Identify and correct security flaws in State of Alabama production applications, application components, and application source code.

SCOPE:

The guidelines in this document apply to all Executive Branch agencies, boards, and commissions except those exempt under The Code of Alabama 1975 (Title 41 Chapter 4 Article 11).

GUIDELINES:

DATABASE SECURITY

Malicious attacks against web applications accessing backend databases are a significant threat. One common method of attack, the Structured Query Language (SQL) injection, takes advantage of vulnerable applications and their fully-privileged access to the backend database providing malicious users the means to execute commands in the application's database and gain access to sensitive data.

The following guidelines should be used to secure information systems utilizing a database management system (DBMS). This general guidance may be applied to any DBMS; however, it is not intended to be used to configure database applications such as Microsoft Access that are designed to be used by a single user or a small number of users. Also, this guidance is meant for use in conjunction with the database host platform operating system (OS) baseline as well as other policies, standards, and guidelines related to the requirements of any applications accessing a database.

SECURITY DESIGN AND CONFIGURATION:

Procedural Review:

Ensure database security policies and procedures are reviewed at least annually and are current and consistent with current state policy, standards, and guidance, vendor-specific guidance and recommendations, and site-specific or other security policy.

Configuration Specifications:

Ensure the database is secured in accordance with the general security requirements specified in State IT policies, standards, and this guideline and with product-specific security guidance in this order of preference as available:

(1) Commercially available practices from independent security organizations such as and the National Institute of Standards and Technology (NIST), DISA, SANS, and the Center for Internet Security (CIS);

(2) Independent testing labs such as ICSA (<http://www.icsalabs.com>);

(3) Vendor security recommendations and literature.

Compliance Testing:

Ensure comprehensive testing plans and procedures for database installations, updates, and patches are defined, documented, and implemented before being deployed in a production environment.

Functional Architecture:

As an application, a secure DBMS implementation must be in accordance with a planned or designed usage or architecture. Whenever changes to the usage, environment, or configuration of a DBMS are made or considered, a review of the DBMS functional architecture needs to be completed.

Ensure unused optional database components or features, applications, and objects are removed from the database and host system. If the optional component cannot be uninstalled or removed, then the Database Administrator (DBA) will ensure the unused component or feature is disabled.

Ensure database applications, user accounts, and objects installed for demonstration of database features, experimentation, or other non-production support purposes have been removed from the database and host system.

Configure the database to disable access from the database to objects stored externally to the database on the local host unless mission and/or operationally required and documented in the functional architecture documentation.

Disable use of external procedures by the database unless mission and/or operationally required and documented in the functional architecture documentation.

Ensure database connections to remote databases or remote or external applications and services are disabled and/or not defined unless database replication is in use or the remote connection is mission and/or operationally required and documented in the functional architecture documentation.

Ensure use of credentials used to access remote databases or other applications are restricted to authorized database accounts and used only for mission and/or operationally required and documented purposes.

Ensure credentials stored in or used by the DBMS that are used to access remote databases or other applications are protected by encryption and access controls.

Ensure credentials used to access remote databases or other applications use fully qualified names, i.e., globally unique names that specify all hierarchical classification names, in the connection specification.

Ensure database accounts used for replication or distributed transactions are not granted DBA privileges.

Ensure OS accounts used for execution of external database procedures have the minimum OS privileges required assigned to them.

Ensure each database service or process runs under a custom, dedicated OS account that is assigned the minimum privileges required for operation where applicable.

Ensure database and host system listeners that provide configuration of network restrictions are configured to restrict network connections to the database to authorized network addresses and protocols.

Ensure all local and network-advertised named database services are uniquely and clearly identified.

Ensure all database application user roles and the privileges assigned to them are authorized by the Data Owner in the functional architecture documentation.

Ensure security requirements specific to the use of the database (e.g., encryption of sensitive data) are configured as identified in the System Security Plan.

Ensure all categories of sensitive data stored or processed by the database are identified in the functional architecture documentation.

Ensure the restoration priority of the database and its supporting subsystems are identified in the System Security Plan.

Non-repudiation:

Where encryption, digital signature, key exchange, or secure hashing is used ensure the application of cryptography complies with state encryption standards.

Partitioning the Application:

Ensure the DBMS host is dedicated to support of the DBMS and is not shared with other application services including web, application, file, print, or other services unless mission or operationally required and documented in the System Security Plan.

Install and maintain database software directories including DBMS configuration files in dedicated directories or disk partitions separate from the host OS and other applications.

Install and maintain database data directories including transaction log and audit files in dedicated directories or disk partitions separate from software or other application files.

Ensure DBMS data files that store DBMS system tables and other system objects dedicated to support the entire DBMS are not shared with data files used for storage of third-party application database objects.

Ensure database data files used by third-party applications are defined and dedicated for each application.

Ports, Protocols, and Services:

Configure DBMS communications to use known and consistent ports, protocols, and services that comply with defined network protection rules.

Ensure random port assignment to network connections is disabled when traversing network firewalls.

Configuration Management (CM) Process:

Ensure configuration management procedures are documented and implemented for changes to the DBMS configuration, software libraries, and other related application software libraries.

System Security Plan:

Ensure the DBMS is included in or has defined for it a System Security Plan.

Define responsibilities and qualifications for those responsible for administering the security of the database system (this includes specifically the DBA in addition to the standard System Administrator (SA) and Information Security Officer (ISO) roles).

System Library Management Controls:

Ensure access to DBMS software is restricted to authorized OS accounts.

Ensure DBMS software is monitored on a regular basis no less frequently than weekly to detect unauthorized modifications.

Ensure database application software is monitored to detect unauthorized modification every week or more often.

Ensure database application software is owned by the authorized application owner account.

Ensure custom application and off-the-shelf source code objects are encoded or encrypted within the production database where supported by the DBMS.

Security Support Structure Partitioning:

Ensure the DBMS is not installed on a host system that provides directory services or other security services except when serving as a required component of the security service (e.g. the DBMS may not be installed on a Windows Domain Controller).

System State Changes:

Ensure all applicable DBMS settings are configured to use trusted files, functions, features, or other components during startup, shutdown, aborts, or other unplanned interruptions (includes the prevention of scanning for automated job submissions at startup and settings to allow only trusted known good data files at startup).

Software Baseline:

Ensure a baseline of database application software and DBMS application objects is maintained for comparison.

IDENTIFICATION AND AUTHENTICATION:

Group Identification and Authentication:

Group authentication does not provide individual accountability for actions taken on the DBMS or data. Whenever a single database account is used to connect to the database, a secondary authentication method that provides individual accountability is required. Ensure actions by a single database account that is accessed by multiple interactive users are attributable to an individual identifier.

Individual Identification and Authentication:

Ensure database user accounts are configured to require individual authentication in order to connect to the DBMS.

Username created by default during installation of the DBMS and components are well-known to potential attackers and provide a known target for malicious intent. Change or delete default account usernames.

Password Storage:

Ensure database account passwords are stored in encrypted format whether stored in database objects, external host files, environment variables or any other storage location.

Encrypt all database account passwords when transmitting across the network.

Do not store database account passwords in batch jobs or application source code.

Password Attributes:

Assign a database account password at database account creation.

Ensure database passwords differ from previous values by more than 4 characters when changed where supported by the DBMS.

Ensure users are not allowed to change their database account passwords more than once every 24 hours without ISO approval where supported by the DBMS. (This requirement does not apply to password changes after password reset actions initiated by the DBA or application administrator).

Ensure database password complexity standards meet current minimum requirements for length and composition where supported by the DBMS.

Set expiration times for interactive database user account passwords to 60 days or less where supported by the DBMS.

Set expiration times for non-interactive database application account passwords to 365 days or less where supported by the DBMS.

Configure database account passwords to be prevented from reuse for a minimum of five changes or one year where supported by the DBMS.

Configure or test database account passwords to prevent use of easily guessed or discovered values.

Assign custom passwords to all default database accounts whether created by the installation of the database software or database components or by third-party applications.

Key Management:

Ensure symmetric keys used for encryption of database user account passwords or other sensitive data used by or for the DBMS are protected and managed in accordance with NSA or NIST-approved key management technology and processes.

Ensure asymmetric keys used for encryption of sensitive data used by or for the DBMS use NSA- or NIST-approved PKI certificates, and ensure the private keys are protected and stored in accordance with NIST-approved key management technology and processes.

Token and Certificate Standards:

Where use of username and password is not a sufficiently secure identification and authentication method to restrict access to sensitive data, ensure a NSA- or NIST-approved PKI certificate and an approved hardware security token or an NSA-certified product is used for identification and authentication to the database.

ENCLAVE AND COMPUTING ENVIRONMENT:

Access for Need-to-Know:

The least possible access to the fewest possible people provides the least opportunity for misuse of the data.

Ensure all access to sensitive administrative DBMS data stored inside the database and in external host files is granted only to DBA and other authorized administrative database and OS accounts.

Ensure all access to sensitive application data stored inside the database, and in external host files, is granted only to database accounts and OS accounts in accordance with user functions as specified by the Data Owner.

Ensure all access to sensitive application data stored or defined within database objects is granted only to database application user roles and not directly to database application user accounts.

Ensure sensitive application data exported from the database for import to remote databases or applications is not provided to personnel or applications not authorized or approved by the Data Owner.

Ensure production data is not exported for import to development databases except in accordance with processes and procedures approved by the Data Owner.

Ensure database client software includes only database identification parameters of databases to which that user is authorized access.

Audit Record Content:

Ensure the DBMS auditing function is enabled.

Ensure all access to DBMS configuration files, database audit data, database credential, or any other DBMS security information is audited.

Ensure all database logons, account locking events, blocking or disabling of a database account or logon source location, or any attempt to circumvent access controls is audited. Where resources are limited, auditing of logons may be reduced to recording only failed logon attempts.

Ensure privileged DBMS actions and changes to security labels or sensitivity markings of data in the DBMS are audited.

Ensure audit records contain the user ID, date and time of the audited event, and the type of the event

Ensure audit records include the reason for any blocking or blacklisting of database accounts or connection source locations.

Audit Trail, Monitoring, Analysis and Reporting:

Review database audit data daily.

Employ automated monitoring tools to review DBMS audit data.

Immediately report suspicious or unauthorized activity.

Include the name of the application used to connect to the database in the audit trail where available.

Changes to Data:

Configure auditing of access or changes to data in accordance with the application requirements specified in the System Security Plan.

Encryption for Confidentiality - Data at Rest:

Ensure applications that access the database are not used with options that display the database account password on the command line.

Ensure sensitive data is encrypted within the database where required by the Data Owner.

Ensure database data files are encrypted where encryption of sensitive data within the DBMS is not available.

Encryption for Confidentiality - Data in Transit:

Ensure remote administrative connections to the database are encrypted.

Ensure database communications are encrypted when transmitting sensitive data across untrusted network segments and in accordance with the application requirements.

Data Change Controls:

Configure the DBMS to enable transaction rollback and transaction journaling or their technical equivalent to maintain data consistency and recovery during operational cancellations, failures, or other interruptions.

Interconnections among Systems and Enclaves:

Ensure interconnections between databases or other applications operating at different classification levels are identified and their communications configured to comply with the interface controls specified in the System Security Plan.

Audit of Security Label Changes:

Some DBMS systems provide a feature that assigns security labels to data elements. When this feature is used, enable auditing of any changes to the classification or sensitivity level assigned to sensitive/confidential data in the DBMS (as required by the Data Owner).

Logon:

Ensure database connection attempts are limited to a specific number of times within a specific time period as specified in the System Security Plan.

The connection attempts counter may be reset to 0 if the maximum number of failed logon attempts does not occur before the timer is reset. Where this requirement is not compatible with the operation of a front-end application, the unsuccessful logon count and time will be specified and the operational need documented in the System Security Plan.

Configure the DBMS to set the duration of database account lockouts due to unsuccessful logon attempts to an unlimited time that requires the DBA to manually unlock the account.

Configure (where supported by the DBMS) a limit of concurrent connections by a single database account to the limit specified in the System Security Plan, a number determined by testing or review of logs to be appropriate for the application. The limit will not be set to unlimited except where operationally required and documented in the System Security Plan.

Separation of Duties and Least Privilege:

Ensure privileges granted to application user database accounts are restricted to those required to perform the specific application functions.

Ensure database application user roles are restricted to SELECT, INSERT, UPDATE, DELETE, and EXECUTE privileges.

Ensure database application user roles are not granted unauthorized access to external database objects.

Ensure database privileges are assigned via roles and not directly assigned to database accounts. Privileges may be assigned directly to application owner accounts where the DBMS does not otherwise support access via roles.

Ensure database administration OS accounts required for operation and maintenance of the DBMS are assigned the minimum OS privileges required by the specific DBMS to perform DBA functions.

Ensure database application objects are owned by an authorized application object owner account.

Ensure the minimum database administrative privileges are assigned to database administrative roles to perform the administrative job function.

Review monthly or more frequently the database privileges assigned to database administrative roles to ensure they are limited to the minimum required.

Restrict restore permissions on databases to DBAs and/or the database owners.

Ensure developers are not granted system privileges within a production database.

Restrict access to the DBMS software installation account to ISO-authorized personnel only.

Ensure use of the DBMS software installation account is logged and/or audited to indicate the identity of the person who accessed the account.

Restrict database privileged role assignments to ISO-authorized accounts.

Ensure privileged database accounts are used only for privileged database job functions, and ensure non-privileged database accounts are used to perform non-privileged job functions.

Ensure custom application owner accounts are disabled or locked when not in use.

Ensure the DBMS software installation account is only used when performing software installation and upgrades or other DBMS maintenance. The ISO will ensure the DBMS software installation account is not used for DBA activities not related to DBMS file permission and ownership maintenance.

Monitor database batch and job queues to ensure no unauthorized jobs are accessing the database.

Marking and Labeling:

Configure DBMS marking and labeling of non-public data where required in accordance with the System Security Plan.

Conformance Monitoring and Testing:

Ensure the DBMS is included in the periodic testing of conformance with vulnerability management and configuration requirements.

Privileged Account Control:

Ensure all database administrative privileges defined within the DBMS and externally to the database are assigned using DBMS or OS roles.

Review DBA role assignments whenever changes to the assignments occur.

Production Code Change Controls:

Ensure application developer database accounts are assigned limited privileges in order to protect production application objects.

Review, at least every three months, privileges granted to developers on shared production/development database systems that allow modification of application code or application objects.

Ensure developer accounts on a shared production/development host system are not granted operating system privileges to production files, directories, or database components.

Resource Control:

Ensure DBMS resource controls are enabled to clear residual data from released object stores.

Security Configuration Compliance:

Ensure the DBMS host and related applications and components comply with all applicable policies and standards.

Software Development Change Controls:

Ensure database applications do not use DDL statements except where dynamic object structures are required. The statements used to define objects in the database are referred to as DDL statements and include CREATE, DROP, and ALTER object statements. (DDL statements do not include CREATE USER, DROP USER, or ALTER USER actions.)

Ensure software development on a production system is separated through the use of separate and uniquely identified data and application file storage partitions and processes/services.

Audit Trail Backup:

Ensure the DBMS audit logs are included in DBMS backup procedures.

Audit Trail Protection:

Ensure DBMS audit records are protected from unauthorized access.

Account Control:

Ensure unauthorized database accounts are removed or disabled.

Monitor database account expiration and inactivity and remove expired accounts and accounts that are inactive for 60 days or longer.

ENCLAVE BOUNDARY DEFENSE:

Ensure the DBMS is protected from direct client connections from public or unauthorized networks.

Ensure remote administration of the database is not enabled or configured unless mission and/or operationally required and authorized by the ISO.

Configure auditing of all actions taken by database administrators during remote sessions.

Review daily audit trails of remote administrative sessions to discover any unauthorized access or actions. Weekly review is required by State IT Standard 670-08S1: Secure System Maintenance.

Restrict remote administration connections to the database to dedicated and encrypted network addresses and ports.

CONTINUITY:

Ensure files critical to database recovery are protected by employment of database and OS high-availability options such as storage on RAID devices.

Ensure access to database backup and recovery files are restricted to the database and/or OS backup and recovery processes, DBAs, and database backup/recovery operators.

Backup database data, configuration, and other files critical to database operation at intervals consistent with the database's assigned criticality level.

Document, implement, and test the DBMS backup and recovery strategy at least semi-annually.

Ensure critical database software directories are backed up.

Configure the DBMS to use only authorized software, data files, or other critical files during recovery.

VULNERABILITY MANAGEMENT:

Remove or upgrade unsupported DBMS software prior to a vendor dropping support.

Create a formal migration plan for removing or upgrading DBMS systems 6 months prior to the date the vendor drops security patch support.

Ensure all applicable vendor-provided security patches are installed in accordance with State policy/standards.

SQL INJECTION

No database is safe. An attack technique that has been widely used is Structured Query Language (SQL) Injection. SQL injection is a method for exploiting web applications that use client-supplied data in SQL queries. SQL Injection refers to the technique of inserting SQL meta-characters and commands into Web-based input fields in order to manipulate the execution of the back-end SQL queries. The parameters of a web-based application are modified in order to manipulate the SQL statements that are passed to the database to return data. For example, it is possible to cause a second query to be executed with the first by adding a single quote (') to the parameters.

SQL injection attacks can occur on Microsoft SQL Server, MySQL® and other SQL database servers, so no matter how strong the firewall rule sets are or how diligent the patching mechanisms may be, Web application developers must follow secure coding practices to prevent attackers from breaking into their database systems. Application developers should be familiar with various types of SQL injection attacks that exist and plan ways to combat them.

SQL INJECTION TECHNIQUES:

SQL Injection happens when an application accepts user input that is directly placed into a SQL Statement and doesn't properly filter out dangerous characters. This can allow an attacker to not only steal data from the database, but also modify or delete it. Certain SQL Servers such as Microsoft SQL Server contain Stored and Extended Procedures (database server functions). If an attacker can obtain access to these procedures it may be possible to compromise the entire machine.

SQL injection techniques include authorization bypass, using the SELECT command, using the INSERT command, and using SQL server stored procedures. Bypassing logon forms is the simplest SQL injection technique. Most web applications that use dynamic content of any kind will build pages using information returned from SELECT queries. SELECT queries are used to retrieve information from a database. The WHERE clause is the part of the query that is most susceptible to manipulation.

Attackers take advantage of the fact that programmers often chain together SQL commands with user-provided parameters, and can therefore embed SQL commands inside these parameters. The result is that the attacker can execute arbitrary SQL queries and/or commands on the backend database server through the Web application. The technique exploits web applications that use client supplied data in SQL queries.

Attackers commonly insert single quotes into a URL's query string, or into a forms input field to test for SQL Injection. If an attacker receives an error message, there is a good chance that the application is vulnerable to SQL Injection.

DETECTING SQL INJECTION:

The most common way of detecting SQL injection attacks is by looking for SQL signatures in the incoming HTTP stream. For example, looking for SQL commands such as UNION, SELECT or xp_.

Pay attention to any occurrence of SQL specific meta-characters such as the single-quote, semicolon, or the double-dash (--). Be aware these signatures may result in a high number of false positives.

PROTECTION AGAINST SQL INJECTION ATTACKS:

Application developers should apply the following recommendations to protect against SQL injection attacks.

Limit user access. Use the principle of least privilege and ensure that the users created for the applications have the privileges needed and no more. All extra privileges, for example, PUBLIC ones, should not be available.

Delete unused accounts.

Do not enable the guest account.

Never use the default system account (sa). Always setup specific accounts for specific purposes.

Require authentication for all SQL Server accounts and ensure that passwords meet State standards.

Ensure that the mapping between database users and logins at the server level is correct. Run sp_change_users_login with the report option regularly to ensure that the mapping is as expected.

Secure the database and ensure that all excess privileges are removed.

Change database default passwords.

Remove all PUBLIC privileges where possible from the database.

Never grant permissions to the PUBLIC database role.

Run the listener as a non-privileged user.

Implement the design principle of least privilege on SQL servers. Run separate SQL Server services under separate accounts with the lowest possible privileges.

Reduce system exposure by running only the services and features that are necessary.

Remove or move to an isolated server the unused extended stored procedures, triggers, user-defined function, etc.

Remove sample databases from production servers.

Delete or archive installation files.

Remove culprit characters and character sequences. Safeguards against SQL injection include sanitizing the data and securing the application. To reduce the chance of an injection attack, remove characters and character sequences such as semicolons, double-dash (--), SELECT, INSERT, and xp_ from user input before building a query.

Check for escape quotes. The majority of injection attacks require the use of single quotes to terminate an expression. Do not allow dynamic SQL that uses concatenation, or at least filter the input values and check for special characters such as quote symbols. Using a simple function to convert all single quotes to two quotes reduces the chance of an injection attack succeeding.

Limit the length of user input. Keep all text boxes and form fields as short as possible to limit the number of characters that can be used to formulate an SQL injection attack. It is poor practice to have a text box on a form that can accept more characters in the field if the field you are comparing it against can only accept fewer characters.

Review the application source code. The solutions vary because the code can be written in many different languages (PHP, JSP, java, PL/SQL, VB, etc.). Review the source code for dynamic SQL where concatenation is used. Find the call that parses the SQL or executes it. Check back to where values are entered. Ensure that input values are validated and that quotes are matched and meta-characters are checked.

Minimize use of dynamic SQL. If dynamic SQL is necessary, use bind variables.

Minimize use of dynamic PL/SQL. If dynamic PL/SQL is necessary, use bind variables.

If PL/SQL is used, use AUTHID CURRENT_USER so that the PL/SQL runs as logged in user and not the creator of the procedure, function or package.

Restrict PL/SQL packages that can be accessed from apache

If concatenation is necessary then check the input for malicious code (i.e., check for UNION in the string passed in or meta-characters such as quotes) and use numeric values for the concatenation part.

Use a firewall (in accordance with State standards and guidelines).

Always block TCP port 1433 and UDP port 1434 on the perimeter firewall to include named instances that are listening on additional ports.

Never install SQL Server on a domain controller.

In a multi-tier environment, run Web logic and business logic on separate computers.

Do not change the default settings for the command xp_cmdshell. Do not grant execute permission on xp_cmdshell to users who are not members of the sysadmin role. Only members of the sysadmin role can execute xp_cmdshell.

Disable ad hoc data access on all providers except SQL OLE DB for all users except members of the sysadmin fixed server role.

Restrict membership of the sysadmin fixed server role to a trusted individual.

Set login auditing level to failure or all.

Enable security auditing of sysadmin actions, fixed role membership changes, all login related activity, and password changes.

Disable cross database ownership chaining. Use sp_dboption to enumerate and validate databases for which cross database ownership chaining is required and has been enabled.

Encrypt sensitive data so it cannot be viewed. Ensure encryption is compliant with state standards.

Install a certificate to enable SSL connections. Certificates should use the fully-qualified DNS name of the server.

Use the SQL Server service account to encrypt database files with EFS (Encrypting File System).

Verify the safety of stored procedures that have been marked for AutoStart.

Do not allow direct catalog updates.

Add Microsoft Baseline Security Analyzer (MBSA) to the weekly maintenance schedule, and follow up on any security recommendations that it makes.

INPUT VALIDATION AND DATA INTEGRITY

One of the major causes of software vulnerabilities is failure to validate input. Common security vulnerabilities found in applications as a result of un-validated input include:

- Buffer Overflow
- Integer Overflow
- Command Injection
- Format String
- Cross-site Scripting

Implement the following guidelines to ensure applications perform checks on the validity of input data, user permissions, and resource existence before performing a function. These guidelines may be used for both in-house application development and to assist in the evaluation of the security of third-party applications. The guidance provided is not specific to any one platform, programming language or application type. Some portions of this guideline may not apply to all applications or in all cases. In some cases, specific guidance has been provided based upon platform, programming language, or language types.

INPUT VALIDATION:

Input may come from a user, data store, network socket, or other source. Before being used applications should validate all user input and input data from any source when it crosses a trust boundary.

When validating data the application should check for known-good data and reject any data not meeting predefined criteria. Checking for known-bad data, while potentially effective, may still allow an attacker to get bad data through the validation by using alternate data encoding or patterns the developers have not considered. Allowing only known-good data through may increase the results of valid data being rejected; however, this significantly lessens the possibility of invalid data being passed on to the application.

COMMON SECURITY VULNERABILITIES:

Managers, designers, developers, and testers should actively monitor application security developments for new classes of vulnerabilities. The following is not an exhaustive list of vulnerabilities, nor is every vulnerability applicable to every application environment.

Buffer Overflow:

Buffer overflow is a situation where data is written beyond the end of an allocated memory block. There are several types of buffer overflow which all lead to the same potential issues in the application, the ability to crash the application or execute arbitrary code on the system. If the vulnerable application is running as a system account, this could lead to a compromise of the system.

The following items may indicate potential buffer overflow within the application:

- Cases where input is not checked before being copied into a buffer
- Use of some of the functions listed in Table 4-1 (below)
- Incorrect calculations to determine buffer sizes
- Incorrect calculations to determine array indexes

The primary methods of detecting buffer overflow are code reviews and Fuzz testing. Fuzz testing is the process of sending large data blocks and invalid data blocks as input to an application in an attempt to cause an application error.

In order to minimize buffer overflow implement the following procedures:

- Replace function to be avoided with safer functions (see Table 4-1 below)
- Recheck all calculations to ensure buffer sizes are calculated correctly
- Recheck all array access and flow control calculations
- (C++) Replace character arrays with STL string classes
- Validate all input before use, allowing only known-good input through

Unsafe Functions:

The following C/C++ functions have a high potential for causing application vulnerabilities. The presence of these functions does not immediately indicate a vulnerability, however if they are used inappropriately a vulnerability may exist.

TABLE 1: C/C++ FUNCTIONS TO AVOID

Unsafe Function	Risk	Potential Replacement Functions
strcpy	Buffer Overflow	strcpy_s, StringCchCopy, StringCbCopy, strlcpy
wscpy		
_tcscpy		
_mbscpy		
strCpy		
strCpyA		
strCpyW		
Lstrcpy		
lstrcpyA		
lstrcpyW		
strcpyA		
strcpyW		
_tccpy		
_mbccpy		

Unsafe Function	Risk	Potential Replacement Functions
strcat	Buffer Overflow	strcat_s, StringCchCat, StringCbCat, strlcat
wscat		
_tcscat		
_mbscat		
strCat		
strCatA		
strCatW		
lstrcat		
lstrcatA		
lstrcatW		
strCatBuffW		
strCatBuff		
strCatBuffA		
strCatChainW		
strcatA		
strcatW		
_tccat		
_mbccat		
lstrlen		

String Considerations:

Many functions, messages, and macros use strings in their parameters. Failing to check strings for null-termination or for length or miscalculating the length of a string or buffer can lead to buffer overflow or data truncation. To handle strings in a safe manner:

- Check strings for null-termination or for the proper length, as appropriate
- Determine the length of a string or buffer, especially when it is a TCHAR
- When creating a string, or using a string that was used previously, initialize it to zero or insert a null-terminator (as appropriate)
- Use the Strsafe.h functions (Microsoft) when dealing with C/C++ strings

Integer Overflow:

Integer overflow results from inconsistent and/or incorrect results from calculations due to the handling of integer data types in various programming languages. These errors can lead to symptoms ranging from crashes and incorrect results to allowing an attacker to exploit a buffer overflow. Integer overflow most often occur because developers are not aware of the implicit conversions occurring when using variables of different data types or when using different operators.

The following may indicate the presence of integer overflow in the application:

- Mixing signed and unsigned data types in calculations or comparisons
- Mixing data types of different sizes in calculations or comparisons
- Comparisons between variables and literal values
- Use of un-validated input
- Calculations not validated before the result is utilized

The primary method of detecting integer overflows in an application is a code review. If a code review cannot be performed, test the application to ensure it correctly handles different numeric values and strings of varying lengths. The following items list some of the tests to be performed.

- Input negative values for numeric input
- Input border case values (i.e., 0, 7, 8, 254, 255, 16353, 16354)
- Input extremely large string values (> 64k)
- Input strings whose lengths equal border cases (32k, 32k-1, 64k, 64k-1)

In order to minimize integer overflow implement the following procedures:

- Use unsigned values whenever possible
- Use only unsigned integers in memory allocation whenever possible
- Use only unsigned array indexing functions whenever possible
- Validate user input of numeric value, allowing only known good data to pass
- Compile with the highest warning level possible
- (C++) Use `size_t` types to hold size variables
- (C++) Use the `SafeInt` class (available from Microsoft)
- (C#) Compile with the `/checked` compiler option
- (GCC) Compile with the `-ftrapv` option

Command Injection:

Command Injection vulnerabilities exist when data that can be modified to execute some task is passed to an interpreter or compiler. This can allow an attacker to execute code, possibly at a higher privilege level, resulting in system compromise.

Potential Command Injection vulnerabilities include the use of functions spawning an interpreter or compiler. Table 4-2 lists some common functions vulnerable to Command Injection.

TABLE 2: FUNCTIONS VULNERABLE TO COMMAND INJECTION

Language	Functions/Characters
C/C++	<code>system()</code> , <code>popen()</code> , <code>execlp()</code> , <code>execvp()</code> , <code>ShellExecute()</code> , <code>ShellExecuteEx()</code> , <code>_wsystem()</code>
Perl	<code>system</code> , <code>exec</code> , <code>`</code> , <code>open</code> , <code> </code> , <code>eval</code> , <code>/e</code>
Python	<code>exec</code> , <code>eval</code> , <code>os.system</code> , <code>os.popen</code> , <code>execfile</code> , <code>input</code> , <code>compile</code>
Java	<code>Class.forName()</code> , <code>Class.newInstance()</code> , <code>Runtime.exec()</code>

In addition to code reviews, testing should be performed to help identify Command Injection vulnerabilities. In order to test for Command Injection vulnerabilities:

- Identify all potential interpreters or compilers used to pass data
- Identify the characters modifying the interpreters' behavior
- Construct input strings containing these characters causing a visible effect on the system, pass them to the application, and observe the behavior (ensure all input vectors are tested in this manner)

In order to minimize Command Injection vulnerabilities, validate all input before passing to an interpreter or compiler, allowing only known-good input through.

Format String:

Format string vulnerabilities occur when specially crafted format strings passed to a function allow flow control information to be viewed or modified or cause the program to read or write arbitrary memory locations. In a worst-case scenario format string vulnerabilities can allow an attacker to execute code of their choice on the system resulting in complete system compromise.

An application taking input and passing it to a formatting function may indicate the presence of format string vulnerabilities. The primary method of detecting this vulnerability is a source code review. If a source code review cannot be performed, then application testing may be used. To test the application insert format string specifiers in all areas of string input and observe the output looking for anomalies. The format specifiers will vary based upon the language used; however, in addition to language-specific specifiers, the C language specifiers should also be tested.

In order to minimize format string vulnerabilities, implement the following procedures:

- Validate all input before passing it to a function; allow only known-good data to pass
- Format strings used by the application should only be accessible by privileged users
- (C++) Use stream operators instead of the printf family of functions

TABLE 3: C/C++ UNSAFE FUNCTIONS (PRINTF FAMILY)

Unsafe Function	Risk	Potential Replacement Functions
Printf	Format String Vulnerabilities	Add format string ("%s") as a first argument
fprintf		
sprintf	Format String Vulnerabilities	snprintf
vsprintf	Format String Vulnerabilities	vsnprintf
wsprintf	Format String Vulnerabilities	StringCbPrintf, StringCbPrintfEx, StringCbVPrintf, StringCbVPrintfEx, StringCchPrintf, StringCchPrintfEx, StringCchVPrintf, StringCchVPrintfEx.
wvsprintf		

Cross Site Scripting (XSS):

XSS is a situation where input is accepted by a web site and then sent back to a web page. This input can include code to be executed by the browser. Since this code is seen as originating from the web server it can access data from the servers' domain such as a cookie, or modify the behavior of the web site by modifying links and other malicious actions. A cross site scripting vulnerability can lead to an attacker gaining personal information or directing a user to a site of the attacker's choice.

If user input is echoed back into the browser, the application may have potential XSS issue.

In addition to code reviews, testing should be performed to identify potential XSS vulnerabilities. In order to test for XSS vulnerabilities:

- Make a request against the application, setting all input parameters to known-bad values
- View the HTML response, looking for the known-bad values sent as input (the response containing the values submitted as input may not be returned immediately, it may go through intermediate processes before being sent back to a browser or may be sent in response to a future request or possibly to a different entity)

In order to minimize XSS vulnerabilities in the application, implement the following procedures:

- Filter all input, allowing only known-good input
- If special characters are required for input, HTML encode user input
- Set a known character for all web pages to eliminate unexpected characters

DATA INTEGRITY CONTROLS:

The application should use a NIST-approved technology to implement a hash (e.g., Secure Hash Algorithm One [SHA-1]), checksum, or digital signature of the data before transmission.

When code-signing is required, the application should invoke a NIST-approved digital signature technology to digitally sign the code prior to transmission.

The application should invoke NIST-approved technology to apply a hash, checksum, or digital signature to the data before storage.

The application should be able to validate the integrity mechanism and reject data for which the integrity mechanism validation fails.

The application should validate parameters before acting on them and reject all parameters for which one or more of the following is true:

- Not formatted as expected;
- Do not fall within the expected bounds (length, numeric value, etc.)

The application should inform the user of the expected characteristics of the input—e.g., length, type (alphanumeric, numeric only, alpha-only, etc.), and numeric or alphabetic range.

The application should validate all data input by users or external processes and reject all input for which one or more of the following is true:

- not formatted as expected
- contains incorrect syntax
- not a valid data string
- contains parameters or characters with invalid values
- falls outside the expected bounds (e.g., length, range)
- contains a numeric value that would cause a routine or calculation in the application to divide any number by zero
- contains any parameters the source of which cannot be validated by the user's session token
- can induce a buffer overflow
- contains HTML
- contains special characters, meta code, or meta-characters that have not been encoded (if encoding is allowed)
- contains direct SQL queries
- contains any other type of unexpected content or invalid parameters
- contains a truncated pathname reference

The application should suspend all processing of the transaction in which input has been received until the input has been completely validated.

All application programs, including CGI and shell scripts, should perform input validation on arguments received before acting on those arguments.

The application should validate all inputs it receives from any external processes, including processes in third-party software integrated into the application, in the same way it validates user input data.

All functions in the application program should perform bounds checking, such that the functions check the size of all buffer or array boundaries before writing to them, or before allowing them to be written to. In addition, the application should limit the size of what it writes to the buffer or array to the size imposed by the buffer/array boundaries (i.e., to prevent what is written from exceeding the buffer/array size and overflowing the boundary).

The application should bounds check all arrays and buffers every time those arrays/buffers are accessed.

The application should validate all input data before copying those data into the database.

The application should reject any input containing HTML (including HTTP strings that contain HTML tags) from an untrusted user or other untrusted source.

The application should be able to recognize questionable URL extensions, and validate all URLs sent to it by clients to ensure they do not contain such extensions or truncate the URL to remove the dubious extension.

The application should not accept Web page content from any untrustworthy source. The application should verify and validate the source of any Web page content before posting that content.

All user input validations should be performed by the server application even if input validation has already been done by the client application. The client application should not be relied on to perform trustworthy input validation. Client input validation may be used to prescreen data before it is validated by the server.

The application's validation of user input data that contains active content (e.g., mobile code) should not result in the execution of the active content.

The application's data update processes should operate correctly, and should not incorrectly reparse, inadvertently introduce errors to, or otherwise corrupt the data they update.

The application should find and validate the digital signature and any hash, checksum, or other additional integrity mechanism applied to that code prior to executing it. If the code's integrity mechanism cannot be validated, or is not present, the application should discard the code without executing it; and audit this discard.

The application should invoke a virus scanning tool to scan all files received from users and external processes to ensure these files do not contain malicious content.

The application should invoke a virus scanning tool whenever it executes a program that may access one of the application's configuration or other parameter-containing files.

The application should ensure that the data to be forwarded do not contain or point to malicious code.

The process that validates the application's executable code integrity mechanism checksum or hash should be invoked every time the application is executed to ensure that the application's executable code state has not changed since the original integrity mechanism was applied. If this validation fails, the validation process should prevent the application from being executed, and notify the administrator that the application code needs to be replaced by an uncorrupted executable.

The application should not execute received code until it:

- verifies that the code has been digitally signed; and
- validates the digital signature on the code.

The application should time/date stamp each data modification or file update.

The application should display to each user who retrieves the data the time and date on which the data was last modified.

The application code should explicitly initialize all of its variables when they are declared.

The application should validate the source of all HTML updates to hidden fields and should reject any HTML field changes from unvalidated sources.

The application should not embed parameter data about fields in HTML forms in hidden fields in those HTML forms.

The application should use hash or digital signature to ensure the integrity of transmitted forms (user-to-server) containing sensitive information.

The application should not trust user input or other user-supplied data that have not been received over a trustworthy channel, unless those data are encrypted and digitally signed.

The application should never return sensitive information in response to input from untrustworthy sources.

The application cannot be used to bypass the access controls or to spoof the trusted user to modify data within database entries/records (data integrity), the relationships between those entries/records (relational integrity), or the references to those entries/records (referential integrity)

ERROR HANDLING

Improper error handling in an application can lead to an application failure, or possibly result in the application entering an insecure state. Improper error handling is usually not directly attackable; however, during a failure attackers must be prevented from gaining unauthorized access to privileged objects that are normally inaccessible. If upon failing, the system reveals sensitive information about the failure to potential attackers then it's possible for the attacker to cause a Denial of Service (DoS) or possibly exploit the insecure state of the application.

Employ the following guidelines to ensure State of Alabama applications are not subject to error handling vulnerabilities. These guidelines may be used for both in-house application development and to assist in the evaluation of the security of third-party applications. The guidance provided is not specific to any one platform, programming language or application type. Some portions of this guide may not apply to all applications.

ERROR HANDLING VULNERABILITY:

When applications fail they may go into a non-secure state unless errors are properly handled. Though the most likely effect of improper error handling is a self-inflicted DoS, an attacker could potentially exploit an error handling vulnerability by causing the right kind of failure to happen and bypassing the failed security controls to gain access to the system.

The following may lead to potential error handling issues:

- Failure to check return codes or exceptions
- Improper checking of return codes or exceptions
- Improper handling of return codes or exceptions

The primary way to detect error handling vulnerabilities is to perform code reviews. If a code review cannot be performed, it may be possible to test some error conditions by specifying invalid filenames and using different accounts to run the application. These tests may give indications of vulnerability, but they are not comprehensive.

To minimize error handling vulnerabilities, ensure return code and exception handling is properly and thoroughly implemented throughout the application.

INFORMATION DISCLOSURE VULNERABILITY:

Information disclosure vulnerabilities result from the disclosure of information about the application or application components that may be useful to or the target of a malicious attack. The information itself may be the target of an attacker, or the information could provide an attacker with data needed to compromise the application or system. Information disclosure vulnerabilities are most often the result of programming errors, insufficient authentication, poor error handling or inadequate data protection.

The following may indicate the presence of information disclosure vulnerabilities in an application:

- Information about the operating environment displayed to a user
- Output not marked at the appropriate protection level
- Data access requests not subject to permission checks
- Error messages revealing information to the users through their wording or timing
- Application responses revealing internal information to the user through their wording or timing

The primary method of identifying information disclosure vulnerabilities is to perform a code review. In addition to a code review the application may be tested by:

- Inducing errors in the program to verify the contents of error messages
- Attempting variations of invalid input combinations to determine if the response contents or timing reveal information about the system (For example, is the response time or error message different for an invalid username/invalid password and a valid username/invalid password combination? If so this would allow an attacker to find valid usernames.)
- Attempting to access data the user should not be able to access

In order to minimize Information Disclosure vulnerabilities:

- Ensure a permission structure is in place to enforce access control of the data
- Display generic error messages to end users; provide only enough information so the user knows the request failed
- Ensure application responses do not divulge unneeded details or sensitive information
- Log specific error information details to a secure log file

DESIGN AND CODING PRINCIPLES:

Secure Failure:

Applications should perform checks on the validity of data, user permissions, and resource existence before performing a function. Secure failure means if a check fails for any reason, the application code should provide exception handling leaving the application in a secure state. Failing to a secure state means the application has not disclosed any data that would not be disclosed ordinarily and that the data still cannot be tampered with. The principle of secure failure design is intended to account for all possible exceptions that could leave the application in a vulnerable state.

Default to Deny:

This principle requires that the default access to an object is none. Whenever access, privileges, or some other security-related attribute is not explicitly allowed, it should be denied. A design or implementation error in an object that requires explicit permission (allow by exception) tends to fail by refusing permission, a safe situation since it will be quickly detected. On the other hand, a design or implementation error in an object that explicitly denies access tends to fail by allowing access, a failure that may go unnoticed.

Graceful Termination:

Application processes that receive invalid input should request the external user or process to reinsert the data. If the reinserted data is still invalid, the application should gracefully terminate the user process with an error message to the user indicating that the process is terminating as a result of an input error.

On Failure Undo Changes:

Before it shuts down, the application should reverse any changes in its operating mode or state that occurred during execution and return to its normal mode and state of operation.

MOBILE CODE

Mobile code is software obtained from remote systems, transferred across a network, and then downloaded and executed on a local system without explicit installation or execution by the recipient. Mobile code technologies include, for example, Java, JavaScript, ActiveX, PDF, Postscript, Shockwave movies, Flash animations, and VBScript. If used maliciously, mobile code has the

potential to cause damage to information systems by introducing malware or enabling unauthorized system access.

The following mobile code usage restrictions and implementation guidance apply to both the selection and use of mobile code installed on organizational servers and mobile code downloaded and executed on individual workstations.

Guidance on the implementation and use of mobile code is based on the mobile code category.

CATEGORY 1 MOBILE CODE:

Category 1 mobile code involves technologies having broad functionality and unmediated access to the services and resources of a computing platform. There are two subgroups of Category 1 mobile code technologies:

Category 1A Mobile Code:

The following mobile code technologies are assigned to Category 1A:

- ActiveX controls
- Shockwave movies (including Xtras)

The use of unsigned Category 1A mobile code in State of Alabama information systems is prohibited.

Category 1A mobile code may only be used under the following conditions:

- The Category 1A mobile code is signed with an NSA- or NIST-approved PKI code-signing certificate
- The signed Category 1A mobile code signature is validated before executing

Category 1X Mobile Code:

The following mobile code technologies are assigned to Category 1X:

- Mobile code scripts executing in Windows Scripting Host (WSH) (e.g., JavaScript, VBScript downloaded via URL file reference or e-mail attachments). When JavaScript and VBScript execute within the browser, they are Category 3; however, when they execute in WSH, they are Category 1.
- HTML Applications (e.g., hta files) downloaded as mobile code
- Scrap objects (e.g., .shs and .shb files)
- Microsoft Disk Operating System (MS-DOS) batch scripts
- Unix shell scripts
- Binary executables (e.g., .exe files) downloaded as mobile code

Category 1X mobile code should not be used in State of Alabama applications.

CATEGORY 2 MOBILE CODE:

Category 2 mobile code involves technologies having full functionality, but mediated or controlled access to the services and resources of a computing platform. The following mobile code technologies are assigned to Category 2:

- Java applets and other Java mobile code
- Visual Basic for Applications (VBA) (e.g., Microsoft Office macros, also used by Corel Office)
- LotusScript (e.g., Lotus Notes scripts)
- PerfectScript (e.g., Corel Office macros)
- Postscript
- Mobile code executing in .NET Common Language Runtime

Unsigned Category 2 mobile code is required to execute in a constrained environment without access to local system and network resources (e.g., file system, Windows Registry, network connections other than to its originating server).

Unsigned Category 2 mobile code executing in a constrained execution environment without access to local system and network resources may be freely used in State of Alabama information systems.

Category 2 mobile code not executing in a constrained execution environment may be used only under the following conditions:

- The mobile code is obtained from an assured channel from a trusted source
- The mobile code is signed with an NSA- or NIST-approved PKI code-signing certificate
- The signed Category 2 mobile code signature is validated before executing

CATEGORY 3 MOBILE CODE:

Category 3 mobile code involves technologies having limited functionality, with no capability for unmediated access to the services and resources of a computing platform.

The following mobile code technologies are assigned to Category 3:

- JavaScript, including Jscript and ECMAScript variants, when executing in the browser
- VBScript, when executing in the browser
- Portable Document Format (PDF)
- Flash animations (e.g., .swf and .spl files) executing in the Shockwave Flash Plugin

Category 3 mobile code technologies may be freely used without restrictions in State of Alabama information systems.

EMERGING MOBILE CODE:

Emerging mobile code technologies refer to all mobile code technologies, systems, platforms, or languages whose capabilities and threat level have not yet undergone a risk assessment and have not been assigned to one of the a risk categories above. Because of the uncertain risk, the use of emerging mobile code technologies in State of Alabama information systems is prohibited.

MOBILE CODE IN E-MAIL:

Mobile code can be embedded in an e-mail body or an e-mail attachment and downloaded as part of the actual e-mail. Alternately, mobile code residing on a remote server can be referenced from within an e-mail body or attachment and can be automatically downloaded and executed. Some types of mobile code execute automatically as soon as the user clicks on the message subject or previews the message; others execute when the user opens an attachment containing mobile code.

Designers should ensure the application only embeds approved categories of mobile code in e-mail messages.

NEW PROCUREMENT AND DEVELOPMENT EFFORTS:

All new procurement and development efforts relying on mobile code technologies should include a mobile code risk mitigation strategy detailing the measures incorporated into the system development to curtail the risk posed by its use.

EXCLUSIONS:

Mobile code originating from and traveling exclusively within a single enclave boundary is exempt from the above restrictions.

The application need not satisfy any of the above mobile code-related restrictions for the following types of mobile code:

(1) Scripts and applets embedded in or linked to Web pages and executed in the context of the Web server (e.g., Java servlets, Java Server Pages, Java RMI, Java Jini, CGI, Active Server Pages, Cold Fusion Markup Language (CFML), PHP Hypertext Processor (PHP), Server Side Include (SSI), server-side JavaScript, and server-side LotusScript).

(2) Local programs and command scripts (e.g., binary executables, shell scripts, batch scripts, Windows Scripting Host [WSH], Perl scripts).

(3) Distributed object-oriented programming systems (e.g., Common Objective Request Broker Architecture [CORBA], Distributed Component Object Model [DCOM]).

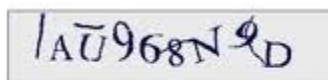
(4) Software patches, updates, including self-extracting updates and software updates that must be invoked explicitly by the user (e.g., Microsoft Windows Update).

CAPTCHA

A CAPTCHA (Completely Automated Public Turing tests to tell Computers and Humans Apart) is a type of challenge-response test used in computing as an attempt to ensure that the response is generated by a person.

Many websites and web applications use CAPTCHAs in an attempt to block automated interactions with their sites, protect the website from bots, and prevent standard automated software from filling out forms. The most widely used CAPTCHA schemes use combinations of distorted characters and obfuscation techniques that humans can recognize but that may be difficult for automated scripts.

FIGURE 1: SAMPLE CAPTCHA



Based on the paper, [Text-based CAPTCHA Strengths and Weaknesses](#), by Elie Bursztein, Matthieu Martin and John C. Mitchel, the following design principles and techniques should be used to customize and strengthen CAPTCHA implementations.

Core Design Principles:

1. Randomize CAPTCHA length: Don't use a fixed length; it gives too much information to the attacker.
2. Randomize Character size: Make sure the attacker can't make educated guesses by using several font sizes and several fonts. Using several fonts reduces the classifier accuracy and the scheme's learnability.
3. Wave the CAPTCHA: Making the CAPTCHA into a wave shape increases the difficulty of finding cut points in case of collapsing and helps mitigate the risk of the attacker finding the added line based on its slope when using lines.

Anti-Recognition Techniques:

4. Rotation, scaling, rotating some characters, and using various font sizes will reduce the recognition efficiency and increase the anti-segmentation security by making character width less predictable.
5. Don't use a complex character set: Using a large character set does not improve significantly the CAPTCHA security scheme and actually hurts human accuracy, thus using a non-confusable character set is the best option.

Anti-Segmentation Techniques:

6. Use collapsing and/or lines: Given the current state of the art, using any sort of complex background as an anti-segmentation technique is considered to be insecure. Using lines or collapsing correctly are the only two secure options currently. Complex backgrounds can be used as a second line of defense.

7. Plan implementation carefully: Anti-segmentation techniques are only effective if the anti-recognition techniques and core design are sound. For example, using collapsing is only effective if the size and the number of characters are random. Failing to randomize either of these leaves the scheme vulnerable to an opportunistic segmentation (this paper explains this in detail; see: <http://cdn.ly.tl/publications/text-based-captcha-strengths-and-weaknesses.pdf>).

8. Create alternative schemes: As with cryptography algorithms, it is good practice to have alternative CAPTCHA schemes that can be rolled out in case of a break. Variations of the same battle-hardened schemes with additional security features are likely the easiest way to prepare alternative schemes.

ADDITIONAL INFORMATION:

Information Technology Policy 661: Application Security

http://cybersecurity.alabama.gov/documents/Policy_661_Application_Security.pdf

Information Technology Dictionary

http://cybersecurity.alabama.gov/documents/IT_Dictionary.pdf

By Authority of the Office of IT Planning, Standards, and Compliance

DOCUMENT HISTORY:

Version	Release Date	Comments
661G1-00	09/01/2011	This document consolidates and replaces Guidelines 660-01G1 through 660-01G4 and Standard 660-01S1.
661G1-01	12/01/2011	Added guidelines for CAPTCHA