ISO/IEC JTC 1/SC 22/OWGV N 0082

James W. Moore and Robert Seacord, "Secure Coding becomes Standard," presentation to Systems and Software Technology Conference (SSTC), June 19, 2007

Date	2 July 2007
Contributed by	James W. Moore and Robert Seacord
Original file name	SSTC2007 rcs jwm 2a fullf.pdf
Notes	

Secure Coding Becomes Standard

James W. Moore, F-IEEE, CSDP
Robert C. Seacord

Agenda

Why Care about Software Security?

CERT Secure Coding Standards

SC22 OWGV

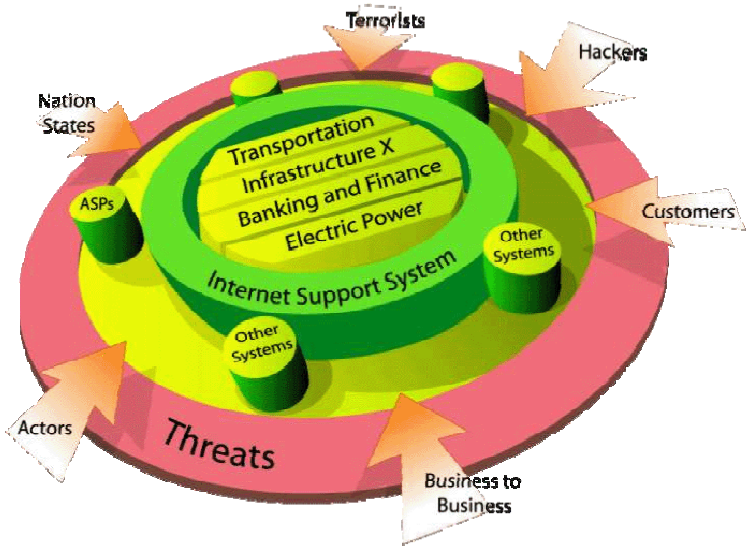
Why Care about Software Security?

Vulnerabilities allow attackers to compromise information security

- confidentiality
- integrity
- availability

Accurate risk assessment requires knowledge about vulnerabilities

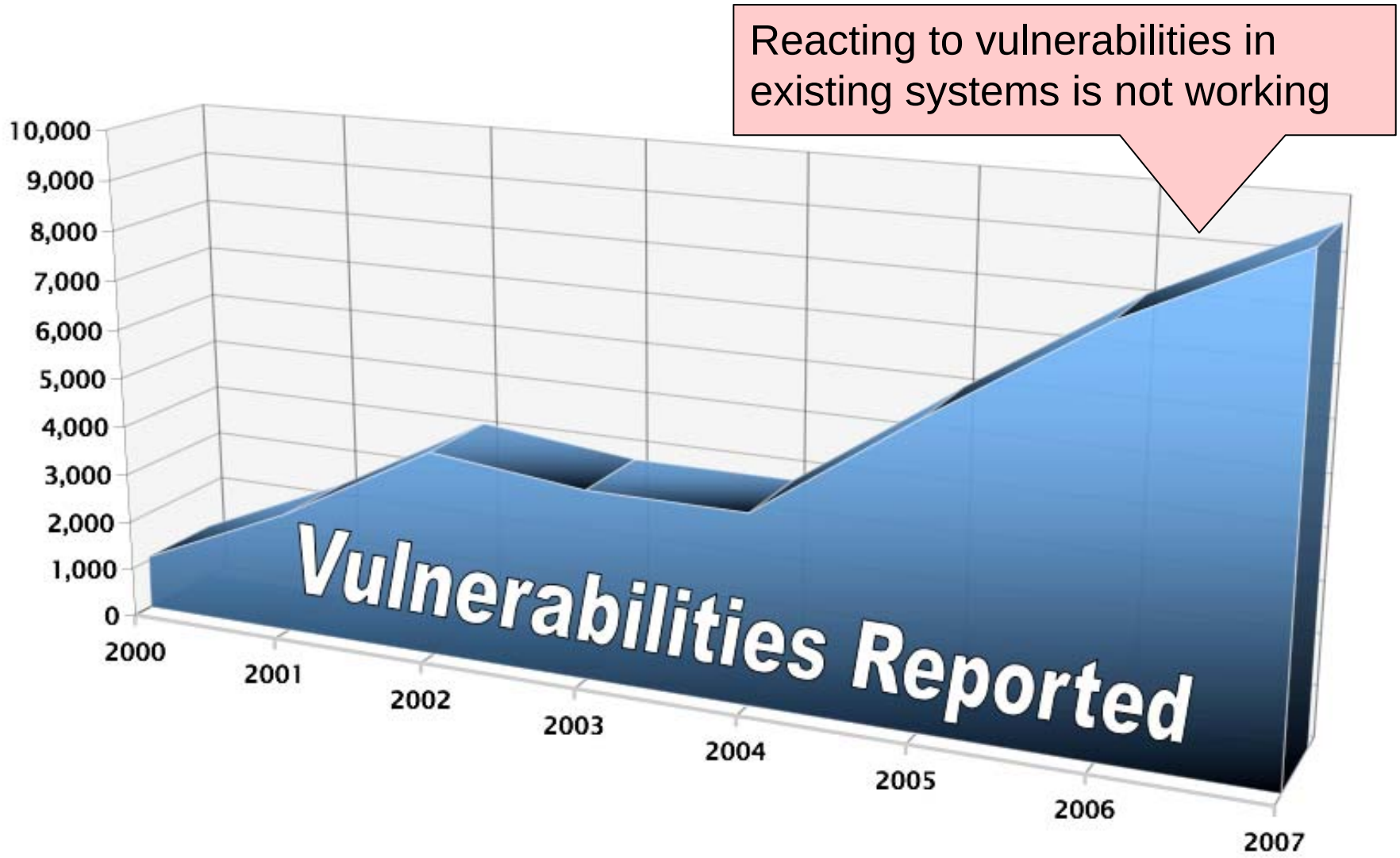
- prerequisite conditions
- technical details
- impacts and mitigation strategies



Increased risk to information and communication systems

- critical infrastructures are affected (and often unprepared to respond)
- software used by control systems vulnerable to attack
- convergence of common technologies
- adversaries leverage failures in technology and people to conduct criminal activity
- economic and physical consequences of cyber attacks

Increasing Vulnerabilities



Most Vulnerabilities caused by Programming Errors

64% of the vulnerabilities in NVD in 2004 are due to programming errors

- 51% of those due to classic errors like buffer overflows, cross-site-scripting, injection flaws
- *Heffley/Meunier (2004): Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?*

Cross-site scripting, SQL injection at top of the statistics (CVE, Bugtraq) in 2006

"We wouldn't need so much network security if we didn't have such bad software security"

--Bruce Schneier

Information Warriors

8 nations have developed cyber-warfare capabilities comparable to that of the United States.

More than 100 countries are trying to develop them. 23 nations have targeted U.S. systems.

North Korea, Libya, Iran, and Syria reportedly have some capability.

Russia, China, India, and Cuba have acknowledged policies of preparing for cyber-warfare and are rapidly developing their capabilities.

China is moving aggressively toward incorporating cyber-warfare into its military lexicon, organization, training, and doctrine. It has the capability to penetrate poorly protected U.S. computer systems and potentially could use computer network attacks to strike specific U.S. civilian and military infrastructures.

Agenda

Why Care about Software Security?

CERT Secure Coding Standards

SC22 OWGV

Unexpected Integer Values

An unexpected value is one you would not expect to get using a pencil and paper

Unexpected values are a common source of software vulnerabilities.

Fun with Integers

```
char x, y;
x = -128;
y = -x;

if (x == y) puts("1");
if ((x - y) == 0) puts("2");
if ((x + y) == 2 * x) puts("3");
if (((char)(-x) + x) != 0) puts("4");
if (x != -y) puts("5");
```

CERT Secure Coding Standards

Identify coding practices that can be used to improve the security of software systems under development

Coding practices are classified as either rules or recommendations

- Rules need to be followed to claim compliance.
- Recommendations are guidelines or suggestions.

Development of Secure Coding Standards is a community effort

Scope

The secure coding standards proposed by CERT are based on documented standard language versions as defined by official or *de facto* standards organizations.

Secure coding standards are under development for:

- C programming language (ISO/IEC 9899:1999)
- C++ programming language (ISO/IEC 14882-2003)

Applicable technical corrigenda and documented language extensions such as the ISO/IEC TR 24731 extensions to the C library are also included.

Secure Coding Web Site (Wiki)

Secure Coding

CERT Secure Coding Standards

Added by Confluence Administrator, last edited by Robert Seacord on Feb 28, 2007 (view change)
Labels: (None)

Welcome to the Secure Coding Web Site

This web site exists to support the development of secure coding standards for commonly used programming languages such as C and C++. These standards are being developed through a broad-based community effort including the CERT Secure Coding Initiative and members of the software development and software security communities. For a further explanation of this effort please read the Rationale.

As this is a development web site, many of the pages on this web site are incomplete or contain errors. If you are interested in furthering this effort you may comment on existing items or send recommendations to secure-coding at cert dot org. You may also apply for an account to directly edit content on the site. Before using this site, please familiarize yourself with the Terms and Conditions.

The Top 10 Secure Coding Practices provides some language independent recommendations.

Secure Coding Standards

CERT C Programming Language Secure Coding Standard

CERT C++ Programming Language Secure Coding Standard

We would like to acknowledge the contributions of the following folks, and we look forward to seeing your name there as well.

http://www.securecoding.cert.org

Rules

Coding practices are defined as rules when

- Violation of the coding practice will result in a security flaw that may result in an exploitable vulnerability.
- There is an enumerable set of exceptional conditions (or no such conditions) where violating the coding practice is necessary to ensure the correct behavior for the program.
- Conformance to the coding practice can be verified.

MEM31-C. Free dynamically allocated memory exactly once

Added by Robert C. Seacord, last edited by Robert Seacord on Mar 30, 2007 (view change)
Labels: (None) EDIT

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted in a way that could introduce security vulnerabilities into a program. These types of issues are referred to as double-free vulnerabilities. In practice, double-free vulnerabilities can be exploited to execute arbitrary code. VU#623332, which describes a double-free vulnerability in the MIT Kerberos 5 function krb5_recvauth(), is one example. To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly one time. Programmers should be wary when freeing memory in a loop or conditional statement; if coded incorrectly, these constructs can lead to double-free vulnerabilities.

Non-Compliant Code Example

In this example, the memory referred to by `x` may be freed twice: once if `error_condition` is true and again at the end of the code.

```
x = malloc (number * sizeof(int));
if (x == NULL) {
  /* Handle Allocation Error */
}
/* ... */
if (error_conditon == 1) {
  /* Handle Error Condition*/
  free(x);
}
/* ... */
free(x);
```

MEM31-C. Compliant Solution

Compliant Solution

Only free a pointer to dynamic memory referred to by `x` once. This is accomplished by removing the call to `free()` in the section of code executed when `error_condition` is true.

```
x = malloc (number * sizeof(int));
if (x == NULL) {
  /* Handle Allocation Error */
}
/* ... */
if (error_conditon == 1) {
  /* Handle Error Condition*/
}
/* ... */
free(x);
```

Recommendations

Coding practices are defined as recommendations when

- Application of the coding practice is likely to improve system security.
- One or more of the requirements necessary for a coding practice to be considered as a rule cannot be met.

MEM00-A. Allocate and free memory in the same module, at the same level of abstraction

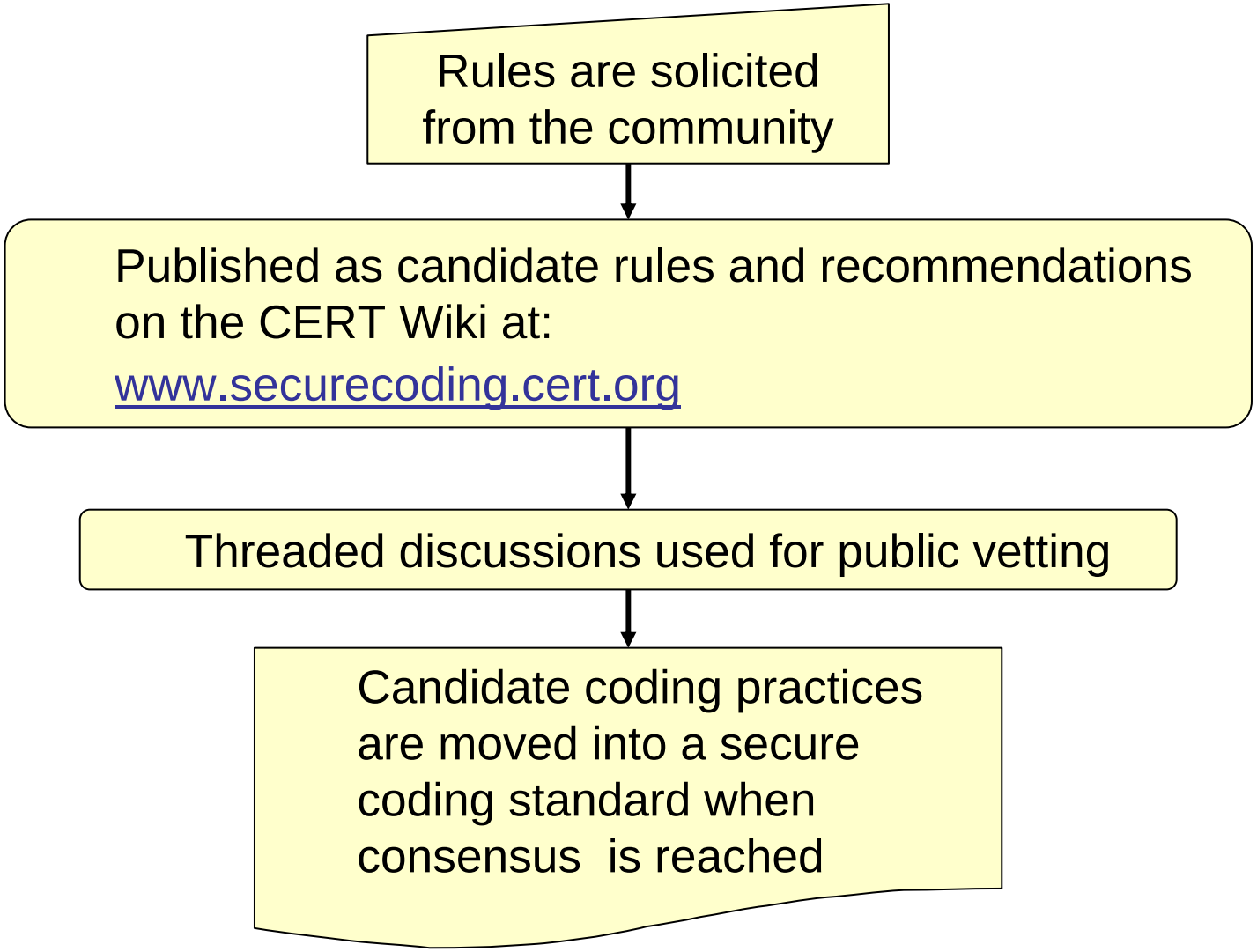
Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory.

This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or writing to unallocated memory.

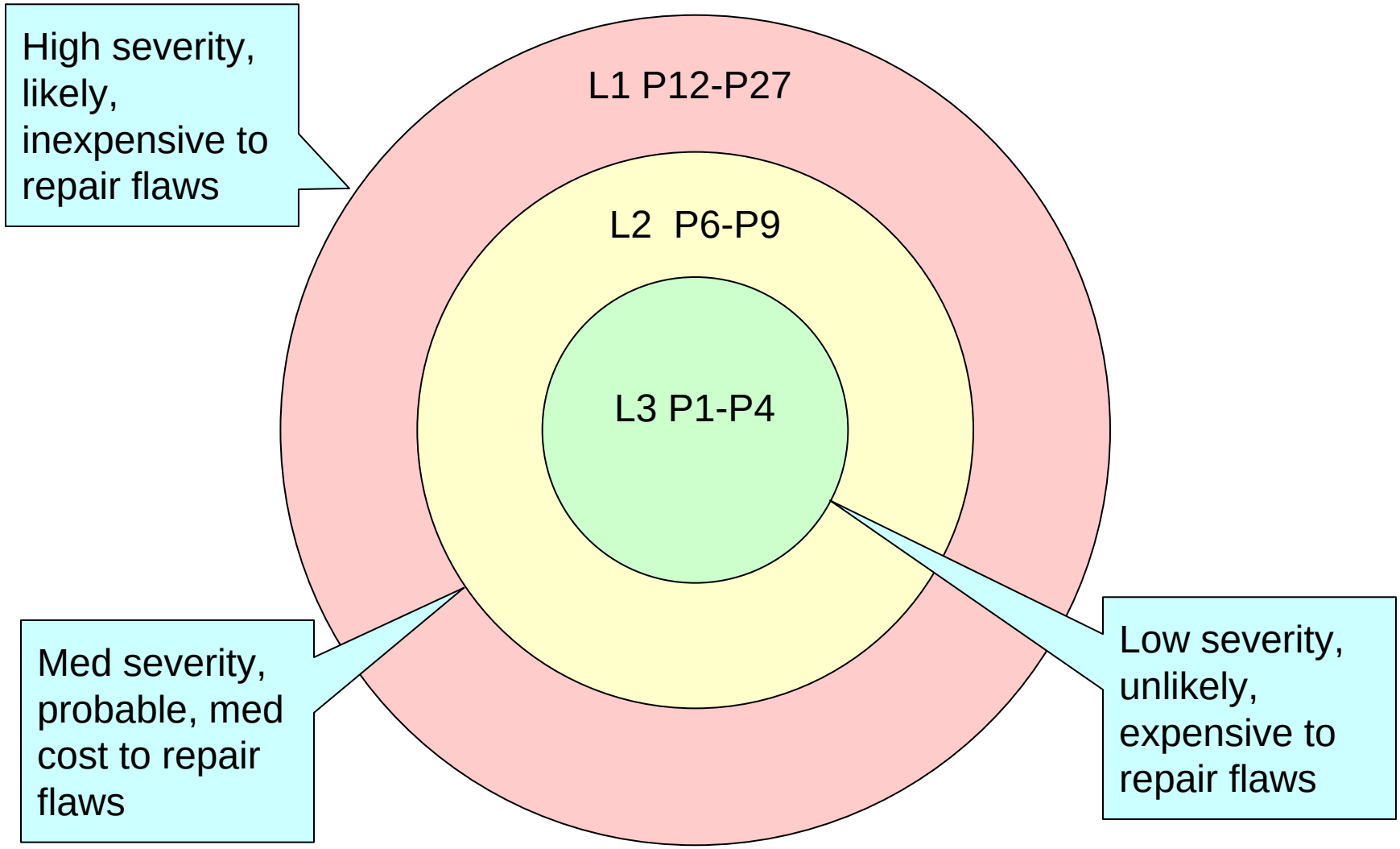
To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

Freeing memory in different modules resulted in a vulnerability in MIT Kerberos 5 MITKRB5-SA-2004-002 .

Community Development Process



Priorities and Levels



Risk-Based Triage of Rules

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	3 (high)	3 (probable)	3 (low)	P27	L1
FIO32-C	3 (high)	2 (probable)	1 (medium)	P6	L2
FIO33-C	1 (low)	1 (low)	3 (medium)	P3	L3
FIO34-C	2 (medium)	2 (probable)	2 (medium)	P8	L2
FIO35-C	1 (low)	1 (unlikely)	2 (medium)	P2	L3
FIO36-C	1 (low)	1 (unlikely)	3 (low)	P3	L3
FI037-C	3 (high)	1 (unlikely)	2 (medium)	P6	L3
FIO38-C	2 (medium)	2 (probable)	2 (medium)	P8	L2
FIO39-C	2 (medium)	2 (probable)	2 (medium)	P8	L2
FIO40-C	2 (medium)	2 (probable)	2 (medium)	P8	L2
FI041-C	2 (medium)	2 (probable)	2 (medium)	P8	L2
FIO42-C	2 (medium)	2 (probable)	2 (medium)	P8	L2
FIO43-C	3 (high)	3 (likely)	2 (low)	P18	L1

Risk Assessment

FIO30-C. Exclude user input from format strings

Failing to exclude user input from format specifiers may allow an attacker to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO30-C	**3** (high)	**3** (probable)	**3** (low)	**P27**	**L1**

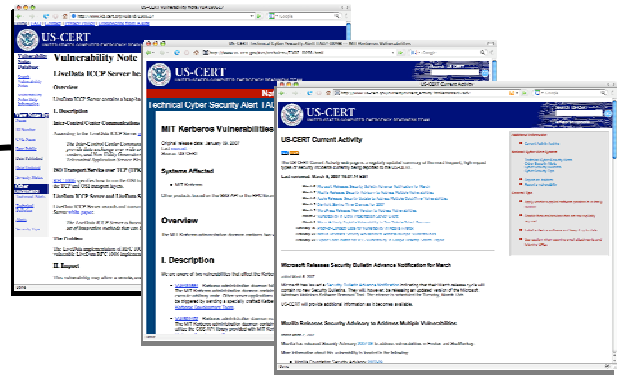
Two recent examples of format string vulnerabilities resulting from a violation of this rule include Ettercap and Samba. In Ettercap v.NG-0.7.2, the ncurses user interface suffers from a format string defect. The `curses_msg()` function in `ec_curses.c` calls `wdg_scroll_print()`, which takes a format string and its parameters and passes it to `vw_printw()`. The `curses_msg()` function uses one of its parameters as the format string. This input can include user-data, allowing for a format string vulnerability [VU#286468]. The Samba AFS ACL mapping VFS plug-in fails to properly sanitize user-controlled filenames that are used in a format specifier supplied to `snprintf()`. This security flaw becomes exploitable when a user is able to write to a share that uses Samba's `afsacl.so` library for setting Windows NT access control lists on files residing on an AFS file system.

Examples of vulnerabilities resulting from the violation of this rule can be found on the CERT website.

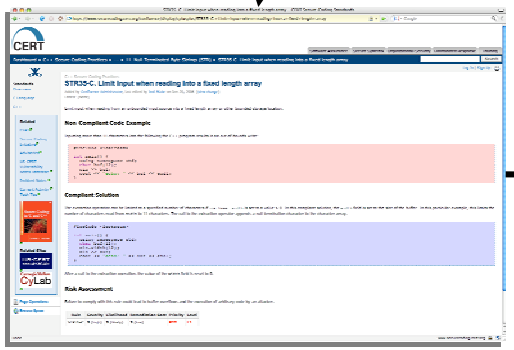
Relating Vulnerability Notes to Secure Coding Rules

Vulnerability Note VU#649732

This vulnerability occurred as a result of failing to comply with rule FIO30-C of the CERT C Programming Language Secure Coding Standard.



US CERT Technical Alerts



CERT Secure Coding Standard

Examples of vulnerabilities resulting from the violation of this recommendation can be found on the CERT website .

Applications

Establish secure coding practices within an organization

- may be extended with organization-specific rules
- cannot replace or remove existing rules

Train software professionals

Certify programmers in secure coding

Establish base-line requirements for software analysis tools

Certify software systems' compliance with secure coding rules

Agenda

Why Care about Software Security?

CERT Secure Coding Standards

SC22 OWGV

International Standards Project

Project title: Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

Initiated in September 2005

Assigned to a cross-cutting group, OWGV (ISO jargon for Other Working Group on Vulnerability)

- Intended to work collaboratively with language-specific working groups of SC 22
- May provide recommendations to language-specific working groups for changing language specifications

Product will be an ISO Technical Report – not a standard

Publication of report is planned for January 2009.

Participating National Bodies

Canada	SCC	Steve Michell
France	AFNOR	Franco Gasperoni
Germany (observing)	DIN	Roman Grahle
Italy	UNI	Tullio Vardanega
Japan	JSA	Kiyoshi Ishihata
UK	BSI IST-5	Derek Jones
USA	INCITS CT22	Rex Jaeschke

Other Participants

Ben Brosgol	RT/SC Java
Hal Burch	CERT
Paul Caseley	UK MOD
Rod Chapman	SPARK
Cesar Gonzalez-Perez	JTC 1/SC 7/WG 19
Barry Hedquist	
Chris Hills	MISRA C
Fred Long	CERT
Bob Martin	CVE, CWE
Ed de Moel	MDC (MUMPS)
Olwen Morgan	
Dan Nagle	WG5 (Fortran), J3 (Fortran)
Erhard Ploedereder	WG9 (Ada), Ada-Europe
Tom Plum	WG14 (C), ECMA TC39 / TG2 (C#)
Clive Pygott	MISRA C++
Robert Seacord	CERT
Bill Spees	FDA
Barry Tauber	WG4 (Cobol), J4 (Cobol)
Tucker Taft	SIGAda
Larry Wagoner	
Brian Wichmann	

Four Audiences

Safety: Products where it is critical to prevent behavior which might lead to human injury, and it is justified to spend additional development money

Security: Products where it is critical to secure data or access, and it is justified to spend additional development money

Predictability: Products where high confidence in the result of the computation is desired

Assurance: Products to be developed for dependability or other important characteristics

OWG: Vulnerability Status 1

The project has two officers

- Convener, John Benito
- Secretary, Jim Moore
- Still need an Editor

A skeleton document has been completed.

A template for vulnerability descriptions has been completed.

An initial set of vulnerabilities has been proposed for treatment.

OWG: Vulnerability Status 2

The body of Technical Report describes vulnerabilities in a generic manner, including:

- Brief description of application vulnerability
- Cross-reference to enumerations, e.g. CWE
- Categorizations by selected characteristics
- Description of failure mechanism, i.e. how coding problem relates to application vulnerability
- Points at which the causal chain could be broken
- Assumed variations among languages
- Ways to avoid the vulnerability or mitigate its effects

Annexes provide language-specific treatments of each vulnerability.

OWG: Vulnerability Status 3

OWGV maintains a web site for its work:

http://aitc.aitcnet.org/isai/

Meeting schedule:

- OWGV #5 2007-07-18/20 SCC, Ottawa, Canada
- OWGV #6 2007-10-1/3 Kona, Hawaii, USA
- OWGV #7 2007-12 (during week of 10 - 14) SEI, Pittsburgh, PA, USA

OWG:Vulnerability Product

A type 3 Technical Report

- A document containing information of a different kind from that which is normally published as an International Standard

Scope:

- The TR describes a set of common mode failures that occur across a variety of languages.
- The document will not contain normative statements, but information and suggestions.

No single programming language or family of programming languages is to be singled out

- As many programming languages as possible should be involved
- Need not be just the languages defined by ISO Standards

Dual Approach to Identifying Vulnerabilities

Empirical approach: Observe the vulnerabilities that occur in the wild and describe them, e.g. buffer overrun, execution of unvalidated remote content

Analytical approach: Identify potential vulnerabilities through analysis of programming languages

The second approach may help us identify tomorrow's vulnerabilities.

Analytical Approach

Vulnerabilities occur when software behaves in a manner that was not predicted by a competent developer. Sources of such vulnerabilities include:

- Issues arising from lack of knowledge
  - Complex language features or interactions of features that may be misunderstood
  - Portions of the language left unspecified by the standard
  - Portions of the language that are implementation-defined
  - Portions of the language that are specified as undefined
- Issues arising from human cognitive limitations, i.e, exceeding the human ability to understand
- Issues arising from non-standard extensions of languages

Language-Independent Vulnerability Description Example 1

6.1 SM-004 Out of bounds array element access

6.1.1 Description of application vulnerability

Unpredictable behaviour can occur when accessing the elements of an array outside the bounds of the array.

6.1.2 Cross reference

CWE: 129

6.1.3 Categorization

Section 5.1.2

6.1.4 Mechanism of failure

Arrays are defined, perhaps statically, perhaps dynamically, to have given bounds. In order to access an element of the array, index values for one or more dimensions of the array must be computed. If the index values do not fall within the defined bounds of the array, then access might occur to the wrong element of the array, or access might occur to storage that is outside the array. A write to a location outside the array may change the value of other data variables or may even change program code.

6.1.5 Possible ways to avoid the failure

The vulnerability can be avoided by not using arrays, by using whole array operations, by checking and preventing access beyond the bounds of the array, or by catching erroneous accesses when they occur. The compiler might generate appropriate code, the run-time system might perform checking, or the programmer might explicitly code appropriate checks.

Language-Independent Vulnerability Description Example 2

6.1.6 Assumed variations among languages

This vulnerability description is intended to be applicable to languages with the following characteristics:

- The size and bounds of arrays and their extents might be statically determinable or dynamic. Some languages provide both capabilities.
- Language implementations might or might not statically detect out of bound access and generate a compile-time diagnostic.
  - At run-time the implementation might or might not detect the out of bounds access and provide a notification at run-time. The notification might be treatable by the program or it might not be.
  - Accesses might violate the bounds of the entire array or violate the bounds of a particular extent. It is possible that the former is checked and detected by the implementation while the latter is not.
  - The information needed to detect the violation might or might not be available depending on the context of use. (For example, passing an array to a subroutine via a pointer might deprive the subroutine of information regarding the size of the array.)
- Some languages provide for whole array operations that may obviate the need to access individual elements.
  - Some languages may automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds. (This may or may not match the programmer's intent.)

Language-Independent Vulnerability Description Example 3

6.1.7 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- If possible, utilize language features for whole array operations that obviate the need to access individual elements.
  - If possible, utilize language features for matching the range of the index variable to the dimension of the array.
  - If the compiler can verify correct usage, then no mitigation is required beyond performing the verification.
- If the run-time system can check the validity of the access, then appropriate action may depend upon the usage of the system (e.g. continuing degraded operation in a safety-critical system versus immediate termination of a secure system).
- Otherwise, it is the responsibility of the programmer:
  - to use index variables that can be shown to be constrained within the extent of the array;
  - to explicitly check the values of indexes to ensure that they fall within the bounds of the corresponding dimension of the array;
  - to use library routines that obviate the need to access individual elements; or
  - to provide some other means of assurance that arrays will not be accessed beyond their bounds. Those other means of assurance might include proofs of correctness, analysis with tools, verification techniques, etc.

Desired Outcomes

Provide guidance to users of programming languages that :

- Assists them in improving the predictability of the execution of their software even in the presence of an attacker
- Informs their selection of an appropriate programming language for their job

Provide feedback to language standardizers, resulting in the improvement of programming language standards.

For More Information

Visit web sites:

https://www.securecoding.cert.org/

http://aitc.aitcnet.org/isai/

Contact presenters:

Robert C. Seacord
rcs@cert.org

James Moore
moorej@mitre.org