Principle 2
High Quality Implementation

Requirements for Principle 2

Principle 2 High Quality Implementation

2.1 - The voting system and its software are implemented using trustworthy materials and best practices in software development.

2.1-A – Unstructured control flow

2.1-B – Goto

2.1-C – Separation of code and data

2.1-D – Hard-coded passwords and keys

2.2 – The voting system is implemented using best practice user-centered design methods, for a wide range of representative voters, including those with and without disabilities, and election workers.

2.2-A – User-centered design process

2.3 - Voting system logic is clear, meaningful, and well-structured.

2.3-A – Acceptable programming languages

2.3-B – Acceptable coding conventions

2.3-C – Published coding conventions

2.3-D – Credible

2.3-D – Header comments

2.4 - Voting system structure is modular, scalable, and robust.

2.4-A – Modularity

2.4-B – Module testability

2.4-C – Module size and identification

2.4-D – Callable unit length limit

2.4-E– Lookup tables in separate files

2.4-F– Limited cylcomatic complexity

2.5 - The voting system supports system processes and data with integrity.

2.5.1– Code coherency

2.5.1-A – Self-modifying code

2.5.1-B – Unsafe concurrency

2.5.1-C – Code integrity

2.5.1-D – Interpreted code, specific COTS interpreter

2.5.2 – Prevent tampering

2.5.2-A – Prevent tampering with code

2.5.2-B – Prevent tampering with data

2.7 - The voting system performs reliably in anticipated physical environments.

2.7-A Electrical disturbances

2.7-A.1 FCC Part 15 Class A and B conformance

2.7-A.2 Power supply – energy service provider

2.7-A.3 Power port connection to the facility power supply

2.7-A.4 Leakage via grounding port

2.7-A.5 Outages, sags and swells

2.7-A.6 Withstand conducted electrical disturbances

2.7-A.7 Emissions from other connected equipment

2.7-A.8 Electrostatic discharge immunity

2.7-A.9 Radiated radio frequency emissions

2.1 - The voting system and its software are implemented using trustworthy materials and best practices in software development.

2.1-A – Unstructured control flow

Application logic must contain no unstructured control constructs.

Discussion
Although it is typically developed by the voting system manufacturer, border logic is constrained by the requirements of the third-party or COTS interface with which it interacts. It is not always possible for border logic to achieve its function while conforming to standard coding conventions. For this reason, border logic should be minimized relative to application logic and where possible, wrapped in a conforming interface. An example of border logic that could not be so wrapped is a customized boot manager that connects a bootable voting application to a COTS BIOS.

2.1-B – Goto

Arbitrary branches (a.k.a. gotos) must not be used.

2.1-C – Separation of code and data

Application logic must not compile or interpret configuration data or other input data as a programming language.

Discussion
The applicable requirement in VVSG2005 reads "Operator intervention or logic that evaluates received or stored data must not re-direct program control within a program routine." That attempt to define what it means to compile or interpret data as a programming language caused confusion.

Distinguishing what is a programming language from what is not requires some professional judgment. However, in general, sequential execution of imperative instructions is a characteristic of conventional programming languages that should not be exhibited by configuration data. Configuration data must be declarative or informative in nature, not imperative.

For example: Configuration data can contain a template that informs a report generating application about the form and content of a report that it should generate. However, configuration data cannot contain instructions that are executed or interpreted to generate a report, essentially embedding the logic of the report generator inside the configuration data.

The reasons for this requirement are…

- mingling code and data is bad design, and

- embedding logic within configuration data evades the conformity assessment process for application logic.

Prior VVSG source:	2007 6.4.1.5-C

2.1-D – Hard-coded passwords and keys

Voting system software must not contain hard-coded

1. Passwords, or
2. cryptographic keys

Discussion

Many examples of this vulnerability have previously been identified in voting system software. Additional information about this vulnerability can be found at CWE-259: Use of Hard-coded Password and CWE-321: Use of Hard-coded Cryptographic Key.

External references:	CWE-259: Use of Hard-coded Password CWE-321: Use of Hard-coded Cryptographic Key

2.2 – The voting system is implemented using best practice user-centered design methods, for a wide range of representative voters, including those with and without disabilities, and election workers.

2.2-A – User-centered design process

The manufacturer must submit a report providing documentation that the system was developed following best practices for a user-centered design process.

The report must include, at a minimum:

- A listing of user-centered design methods used
- The types of voters and election workers included in those methods
- How those methods were integrated into the overall implementation process
- How the results of those methods contributed to developing the final features and design of the voting system

Discussion

The goal of this requirement is to allow the manufacturer to demonstrate, through the report, the way their implementation process included user-centered design methods.

"ISO-9241-210:2010 Ergonomics of human-system interaction—Part 210: Human-centered design for interactive systems provides requirements and recommendations for human-centered principles and activities throughout the life-cycle of computer-based interactive systems." It includes the idea of iterative cycles of user research to understand the context of use and user needs, creating prototypes or versions, and testing to confirm that the product meets the identified requirements.

This requirement does not specify the exact user-centered design methods to be used, or their number or timing.

The ISO group of requirements, *Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Common Industry Format (CIF)* includes several standards that are a useful framework for reporting on user-centered design activities and usability reports.

- ISO/IEC TR 25060:2010: General framework for usability-related information
- ISO/IEC 25063:2014: Context of use description
- ISO/IEC 25062:2006: Usability test reports
- ISO/IEC 25064:2013: User needs report
- ISO/IEC 25066:2016 Evaluation report

The final research report from the Los Angeles *Voting Systems for All* People project is an example of a summary report of a user-centered design process to design a voting system.

External reference:	ISO 9241-210:2010 – Human-centered design for interactive systems

Related requirements:	8.3-A-Usability testing with voters 8.4-A-Usability testing with election workers

2.3 - Voting system logic is clear, meaningful, and well-structured.

2.3-A – Acceptable programming languages

Application logic must be produced in a high-level programming language that has all of the following control constructs:

1. Sequence
2. Loop with exit condition (for example, for, while, or do-loops)
3. If/Then/Else conditional
4. Case conditional
5. Block-structured exception handling (for example, try/throw/catch).

This requirement can be satisfied by using COTS extension packages to add missing control constructs to languages that could not otherwise conform.

Discussion
By excluding border logic, this requirement allows the use of assembly language for hardware-related segments, such as device controllers and handler programs. It also allows the use of an externally-imposed language for interacting with an Application Program Interface (API) or database query engine. However, the special code should be insulated from the bulk of the code, for example, by wrapping it in callable units expressed in the prevailing language to minimize the number of places that special code appears.

For example, C99 [ISO99] does not support block-structured exception handling, but the construct can be retrofitted using, for example [Sourceforge00] or another COTS package.

The use of non-COTS extension packages or manufacturer-specific code for this purpose is not acceptable, as it would place an unreasonable burden on the test lab to verify the soundness of an unproven extension (effectively a new programming language). The package needs to have a proven track record of performance supporting the assertion that it would be stable and suitable for use in voting systems, just as the compiler or interpreter for the base programming language does.

Prior VVSG source:	2007 6.4.1.2-A 2007 6.4.1.5-A.1

2.3-B – Acceptable coding conventions

Application logic must adhere to a published, credible set of coding rules, conventions or standards (called "coding conventions") that enhance the workmanship, security, integrity, testability, and maintainability of applications.

Discussion

Coding conventions that are excessively specialized or simply inadequate can be rejected on the grounds that they do not enhance workmanship, security, integrity, testability, or maintainability.

Prior VVSG source:	2007 6.4.1.3-A

2.3-C – Published coding conventions

Coding conventions must be considered published if they appear in a publicly available book, magazine, journal, or new media with analogous circulation and availability, or if they are publicly available on the Internet.

Discussion

This requirement attempts to clarify the "published, reviewed, and industry-accepted" language appearing in previous iterations of the VVSG, but the intent of the requirement is unchanged. Following are examples of published coding conventions (links valid as of 2007-02). These are only examples and are not necessarily the best available for the purpose.

1. Ada: Christine Ausnit-Hood, Kent A. Johnson, Robert G. Pettit, IV, and Steven B. Opdahl, Eds., Ada 95 Quality and Style, Lecture Notes in Computer Science #1344, Springer-Verlag, 1995-06. Content available at http://www.iste.uni-stuttgart.de/ps/ada-doc/style_guide/cover.html and elsewhere.
2. C++: Mats Henricson and Erik Nyquist, Industrial Strength C++, Prentice-Hall, 1997. Content available at http://hem.passagen.se/erinyq/industrial/.
3. C#: "Design Guidelines for Class Library Developers," Microsoft. http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp.
4. Java: "Code Conventions for the Java™ Programming Language," Sun Microsystems. http://java.sun.com/docs/codeconv/.

Prior VVSG source:	2007 6.4.1.3-A.1

2.3-D – Credible

Coding conventions must be considered credible if at least two different organizations with no ties to the creator of the rules or to the manufacturer seeking conformity assessment, independently decided to adopt them and made active use of them within the three years before conformity assessment was first sought. These two organizations cannot be voting equipment manufacturers.

Discussion
This requirement attempts to clarify the "published, reviewed, and industry-accepted" language appearing in previous iterations of the VVSG, but the intent of the requirement is unchanged.

Coding conventions evolve, and it is desirable for voting systems to be aligned with modern practices. If the "three-year rule" was satisfied at the time that a system was first submitted for testing, it is considered satisfied for the purpose of subsequent reassessments of that system. However, new systems need to meet the three-year rule as of the time that they are first submitted for testing, even if they reuse parts of older systems.

Prior VVSG source:	2007 6.4.1.3-A.2

2.3-D – Header comments

Voting system software must have the capability to include header comments that provide at least the following information for each callable unit (function, method, operation, subroutine, procedure, etc.):

- The purpose of the unit and how it works (if not obvious);
- A description of input parameters, outputs and return values, exceptions thrown, and side-effects;
- Any protocols that must be observed (for example, unit calling sequences);
- File references by name and method of access (read, write, modify, append, etc.);
- Global variables used (if applicable);
- Audit event generation;
- Date of creation; and
- Change log (revision record).

Discussion
Header comments and other commenting conventions should be specified by the selected coding conventions consistent with the idiom of the programming language chosen. If the coding conventions specify a coding style and commenting convention that make header comments redundant, then they can be omitted. Otherwise, if the coding conventions fail to specify the content of header comments, the non-redundant portions of this generic guideline should be applied.

Change logs need not cover the nascent period, but they do need to go back as far as the first baseline or release that is submitted for testing, and should go back as far as the first baseline or release that is deemed reasonably coherent.

Prior VVSG source: 2007 6.4.1.6-A

2.4 - Voting system structure is modular, scalable, and robust.

2.4-A – Modularity

Application logic must be designed in a modular fashion.

Discussion

The modularity rules described under the 2.4 guideline apply to the component submodules of a library.

Prior VVSG source:	2007 6.4.1.4-A

2.4-B – Module testability

Each module must have a specific function that can be tested and verified independently of the remainder of the code.

Discussion

In practice, some additional modules (such as library modules) can be needed to compile the module being tested, but the modular construction allows the supporting modules to be replaced by special test versions that support test objectives.

Prior VVSG source:	2007 6.4.1.4-A.1

2.4-C – Module size and identification

Modules must be small and easily identifiable.

Prior VVSG source:	2007 6.4.1.4-B

2.4-D – Callable unit length limit

The code length of callable units (such as functions, methods, operations, subroutines, and procedures) excluding comments, blank lines, and initializers for read-only lookup tables, must be limited in length to no more than:

- 25 lines of code for 50 % of all callable units
- 60 lines of code for 5 % of all callable units
- 180 lines of code for any callable unit

Discussion

"Lines," in this context, are defined as executable statements or flow control statements with suitable formatting.

Prior VVSG source:	2007 6.4.1.4-B.1

2.4-E– Lookup tables in separate files

Read-only lookup tables longer than 25 lines must be placed in separate files from other source code if the programming language permits it.

Prior VVSG source:	2007 6.4.1.4-B.2

2.4-F– Limited cylcomatic complexity

The cyclomatic complexity measure of any given module must not exceed 15.

Discussion
Cyclomatic complexity measures the amount of decision logic in a source code function. It is completely independent of text formatting and is nearly independent of programming language since the same fundamental decision structures are available and uniformly used in all procedural programming languages [NIST SP 500-235]. Modules with high cyclomatic complexity measures tend to be difficult to understand, test, and maintain.
https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-235.pdf

External reference:	NIST SP 500-235

2.5 - The voting system supports system processes and data with integrity.

2.5.1– Code coherency

2.5.1-A – Self-modifying code

Application logic must not be self-modifying.

Prior VVSG source: 2007 6.4.1.7-A.1

2.5.1-B – Unsafe concurrency

Application logic must be free of race conditions, deadlocks, livelocks, and resource starvation.

Prior VVSG source: 2007 6.4.1.7-A.2

2.5.1-C – Code integrity

If compiled code is used, it must only be compiled using a COTS compiler.

Discussion
This prohibits the use of arbitrary, nonstandard compilers and, consequently, the invention of new programming languages.

Prior VVSG source: 2007 6.4.1.7-A.3

2.5.1-D – Interpreted code, specific COTS interpreter

If interpreted code is used, it must only be run under a specific, identified version of a COTS runtime interpreter.

Discussion
This ensures that…

- no arbitrary, nonstandard interpreted languages are used, and
- the software tested and approved during the conformity assessment process does not change behavior because of a change to the interpreter.

Prior VVSG source: 2007 6.4.1.7-A.4

2.5.2 – Prevent tampering

2.5.2-A – Prevent tampering with code

Programmed devices must prevent replacement or modification of executable or interpreted code (for example, by other programs on the system, by people physically replacing the memory or medium containing the code, or by faulty code) except where this access is necessary to conduct the voting process.

Discussion

This requirement can be partially satisfied through a combination of…

- read-only memory (ROM),
- the memory protection implemented by most popular COTS operating systems,
- error checking as described in Part 1:6.4.1.8 "Error checking", and
- access and integrity controls.

Prior VVSG source:	2007 6.4.1.7-B

2.5.2-B – Prevent tampering with data

All voting devices must prevent access to or manipulation of configuration data, vote data, or audit records (for example, by physical tampering with the medium or mechanism containing the data, by other programs on the system, or by faulty code) except where this access is necessary to conduct the voting process.

Discussion

This requirement can be partially satisfied through a combination of…

- the memory protection implemented by most popular COTS operating systems,
- error checking as described in Part 1:6.4.1.8 "Error checking", and
- access and integrity controls.

Systems using mechanical counters to store vote data need to protect the counters from tampering. If vote data are stored on paper, the paper needs to be protected from tampering. Modification of audit records after they are created is never necessary.

Prior VVSG source:	2007 6.4.1.7-C

2.5.3 – Monitor errors

2.5.3-A – Monitor I/O errors

Programmed devices must provide the capability to monitor the transfer quality of I/O operations, reporting the number and types of errors that occur and how they were corrected.

Prior VVSG source:	2007 6.4.1.7-D

2.5.3-B – Detect garbage input

Programmed devices must check information inputs for completeness and validity.

Discussion
This general requirement applies to all programmed devices, while the specific ones following are only enforceable for application logic.

Prior VVSG source:	2007 6.4.1.8-A

2.5.3-C – Defend against garbage input

Programmed devices must ensure that incomplete or invalid inputs do not lead to irreversible error.

Prior VVSG source:	2007 6.4.1.8-A.1

2.5.3-D – Validate and filter input

The voting system must validate all input against expected parameters, such as data presence, length, type, format, uniqueness, or inclusion in a set of whitelisted values.

Discussion
Input includes data from any input source: input devices (such as touchscreens, keyboards, keypads, optical/digital scanners, and assistive devices), networking port, data port, or file.

Prior VVSG source:	2007 6.4.1.8-A.1

2.5.4 – Output

2.5.4-A – Escaping and encoding output

Software output must be properly encoded and escaped.

Discussion
The output of a software module can be manipulated or abused by attackers in unexpected ways to perform malicious actions. Ensuring that outputted data is of an expected type or format assists in preventing this abuse. Additional information about this software weakness can be viewed in CWE 116: Improper Encoding or Escaping of Output.

External sources:	CWE 116: Improper Encoding or Escaping of Output

2.5.4-B – Sanitize output

The voting system must sanitize all output to remove or neutralize the effects of any escape characters, control signals, or scripts contained in the data which could adversely manipulate the output source.

Discussion
Output includes data to any output source: output devices (such as touchscreens, LCD screens, printers, and assistive devices), networking port, data port, or file. This applies to all parts of the voting system including the EMS.

2.5.4-C – Stored injection

The voting system must sanitize all output to files and databases to remove or neutralize the effects of any escape characters, control signals, or scripts contained in the data which could adversely manipulate the voting system if the stored data is read or imported at a later date or by another part of the voting system.

Discussion
A stored injection attack saves malicious data which is harmless when stored, but which is potent when read later in a different context or when converted to a different format. For example, a malicious script might be written to a file and do no harm to the voting machine, but later be evaluated and harmful when the file is transferred and read by the EMS. Input should also be filtered, but sanitizing stored output provides defense in depth. Examples of stored injection include

2.5.5 – Error checking

2.5.5-A – Mandatory internal error checking

Application logic that is vulnerable to the following types of errors must check for these errors at run time and respond defensively when they occur:

1. Common memory management errors, such as out-of-bounds accesses of arrays, strings, and buffers used to manage data
2. Uncontrolled format strings
3. CPU-level exceptions such as address and bus errors, dividing by zero, and the like
4. Variables that are not appropriately handled when out of expected boundaries
5. Numeric and integer overflows
6. Validation of array indices
7. Known programming language specific vulnerabilities

Discussion

Logic verification will show that some error checks cannot logically be triggered, and some exception handlers cannot logically be invoked. These checks and exception handlers are not redundant – they provide defense-in-depth against faults that escape detection during logic verification.

Prior VVSG source:	2007 6.4.1.8-B

2.5.5-B – Array overflows

If the application logic uses arrays, vectors, or any analogous data structures, and the programming language does not provide automatic run-time range checking of the indices, the indices must be ranged-checked on every access.

Discussion

Range checking code should not be duplicated before each access. Clean implementation approaches include:

- consistently using dedicated accessors (such as functions, methods, operations, subroutines, and procedures) that range-check the indices;
- defining and consistently using a new data type or class that encapsulates the range-checking logic;
- declaring the array using a template that causes all accessors to be range-checked; or
- declaring the array index to be a data type whose enforced range is matched to the size of the array.

Range-enforced data types or classes can be provided by the programming environment or they can be defined in application logic. If acceptable values of the index do not form a contiguous range, a map structure can be more appropriate than a vector.

Prior VVSG source:	2007 6.4.1.8-B.1

2.5.5-C – Buffer overflows

If an overflow does not automatically result in an exception, the application logic must explicitly check for and prevent the overflow.

Discussion

Embedded system developers use a variety of techniques for avoiding stack overflow. Commonly, the stack is monitored, and warnings and exceptions are thrown when thresholds are crossed. In non-embedded contexts, stack overflow often manifests as a CPU-level exception related to memory segmentation. Types of software weakness covered under this requirement include CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'), CWE-121: Stack-based Buffer Overflow, and CWE-122: Heap-based Buffer Overflow.

Prior VVSG sources:	2007 6.4.1.8-B.2

2.5.5-D – CPU traps

The application logic must implement such handlers as needed to detect and respond to CPU-level exceptions.

Discussion

For example, under Unix, a CPU-level exception would manifest as a signal, so a signal handler is needed. If the platform supports it, it is preferable to translate CPU-level exceptions into software-level exceptions so that all exceptions can be handled in a consistent fashion within the voting application. However, not all platforms support it.

Prior VVSG source:	2007 6.4.1.8-B.3

2.5.5-E – Garbage input parameters

All scalar or enumerated type parameters whose valid ranges as used in a callable unit (such as function, method, operation, subroutine, and procedure) do not cover the entire ranges of their declared data types must be range-checked on entry to the unit.

Discussion

This applies to parameters of numeric types, character types, temporal types, and any other types for which the concept of range is well-defined.[7] In cases where the restricted range is frequently used or associated with a meaningful concept within the scope of the application, the best approach is to define a new class or data type that encapsulates the range restriction, eliminating the need for range checks on each use.

This requirement differs from Requirement Part 1:6.4.1.8-A, which deals with user input that is expected to contain errors. This requirement deals with program internal parameters, which are expected to conform to the expectations of the designer. User input errors are a normal occurrence; the errors discussed here are grounds for throwing exceptions.

Prior VVSG source:	2007 6.4.1.8-B.4

2.5.5-F – Numeric overflows

If the programming language does not provide automatic run-time detection of numeric overflow, all arithmetic operations that could potentially overflow the relevant data type must be checked for overflow.

Discussion

Encapsulate overflow checking as much as possible.

Prior VVSG source:	2007 6.4.1.8-B.5

2.5.5-G – Uncontrolled format strings

Voting system software must not contain uncontrolled format strings.

Discussion

Many examples of this vulnerability have previously been identified in voting system software. Additional information about this vulnerability can be found at CWE 134: Use of Externally-Controlled Format String.

External reference:	CWE 134: Use of Externally-Controlled Format String

2.5.5-H – Recommended internal error checking

Application logic that is vulnerable to the following types of errors must check for these errors at run time and respond defensively when they occur:

1. Pointer variable errors
2. Dynamic memory allocation and management errors

Prior VVSG source:	2007 6.4.1.8-C

2.5.5-I – Pointers

If application logic uses pointers or a similar mechanism for specifying absolute memory locations, the application logic must validate these pointers or addresses before they are used.

Discussion

Prevent improper overwriting. Even if read-only memory would prevent the overwrite from succeeding, an attempted overwrite indicates a logic fault that must be corrected.

Pointer use that is fully encapsulated within a standard platform library is treated as COTS software.

Prior VVSG source: 2007 6.4.1.8-C.1

2.5.5-J – Memory mismanagement

If dynamic memory allocation is performed in application logic, the application logic must be able to be instrumented or analyzed with a COTS tool for detecting memory management errors.

Discussion

Dynamic memory allocation that is fully encapsulated within a standard platform library is treated as COTS software. This is "should" not "must" only because such tooling might not be available or applicable in all cases.

Prior VVSG source: 2007 6.4.1.8-D

2.5.5-K – Nullify freed pointers

If pointers are used, any pointer variables that remain within scope after the memory they point to is deallocated must be set to null or marked as invalid (pursuant to the idiom of the programming language used).

Discussion

If this is not done automatically by the programming environment, a callable unit should be dedicated to the task of deallocating memory and nullifying pointers. Equivalently, "smart pointers" like the C++ std::auto_ptr can be used to avoid the problem. One should not add assignments after every deallocation in the source code.

In languages using garbage collection, memory is not deallocated until all pointers to it have gone out of scope, so this requirement is moot.

Prior VVSG source: 2007 6.4.1.8-E

2.5.5-L – React to errors detected

Detecting any of the errors enumerated in the previous 2.5.5 requirements must be treated as a complete failure of the callable unit in which the error was detected.

1. An appropriate exception must be thrown, and

2. Control must pass out of the unit immediately.

Prior VVSG source:	2007 6.4.1.8-F

2.5.5-M –Error checks

Error checks detailed in the previous 2.5.5 requirements must remain active in production code.

Discussion
These errors are incompatible with voting integrity, so masking them is unacceptable.

Manufacturers should not implement error checks using the C/C++ assert() macro. It is often disabled, sometimes automatically, when software is compiled in production mode. Furthermore, it does not appropriately throw an exception, but instead aborts the program.

"Inevitably, the programmed validity checks of the defensive programming approach will result in run-time overheads and, where performance demands are critical, many checks are often removed from the operational software; their use is restricted to the testing phase where they can identify the misuse of components by faulty designs. In the context of producing complex systems which can never be fully tested, this tendency to remove the protection afforded by programmed validity checks is most regrettable and is not recommended here." [Moulding89]

Prior VVSG source:	2007 6.4.1.8-G

2.5.5-N – Election integrity monitoring

To the extent possible, electronic devices must proactively detect or prevent basic violations of election integrity (for example, stuffing the ballot box or accumulating negative votes) and alert an election official or administrator if they occur.

Discussion
Equipment can only verify those conditions that are within the scope of what the equipment does. However, if the equipment can detect something that is blatantly wrong, it should do so and raise the alarm. This provides defense-in-depth to supplement procedural controls and auditing practices.

Prior VVSG source:	2007 6.4.1.8-K

2.5.6 – Data structure

2.5.6-A – SQL injection

The voting system application must defend against SQL injection.

Discussion
SQL injection is a classic type of software weakness still prevalent today. SQL injection is not just a web-based issue, as any application accepting untrusted user input and passing it to a database can be vulnerable. Additional information about this software weakness can be viewed in within CWE 89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

External source:	CWR 89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

2.5.6-B – Parameterized queries

Any structured statement or command being prepared using dynamic data (including user input) to be sent to a database or other process must parameterize the data inputs and apply strict type casting and content filters on the data (such as, prepared statements).

Discussion
Parametrized queries are a common defense against this class of software weakness.

2.6 - The voting system handles errors robustly and gracefully recovers from failure.

2.6-A – Block-structured exception handling

Application logic must handle exceptions using block-structured exception handling constructs.

Discussion

Block-structured exception handling "…is the ability to associate exception handlers with blocks of logic, and implicitly, the presence of the exception concept in the programming language." (This simply means try/throw/catch or equivalent statements and should not be confused with the specific implementation known as Structured Exception Handling (SEH) [Pietrek97].[2]) Unlike deeply nested blocks, exceptions cannot be eliminated by restructuring logic. "When exceptions are not used, the errors cannot be handled but their existence is not avoided." [ISO00a]

Prior VVSG source: 2007 6.4.1.5-A

2.6-B – Wrapping legacy library units

If application logic makes use of any COTS that do not throw exceptions when exceptional conditions occur, those units need to be wrapped in callable units that check for the relevant error conditions and translate them into exceptions, and the remainder of application logic need to use only the wrapped version.

Discussion

For example, if an application written in C99 [ISO99] + cexcept [Sourceforge00] used the malloc function of libc, which returns a null pointer in case of failure instead of throwing an exception, the malloc function would need to be wrapped. Here is one possible implementation:

```
void *checkedMalloc (size_t size) {
    void *ptr = malloc (size);
    if (!ptr)
            Throw bad_alloc;
    return ptr;
}
#define malloc checkedMalloc
```

Wrapping legacy functions avoids the need to check for errors after every invocation, which both obfuscates the application logic and creates a high likelihood that some or many possible errors will not be checked for.

In C++, it would be preferable to use one of the newer mechanisms that already throw exceptions on failure and avoid use of legacy functions altogether.

Prior VVSG source:	2007 6.4.1.5-A.1

2.6-C – Intentional exceptions

Exceptions must only be used for abnormal conditions, and not be used to redirect the flow of control in normal ("non-exceptional") conditions.

Discussion
"Intentional exceptions" cannot be used as a substitute for arbitrary branch. Normal, expected events, such as reaching the end of a file that is being read from beginning to end or receiving invalid input from a user interface, are not exceptional conditions and should not be implemented using exception handlers.

Prior VVSG source:	2007 6.4.1.5-B.2

2.6-D – Unstructured exception handling

Unstructured exception handling (e.g., On Error GoTo, setjmp/longjmp, or explicit tests for error conditions after every executable statement) must not be used.

Discussion
The internal use of such constructs by a COTS extension package that adds block-structured exception handling to a programming language that otherwise would not have it, as described in Requirement Part 1:6.4.1.2-A.1, is allowed. Similarly, it is not a problem that source code written in a high-level programming language is compiled into low-level machine code that contains arbitrary branches. It is only the direct use of low-level constructs in application logic that presents a problem.

Prior VVSG source:	2007 6.4.1.5-B.3

2.6-E – Surviving device failure

All systems must be capable of resuming normal operation following the correction of a failure in any device.

Prior VVSG source:	2007 6.4.1.9-A

2.6-F – No compromising voting or audit data

Exceptions and system recovery must be handled in a manner that protects the integrity of all recorded votes and audit log information.

Prior VVSG source: 2007 6.4.1.9-A

2.6-G – Surviving component failure

All voting devices must be capable of resuming normal operation following the correction of a failure in any component (for example, memory, CPU, ballot reader, or printer) provided that catastrophic electrical or mechanical damage has not occurred.

Prior VVSG source: 2007 6.4.1.9-C

2.6-H – Controlled recovery

Error conditions must be corrected in a controlled fashion so that system status can be restored to the initial state existing before the error occurred.

Discussion
"Initial state" refers to the state existing at the start of a logical transaction or operation. Transaction boundaries must be defined in a conscientious fashion to minimize the damage. Language changed to "can" because election officials responding to the error condition might want the opportunity to select a different state (such as a controlled shutdown with memory dump for later analysis).

Prior VVSG source: 2007 6.4.1.9-D

2.6-I – Nested error conditions

Nested error conditions that are corrected without reset, restart, reboot, or shutdown of the voting device must be corrected in a controlled sequence so that system status can be restored to the initial state existing before the first error occurred.

Prior VVSG source: 2007 6.4.1.9-D.1

2.6-J – Reset CPU error states

CPU-level exceptions that are corrected without reset, restart, reboot, or shutdown of the voting device must be handled in a manner that restores the CPU to a normal state and allows the system to log the event and recover as with a software-level exception.

Discussion
System developers should test to see how CPU-level exceptions are handled and make any changes necessary to ensure robust recovery. Invocation of any other error routine while the CPU is in an exception handling state is to be avoided – software error handlers often do not operate as intended when the CPU is in an exception handling state.

If the platform supports it, it is preferable to translate CPU-level exceptions into software-level exceptions so that all exceptions can be handled in a consistent fashion within the voting application. However, not all platforms support it.

Prior VVSG source: 2007 6.4.1.9-D.2

2.6-K – Coherent checkpoints

When recovering from non-catastrophic failure of a device or from any error or malfunction that is within the operator's ability to correct, the system must restore the device to the operating condition existing immediately before the error or failure, without loss or corruption of voting data previously stored in the device.

Discussion

If, as discussed in Requirement Part 1:6.4.1.9-D, the system is left in something other than the last known good state for diagnostic reasons, this requirement clarifies that it must revert to the last known good state before being placed back into service.

Prior VVSG source: 2007 6.4.1.9-E

2.7 - The voting system performs reliably in anticipated physical environments.

The requirements in this section deal with the capability of voting devices to withstand electrical disturbances as well as for voting devices to not cause electrical disturbances in other devices or people. Conformance to the Federal Communications Commission, Part 15, Class B requirements largely deals with these issues, and the requirements here address items not covered by Class B.

2.7-A Electrical disturbances

All voting devices must continue to operate in the presence of electrical disturbances generated by other devices and people and must not cause electrical disruption to other devices and people.

Discussion
Voting devices located in a polling place or other places need to continue to operate despite disruption from electrical emanations generated by other devices, including static discharges from people. Likewise, voting devices need to operate without causing disruption to other devices and people due to electrical emanations from the devices.

2.7-A.1 FCC Part 15 Class A and B conformance

Voting devices must comply with the requirements of the Federal Communications Commission, Part 15:

1. Voting devices located in polling places must comply with Class B requirements.
2. Voting devices located in non-place setting, e.g., back offices, must minimally comply with Class A requirements.

2.7-A.2 Power supply – energy service provider

Voting devices located in polling places must be powered by a 120 V, single phase power supply derived from typical energy service providers.

Discussion
It is assumed that the AC power necessary to operate the voting system will be derived from the existing power distribution system of the facility housing the polling place. This single-phase power may be a leg of a 120/240 V single phase system, or a leg of a 120/208 V three-phase system, at a frequency of 60 Hz.

2.7-A.3 Power port connection to the facility power supply

All electronic voting systems installed in a polling place must comply with Class B emission limits affecting the power supply connection to the energy service provider.

Discussion

The normal operation of an electronic system can produce disturbances that will travel upstream and affect the power supply system of the polling place, creating a potential deviation from the expected electromagnetic compatibility of the system. The issue is whether these actual disturbances (after possible mitigation means incorporated in the equipment) reach a significant level to exceed stipulated limits.

2.7-A.4 Leakage via grounding port

All electronic voting systems installed in a polling place must comply with limits of leakage currents effectively established by the trip threshold of all listed Ground Fault Current Interrupters (GFCI), if any, installed in the branch circuit supplying the voting system.

Discussion

Excessive leakage current is objectionable for two reasons:

For a branch circuit or wall receptacle that could be provided with a GFCI (depending upon the wiring practice applied at the particular polling place), leakage current above the GFCI built-in trip point would cause the GFCI to trip and therefore disable the operation of the system.

Should the power cord lose the connection to the equipment grounding conductor of the receptacle, a personnel hazard would occur. (Note the prohibition of "cheater" adapters in the discussion of general requirements for the polling place.)

2.7-A.5 Outages, sags and swells

All electronic voting systems must be able to withstand, without disruption of normal operation or loss of data, a complete loss of power lasting two hours.

Discussion

The Information Technology industry has adopted a recommendation that IT equipment should be capable to operate correctly for swells reaching 120 % of the nominal system voltage with duration ranging from 3 ms to 0.5 s and permanent overvoltages up to 110 % of nominal system voltage.

2.7-A.6 Withstand conducted electrical disturbances

All electronic voting systems shall withstand conducted electrical disturbances that affect the power ports of the system.

2.7-A.7 Emissions from other connected equipment

All elements of an electronic voting system shall be able to withstand the conducted emissions generated by other elements of the voting system.

2.7-A.8 Electrostatic discharge immunity

All electronic voting systems shall withstand, without disruption of normal operation or loss of data, electrostatic discharges associated with human contact and contact with mobile equipment (service carts, wheelchairs, etc.).

Discussion

Electrostatic discharge (ESD) events can originate from direct contact between an "intruder" (person or object) charged at a potential different from that of the units of the voting system, or from an approaching person about to touch the equipment – an "air discharge." The resulting discharge current can induce disturbances in the circuits of the equipment. This requirement is meant to ensure that voting devices are conformant to the typical ESD specifications met by other electronic devices used by the public such as ATMs and vending kiosks.

2.7-A.9 Radiated radio frequency emissions

All electronic voting systems installed in a polling place shall comply with emission limits according to the Rules and Regulations of Class B for radiated radio-frequency emissions.

Discussion

Electronic equipment in general and modern high-speed digital electronic circuits in particular have the potential to produce unintentional radiated and conducted radio-frequency emissions over wide frequency ranges. These unintentional signals can interfere with the normal operation of other equipment, especially radio receivers, in close proximity.