

Hands-On Tutorial: Auditing Static Analysis Alerts Using a Lexicon & Rules

Lori Flynn, David Svoboda, William Snavelly

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Software Engineering Institute

Carnegie Mellon University

© 2017 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been
approved for public release and unlimited distribution.

Copyright 2017 Carnegie Mellon University. All Rights Reserved.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM17-0664



Virtual Machine Setup

1. Unpack tutorial bundle:

This should produce an **.ovf** file

2. Open VMWare Player and select **File->Open**

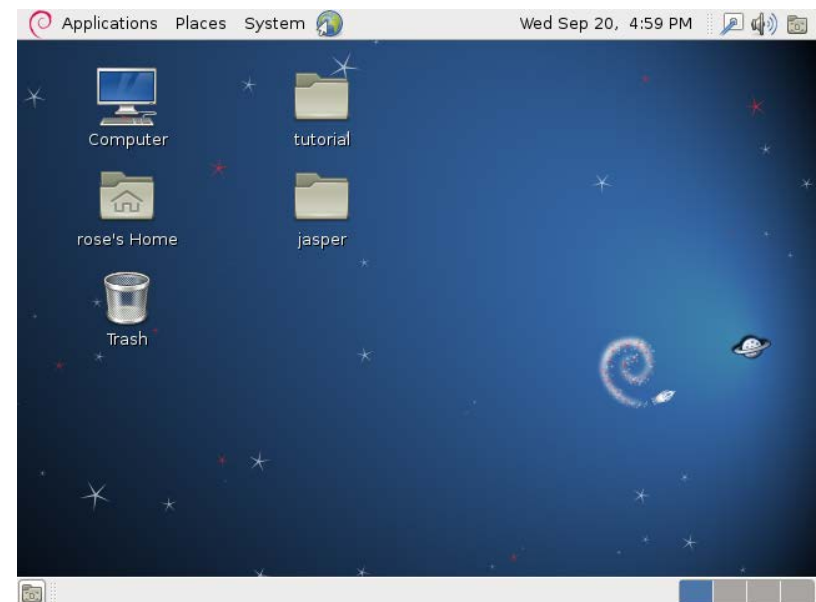
3. Select **.ovf** file

This will take about 5 minutes to extract VM.

4. Boot VM: **VM->Power->Start up Guest**

5. Dismiss any warning dialogs.

This will take 5 minutes to boot to a Linux desktop.



Virtual Machine and Printouts

VM contains:

- Source code (JasPer plus example programs we developed)
- Static analysis tools: rosecheckers, cppcheck, GCC
- Static analysis tool output

Printouts

- Source code for example programs
- Relevant parts of static analysis tool output

Auditing Lexicon And Rules

- This tutorial covers a set of **auditing rules** and part of the **lexicon** we've developed.
- Lexicon contains a set of **determinations** for static analysis alerts
- Includes a set of **auditing rules** to help auditors make consistent decisions in commonly-encountered situations

Different auditors should make the **same determination** for a given alert!

Improve the **quality and consistency** of audit data for the purpose of building **machine learning classifiers**

Help organizations make **better-informed** decisions about **bug-fixes, development, and future audits.**

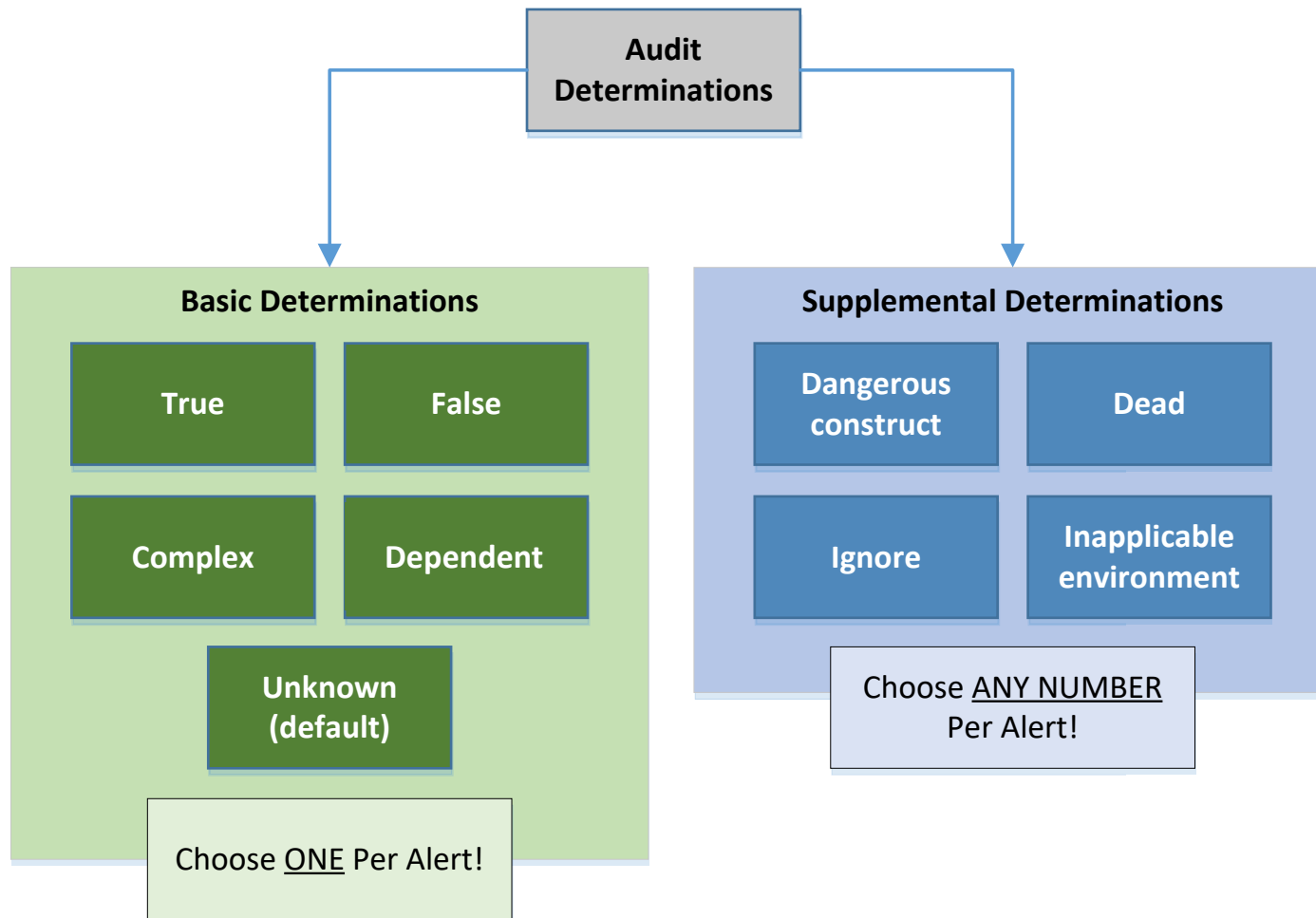
This Tutorial

- Teach basics of static analysis alert auditing
- Clarify **ambiguous or complex** auditing scenarios
- **Hands-on:** make audit determinations on static analysis alerts and open-source code
- Learn how to make audit determinations **more consistent**

Focus on **12 rules**

- Drew on our own experiences auditing code bases at CERT
- Trained groups of engineers on the rules, and incorporated their feedback
- Your feedback is welcome!

Lexicon: Audit Determinations



Lexicon: Basic Determinations

Unknown

- Nothing is known about the correctness of the alert.

True

- The code in question violates the **condition** indicated by the alert.

False

- The code in question does not violate the **condition** indicated by the alert.
- A **condition** is a constraint or property of validity.
 - E.g. A valid program should not dereference NULL pointers.
- The condition can be determined from the definition of the alert itself, or from the **coding taxonomy** the alert corresponds to.
 - CERT Secure Coding Rules
 - CWEs

Audit Rules



1. Assume external inputs to the program are malicious.
2. Some alerts are too difficult to judge; they should be marked *Complex*.
3. If an alert is true only when a previous alert is true, mark it *Dependent*.
4. Handle an alert in unreachable code depending on whether it is exportable.
5. An alert might indicate a true violation of the condition it is mapped to, even if the alert's message is useless or incorrect.
6. Mark an alert *True* even if code maintainers will protest.
7. Unless instructed otherwise, assume code must be portable.
8. When auditing an alert, if a second true violation is discovered, its alert should be marked *True*.
9. Auditors must understand the language and the current alert's condition.
10. Do not arbitrarily extend the scope of a condition.
11. Code that behaves as expected might still violate a condition.
12. Multiple messages help in understanding an alert.

JasPer

The [JasPer Project](#) is an open-source initiative to provide a free software-based reference implementation of the codec specified in the JPEG-2000 Part-1 standard (i.e., ISO/IEC 15444-1).

Some of our code examples come from this library.

It is designed to run on 32-bit Intel platforms.

This means we can rely on the sizes of various integer types.

Audit Rules

1. Assume external inputs to the program are malicious.

1. Assume external inputs to the program are malicious.

External inputs may be:

- Arguments to a library function
- Program arguments (**argv** in C)
- Environment variables
- External files (that aren't bundled with the program)
- Data over the network
- User data

VM Exercise: Make an Alert Determination

- Examine the diagnostic alert info at line **23** of the file **tutorial/analysis/rosecheckers.txt**
- Examine the source code at line **24** of the file **tutorial/src/file4.c**
- Examine CERT rule [INT33-C](#) in the SEI Secure Coding Rules PDF, on page **163**.
- Study for 2 minutes, then we'll discuss.

Basic

Unknown

Complex

True

False

Dependent

Supplemental

☐ Dangerous

☐ Ignored

☐ Dead

☐ Inapplicable

Example

The screenshot shows the SCALE Analysis Tool interface. At the top, the browser address bar displays `https://scale.cert.org/projects/76`. The page header includes "SCALE Analysis Tool", "Help", "SCALE at CERT", and "Copyright (c) 2007-2015 Carnegie Mellon University".

Below the header, there is a control bar with "Set all selected to:" followed by two dropdown menus (both set to "--") and an "Update" button.

The main content area features a table with the following columns: ☐, ID, Flag, Verdict, Previous, Path, Line, Message, Checker, Tool, and Rule.

☐	ID	Flag	Verdict	Previous	Path	Line	Message	Checker	Tool	Rule
☐	1412	[]	[Unknown]	Unknown	file4.c	24	Ensure that division and modulo operations do not result in divide-by-zero errors	INT33-C	rosecheckers	INT33-C

Below the table, a code editor displays the source code for `file4.c`. The code is as follows:

```
10  if(list == NULL) {
11      return 0;
12  }
13
14  while(*list != 0) {
15      if (INT_MAX - sum < *list) {
16          printf("Sum too large!\n");
17          exit(1);
18      }
19      sum += *list;
20      list++;
21      count++;
22  }
23
24  return sum / count;
25 }
```

The line `return sum / count;` (line 24) is highlighted in yellow, corresponding to the alert in the table above. At the bottom of the code editor, there are navigation icons and the filename `file4.c`.

CERT Coding Rule INT33-C

The screenshot shows a web browser window with the address bar displaying the URL: <https://www.securecoding.cert.org/confluence/display/c/INT33-C.+Ensure+that+divi...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Spaces", "People", and a "Create" button. The main content area is titled "INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors". It is created by the Confluence Administrator and last modified by Will Snively on Feb 09, 2016. The text states: "The C Standard identifies the following condition under which division and remainder operations result in **undefined behavior (UB)**:" followed by a table.

UB	Description
45	The value of the second operand of the / or % operator is zero (6.5.5).

Below the table, the text reads: "Ensure that division and remainder operations do not result in divide-by-zero errors."

Simple Code Example

```
int main(int argc, char **argv) {  
    int list[10];  
    int result;
```

```
    for (int i = 0; i < 10; i++) {  
        if (i+1 >= argc) {  
            break;  
        }  
        read_integer(argv[i+1], &list[i]);  
        if (list[i] <= 0) {  
            break;  
        }  
    }  
}
```

`read_integer()` fills 2nd
arg with integer value of 1st arg.

```
    result = compute_int_average(list);  
    printf("Average is: %d\n", result);  
    return 0;
```

```
}
```

The compute_int_average() function

```
int main(int argc, char **argv) {
    int list[10];
    int result;

    for (int i = 0; i < 10; i++) {
        if (i+1 >= argc)
            break;
        read_integers(list[i]);
        if (list[i] == 0)
            break;
    }

    result = compute_int_average(list);
    printf("Average: %d\n", result);
    return 0;
}

int compute_int_average(int* list) {
    int sum = 0;
    int count = 0;

    if (list == NULL) {
        return 0;
    }
    while (*list != 0) {
        if (INT_MAX - sum > *list) {
            printf("Sum too large!\n");
            exit(1);
        }
        sum += *list;
        list++;
        count++;
    }
    return sum / count;
}
```

Alert to Audit

```
int main(int argc, char **argv) {
    int list[10];
    int result;

    for (int i = 0; i < 10; i++) {
        if (i+1 >= argc)
            break;
        read_int(&list[i]);
    }

    int compute_int_average(int* list) {
        int sum = 0;
        int count = 0;

        if (list == 0)
            return 0;

        while (*list != 0) {
            sum += *list;
            count++;
            list++;
        }

        return sum / count;
    }

    result = compute_int_average(list);
    printf("Average: %d\n", result);
    return 0;
}
```

What kind of input could cause this **while** loop to never execute?

Line	Message	Checker	Tool	Rule
24	Ensure that division and modulo operations do not result in divide-by-zero errors	INT33-C	rosecheckers	INT33-C

→ **return sum / count;**

Marking the alert

```

int main(int argc, char **argv) {
    int list[10];
    int result;

    for (int i = 0; i < 10; i++) {
        if (i+1 >= argc)
            break;
        read_int(argv[i], &list[i]);
    }

    if (compute_int_average(list) != 0) {
        printf("Average is %d\n", result);
    }
}

int compute_int_average(int* list) {
    int sum = 0;
    int count = 0;

    if (list == NULL) {
        return 0;
    }

    while (*list != 0) {
        sum += *list;
        count++;
        list++;
    }

    return sum / count;
}

```

Line	Message	Checker	Tool	Rule
24	Ensure that division operations do not result in zero errors	INT33-C	rosecheckers	INT33-C

Unknown Complex

True ←

False

Dependent

→ **return sum / count;**

Audit Rules

2. Some alerts are too difficult to judge; they should be marked *Complex*.

2. Some alerts are too complex to judge; they should be marked *Complex*.

Auditors typically have many alerts to examine.

So don't let a difficult alert slow you down.

Complex: The validity of the alert cannot be assessed given a reasonable amount of auditor effort.

Many of the other audit rules exist to reduce the number of alerts that must be marked *Complex*.

Example

The screenshot shows the SCALe Analysis Tool interface. The browser address bar displays <https://scale.cert.org/projects/74>. The page title is "SCALe Analysis Tool" with a "Help" link and "SCALe at CERT" logo. The copyright notice is "Copyright (c) 2007-2015 Carnegie Mellon University".

At the top, there is a "Page: 10" dropdown, a "Go" button, and a "New Diagnostic" button. Below this is a "--" dropdown and an "Update" button.

The main content area displays a table of diagnostic messages:

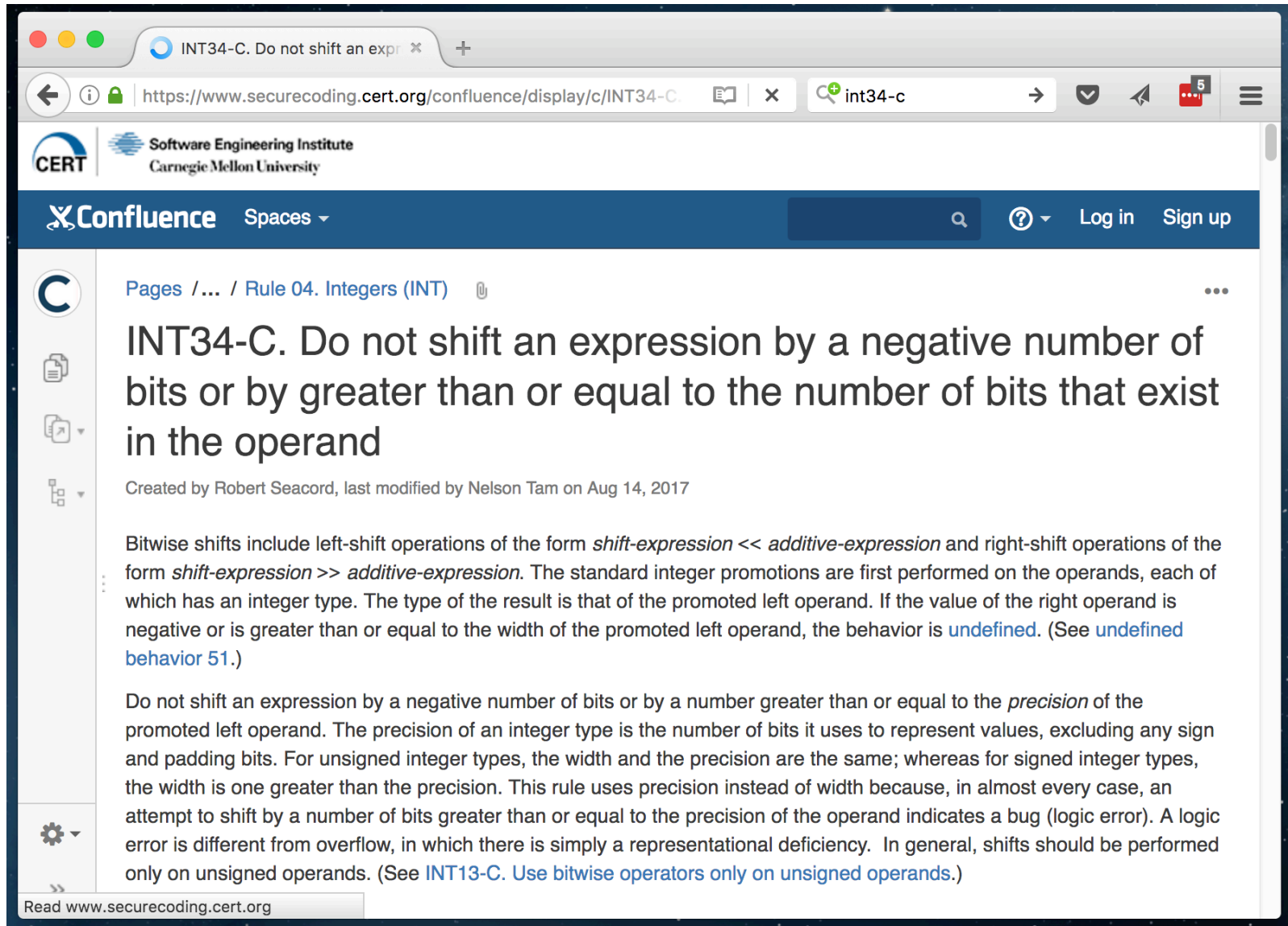
	Previous	Path	Line	Message	Checker	Tool	Rule	Title
	Unknown	/jasper-1.900.1/src/libjasper/base/jas_cm.c	1237	Do not shift a negative number of bits or more bits than exist in the operand	INT34-C	rosecheckers	INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

Below the table, the source code is displayed with line numbers 1230 to 1239. Line 1237 is highlighted in yellow:

```
1230     int m;  
1231     v = **bufptr;  
1232     if (sgnd) {  
1233         m = (1 << (prec - 1));  
1234         if (v < -m || v >= m)  
1235             return -1;  
1236     } else {  
1237         if (v < 0 || v >= (1 << prec))  
1238             return -1;  
1239     }
```

At the bottom, there are navigation icons and a file path: [jasper-1.900.1/src/libjasper/base/jas_cm.c](#).

CERT Coding Rule INT34-C



The screenshot shows a web browser displaying the CERT Coding Rule INT34-C. The browser's address bar shows the URL <https://www.securecoding.cert.org/confluence/display/c/INT34-C>. The page header includes the CERT logo and the Software Engineering Institute Carnegie Mellon University name. The Confluence navigation bar shows the rule title "INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand". The main content area contains the following text:

Created by Robert Seacord, last modified by Nelson Tam on Aug 14, 2017

Bitwise shifts include left-shift operations of the form *shift-expression* << *additive-expression* and right-shift operations of the form *shift-expression* >> *additive-expression*. The standard integer promotions are first performed on the operands, each of which has an integer type. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is [undefined](#). (See [undefined behavior 51](#).)

Do not shift an expression by a negative number of bits or by a number greater than or equal to the *precision* of the promoted left operand. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. For unsigned integer types, the width and the precision are the same; whereas for signed integer types, the width is one greater than the precision. This rule uses precision instead of width because, in almost every case, an attempt to shift by a number of bits greater than or equal to the precision of the operand indicates a bug (logic error). A logic error is different from overflow, in which there is simply a representational deficiency. In general, shifts should be performed only on unsigned operands. (See [INT13-C. Use bitwise operators only on unsigned operands](#).)

Read www.securecoding.cert.org

Example

```
static int jas_cmgetint(long **bufptr, int sgnd,
                      int prec, long *val)
{
    long v;
    int m;
    v = **bufptr;
    if (sgnd) {
        m = (1 << (prec - 1));
        if (v < -m || v >= m)
            return -1;
    } else {
        if (v < 0 || v >= (1 << prec))
            return -1;
    }
    ...
}
```

Alert to Audit

```
static int jas_cmgetint(long **bufptr, int sgnd,
                      int prec, long *val)
{
    long v;
    int m;
    v = **bufptr;
    if (sgnd) {
        m = (1 << (prec - 1));
        if (v < -m || v >= m)
            return -1;
    } else {
        → if (v < 0 || v >= (1 << prec))
```

How big is **prec**?

Path	Line	Message	Checker	Tool	Rule	Title
/jasper-1.900.1/src/libjasper/base/jas_cm.c	1237	Do not shift a negative number of bits or more bits than exist in the operand	INT34-C	rosecheckers	INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

Getting the size of `prec`

Determining how big `prec` is a complicated and time-consuming process, tracing back several function calls and data structures.

Consequently, an auditor can mark this alert *Complex*.

Marking the Alert

```
static int jas_cmgetint(long **bufptr, int sgnd,
                      int prec, long *val)
{
    long v;
    int m;
    v = **bufptr;
    if (sgnd) {
        m = (1 << (prec - 1));
        if (v < -m || v >= m)
            return -1;
    } else {
```

How big is **prec**?

(v < 0 || v >= (1 << prec))

Unknown

Complex

True

False

Dependent

Path	Line	Message	Checker	Tool	Rule	Title
/jasper-1.900.1/src/libjasper/base/jas_cm.c	1237	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand	INT34-C	rosecheckers	INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

Audit Rules

3. If an alert is true only when a previous alert is true, mark it *Dependent*.

3. If an alert is true only when a previous alert is true, mark it *Dependent*.

When code violates a secure coding condition, subsequent program behavior is hard to predict.

An auditor should begin with the assumption that the program is well-behaved before reaching the line in question.

- It does not matter whether unexpected behavior is possible, just that it does not happen.

Assuming that a program complies with secure coding conditions helps to minimize the difficulty of auditing code.

Earlier Violations

If an auditor is convinced that an alert might be true only if violation of a secure coding condition occurs earlier in the code:

- Mark the alert as *Dependent*.
- See Rule #8 for how to handle the earlier violation.
 - 8. When auditing an alert, if a second true violation is discovered, its alert should be marked *True*.

Dependent: The alert can only be valid if a secure coding condition was violated earlier in the code.

Weakness to Violations

An auditor might not be familiar with every secure coding condition, but if the code demonstrates undefined behavior or some other unexpected behavior, it is likely to violate a secure coding condition.

VM Exercise: Make an Alert Determination

- Examine the diagnostic alert info at lines **28** and **30** of the file **tutorial/analysis/rosecheckers.txt**
- Examine the source code at lines **24** and **27** of the file **tutorial/src/file5.c**
- Examine CERT rule [INT33-C](#) in the SEI Secure Coding Rules PDF, on page **163**.
- Study for 2 minutes, then we'll discuss.

Basic

Unknown

Complex

True

False

Dependent

Supplemental

☐ Dangerous

☐ Ignored

☐ Dead

☐ Inapplicable

Example

The screenshot displays the SCALe Analysis Tool interface. At the top, the browser address bar shows the URL `https://scale.cert.org/projects/77`. Below the browser, the tool's header includes "SCALe Analysis Tool", "Help", and "SCALe at CERT". A copyright notice for Carnegie Mellon University (2007-2015) is also present.

The main content area features a table with analysis results. The table has columns for various attributes, including line numbers, file names, and descriptions of the errors. Two entries are visible:

Line	File	Severity	Category	Rule	Checkers	Link	Description
1712	file5.c	24	[Unknown]	Unknown	INT33-C	rosecheckers	Ensure that division and modulo operations do not result in divide-by-zero errors
1912	file5.c	27	[Unknown]	Unknown	INT33-C	rosecheckers	Ensure that division and modulo operations do not result in divide-by-zero errors

Below the table, a code editor displays the source code for `file5.c`. The code is as follows:

```
17     exit(1);
18 }
19 sum += *list;
20 list++;
21 count++;
22 }
23
24 if (sum / count < 10) {
25     puts("Average is too low");
26 }
27 return sum / count;
28 }
```

The code editor includes navigation icons (back, forward, search, etc.) and the filename `file5.c` is displayed at the bottom.

CERT Coding Rule INT33-C

The screenshot shows a web browser window displaying the CERT Coding Rule INT33-C on the Confluence website. The browser's address bar shows the URL: <https://www.securecoding.cert.org/confluence/display/c/INT33-C.+Ensure+that+divi...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar is visible, showing "Spaces", "People", and a "Create" button. The main content area is titled "INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors". Below the title, it states "Created by Confluence Administrator, last modified by Will Snively on Feb 09, 2016". The text describes the condition under which division and remainder operations result in undefined behavior (UB). A table lists the UB conditions:

UB	Description
45	The value of the second operand of the / or % operator is zero (6.5.5).

Below the table, the text states: "Ensure that division and remainder operations do not result in divide-by-zero errors."

Example

```
int main(int argc, char **argv) {  
    int list[10];  
    int result;
```

```
    for (int i = 0; i < 10; i++) {  
        if (i+1 >= argc) {  
            break;  
        }  
        read_integer(argv[i+1], &list[i]);  
        if (list[i] <= 0) {  
            break;  
        }  
    }  
}
```

`read_integer()` fills 2nd arg with integer value of 1st arg.

Code unchanged from previous example.

```
    result = compute_int_average(list);  
    printf("Average is: %d\n", result);  
    return 0;
```

```
}
```

The compute_int_average() function

```
int main(int argc, char **argv) {
    int list[10];
    int result;

    for (int i = 0; i < 10; i++) {
        if (i+1 < argc) {
            read_int(&list[i]);
            if (list[i] < 0) {
                break;
            }
        }
    }

    result = compute_int_average(list);
    printf("Average is: %d\n", result);
    return 0;
}
```

```
int compute_int_average(int* list) {
    ...
    if (sum / count < 10) {
        puts("Average is too low");
    }
    return sum / count;
}
```

New lines, rest of function unchanged

Alerts to Audit

```

int main(int argc, char **argv) {
    int list[10];
    int result;
    for (int i = 0; i < 10; i++) {
        if (i+1 == 0) {
            break;
        }
        read_int(&list[i]);
        if (list[i] == 0) {
            break;
        }
    }
}

int compute_int_average(int* list) {
    ...
    if (sum / count < 10) {
        puts("Average is too low");
    }
    return sum / count;
}

```

file5.c	24	Ensure that division and modulo operations do not result in divide-by-zero errors	INT33-C	rosecheckers	INT33-C
file5.c	27	Ensure that division and modulo operations do not result in divide-by-zero errors	INT33-C	rosecheckers	INT33-C

Review

We know that count could be 0 during the first division, on line 24.

- This violates INT33-C.

The same division occurs three lines later, on line 27.

Why does this violate INT33-C?

Why doesn't this violate INT33-C?

Resolution

The first alert is clearly *true*.

The last alert could not be *true* unless the first alert is also *true*.

Therefore the last alert should be marked *Dependent*.

Marking the Alerts

```

int main(int argc, char **argv) {
    int list[10];
    int result;
    int compute_int_average(int* list) {
        for (int i = 0; i < 10; i++) {
            if (i+1 == 10)
                break;
            ...
            if (sum / count < 10) {
                puts("Average is too low");
            }
            result = sum / count;
        }
    }
}

```

file5.c	24	Ensuring operation is zero	Unknown	INT33-C	rosecheckers	INT33-C
file5.c	27	Ensuring operation is zero	Complex	INT33-C	rosecheckers	INT33-C
			True			
			False			
			Dependent			

Audit Rules

4. Handle an alert in unreachable code depending on whether it is exportable.

4. Handle an alert in unreachable code depending on whether it is exportable.

Certain code segments may be **unreachable** at runtime. Also called **dead code**.

A static analysis tool might not be able to realize this, and **still mark alerts** in code that **cannot be executed**.

The **Ignored** verdict can be applied to these alerts.

However, an auditor should **be careful** when deciding if a piece of code is truly dead.

In particular: just because a given program module (function, class) is not used does **not** mean it is dead. The module might be exported as a **public interface**, for use by another application.

This rule was developed as a result of a scenario encountered by one of our collaborators.

VM Exercise: Make an Alert Determination

- Examine the diagnostic alert info at line **40** of the file **tutorial/analysis/rosecheckers.txt**
- Examine the source code at line **24** of the file **tutorial/src/file6.c**
- Examine CERT rule [INT33-C](#) in the SEI Secure Coding Rules PDF, on page **163**.
- Study for 2 minutes, then we'll discuss.

Basic

Unknown

Complex

True

False

Dependent

Supplemental

☐ Dangerous

☐ Ignored

☐ Dead

☐ Inapplicable

Example

The screenshot shows the SCALE Analysis Tool interface. At the top, the browser address bar displays `https://scale.cert.org/projects/77`. Below the browser, the tool's header includes "SCALE Analysis Tool", "Help", and "SCALE at CERT". A copyright notice "Copyright (c) 2007-2015 Carnegie Mellon University" is on the right. A table lists project entries, with the selected entry having ID 2212, name "file6.c", and a description "Ensure that division and modulo operations do not result in divide-by-zero errors". The code editor shows a C function with a while loop and a return statement. The return statement is highlighted in yellow. The file name "file6.c" is shown at the bottom of the editor.

☐	2212	[]	[Unknown]	Unknown	file6.c	24	Ensure that division and modulo operations do not result in divide-by-zero errors	INT33-C	rosecheckers	INT33-C	Ens...
--------------------------	------	-----	-----------	---------	---------	--------------------	---	---------	--------------	-------------------------	--------

```
12     }
13
14     while(*list != 0) {
15         if (INT_MAX - sum < *list) {
16             printf("Sum too large!\n");
17             exit(1);
18         }
19         sum += *list;
20         list++;
21         count++;
22     }
23
24     return sum / count;
25 }
26
27 void read_integer(char *s, int *val) {
```

file6.c

CERT Coding Rule INT33-C

The screenshot shows a web browser window displaying the CERT Coding Rule INT33-C on the Confluence website. The browser's address bar shows the URL: <https://www.securecoding.cert.org/confluence/display/c/INT33-C.+Ensure+that+divi...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Spaces", "People", and a "Create" button. The page title is "INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors". Below the title, it says "Created by Confluence Administrator, last modified by Will Snively on Feb 09, 2016". The main content area states: "The C Standard identifies the following condition under which division and remainder operations result in **undefined behavior (UB)**:" followed by a table. The table has two columns: "UB" and "Description". The first row shows "45" in the "UB" column and "The value of the second operand of the / or % operator is zero (6.5.5)." in the "Description" column. Below the table, it says "Ensure that division and remainder operations do not result in divide-by-zero errors." The browser window also shows a user profile "David" in the top right corner.

INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors

Created by Confluence Administrator, last modified by Will Snively on Feb 09, 2016

The C Standard identifies the following condition under which division and remainder operations result in **undefined behavior (UB)**:

UB	Description
45	The value of the second operand of the / or % operator is zero (6.5.5).

Ensure that division and remainder operations do not result in divide-by-zero errors.

Example

```
int main(int argc, char **argv) {  
    int list[10];  
    int result;
```

```
    for (int i = 0; i < 10; i++) {  
        if (i+1 >= argc) {  
            break;  
        }  
        read_integer(argv[i+1], &list[i]);  
        if (list[i] <= 0) {  
            break;  
        }  
    }  
}
```

read_integer() fills 2nd
arg with integer value of 1st arg.

These lines commented out.
No other changes.

```
// result = compute_int_average(list);  
// printf("Result is: %d\n", result);  
return 0;
```

```
}
```

The compute_int_average() function

```
int main(int argc, char **argv) {
    int list[10];
    int result;

    for (int i = 0; i < 10; i++) {
        if (i+1 >= 10)
            break;
        read_integers(list[i]);
        if (list[i] >= 0)
            break;
    }

    // result = compute_int_average(list);
    // printf("Result: %d\n", result);
    return 0;
}

int compute_int_average(int* list) {
    int sum = 0;
    int count = 0;

    if (list == NULL) {
        return 0;
    }
    while (*list != 0) {
        if (INT_MAX - sum > *list) {
            printf("Sum too large!\n");
            exit(1);
        }
        sum += *list;
        list++;
        count++;
    }
    return sum / count;
}
```

No changes to this function

Alert to Audit

```

int main(int argc, char **argv) {
    int list[10];
    int result;

    for (int i = 0; i < argc; i++) {
        if (i+1 >= argc)
            break;
        read_int(&list[i]);
    }

    int compute_int_average(int* list) {
        int sum = 0;
        int count = 0;

        if (list == NULL) {
            return 0;
        }
        while (*list != 0) {
            sum += *list;
            count++;
        }
        return sum / count;
    }

    // result = compute_int_average(list);
    // printf("Result: %d\n", result);
    return 0;
}

```

Line	Message	Checker	Tool	Rule
24	Ensure that division and modulo operations do not result in divide-by-zero errors	INT33-C	rosecheckers	INT33-C

→ **return sum / count;**

Recap

Once again, we have what looks like a genuine division by zero.

But, the **compute_int_average()** function is never actually called. This is dead code.

Marking the Alert

```
int main(int argc, char **argv) {
    int list[10];
    int result;

    for (int i = 0; i < 10; i++) {
        if (i+1 >= 10)
            break;
        // read integer
        int compute_int_average(int* list) {
            int sum = 0;
            int count = 0;

            if (list == NULL) {
                return 0;
            }
            while (*list != 0) {
                // read integer
                int value;
                if (scanf("%d", &value) != 1) {
                    return 0;
                }
                sum += value;
                count++;
            }
            return sum / count;
        }
    }
}
```

Line	Message	Complexity	Category	Rule
24	Ensure that operations do not result in zero errors	Unknown	Dead	INT33-C

Arrows indicate the mapping from the alert message to the 'Unknown' complexity level and the 'Dead' category.

Audit Rules

5. An alert might indicate a true violation of the condition it is mapped to, even if the alert's message is useless or incorrect.

5. An alert might indicate a true violation of the condition it is mapped to, even if the alert's message is useless or incorrect.

An alert's message may be correct, yet the alert may be *false*.

Conversely, the message may be incorrect, yet the alert may be *true*.

The alert message should be taken as a hint, but the auditor is free to ignore the message when considering an alert.

Audit Rules

**6. Mark an alert *True* even if
code maintainers will protest.**

6. Mark an alert *True* even if code maintainers will protest.

Tension between code maintainers and auditors:

- Maintainers wish to avoid superfluous changes.
- Auditors wish for code security.

Not marking an alert *true* may constitute a *false negative*: a security vulnerability that is never recognized or fixed.

Example

Suppose a C program accesses memory 0 (**null**) because the program runs only on a hardware platform that allows this.

Diagnosing a flaw in the code may result in:

- Changing the code (a fix)
- Better documentation (better understanding of the platform)
- Discussion of what is secure, and allowed.

Audit Rules

**7. Unless instructed
otherwise, assume code must
be portable.**

7. Unless instructed otherwise, assume code must be portable.

Ideally you will have one of two portability specifications:

- Is this code secure for **one** particular platform (e.g., 32-bit x86)
- Is this code secure for all platforms (e.g., strictly conformant to the ISO C standard)?

If a line of code malfunctions on certain platforms, and in doing so violates a secure coding rule, this is suitable justification for marking the alert *true*.

Example

One recently-audited C codebase has a pointer that iterates from memory location 0x0 to another hardcoded memory address.

This violates several CERT C rules:

- [ARR36-C. Do not subtract or compare two pointers that do not refer to the same array](#)
- [EXP34-C. Do not dereference null pointers](#)

This is embedded code, and it is believed that its platform specifically permits this behavior, as the code would not run otherwise.

So these alerts are *true* if and only if the code is portable.

CERT Coding Rule ARR36-C

The screenshot shows a web browser window displaying the CERT Coding Rule ARR36-C on the Confluence website. The browser's address bar shows the URL <https://www.securecoding.cert.org/confluence/display/c/ARR36-C...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Spaces", "People", and a "Create" button. The page title is "ARR36-C. Do not subtract or compare two pointers that do not refer to the same array". The content area contains the following text:

When two pointers are subtracted, both must point to elements of the same array object or just one past the last element of the array object (C Standard, 6.5.6 [ISO/IEC 9899:2011]); the result is the difference of the subscripts of the two array elements. Otherwise, the operation is [undefined behavior](#) (see [undefined behavior 48](#)).

Similarly, comparing pointers using the relational operators `<`, `<=`, `>=`, and `>` gives the positions of the pointers relative to each other. Subtracting or comparing pointers that do not refer to the same array is undefined behavior (see [undefined behavior 48](#) and [undefined behavior 53](#)).

Comparing pointers using the equality operators `==` and `!=` has well-defined semantics regardless of whether or not either of the pointers is null, points into the same object, or points one past the last element of an array object or function.

CERT Coding Rule EXP34-C

The screenshot shows a web browser window with the address bar displaying <https://www.securecoding.cert.org/confluence/d...>. The browser's address bar also shows the page title "EXP34-C. Do not dereferen" and a user profile "David". The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Confluence", "More", "Create", and a search bar. The main content area displays the page title "EXP34-C. Do not dereference null pointers" and the text "Created by Jeff Gennari, last modified by Will Snively on Feb 10, 2016". The page content includes the text "Dereferencing a null pointer is **undefined behavior**." and "On many platforms, dereferencing a null pointer results in **abnormal program termination**, but this is not required by the standard. See "Clever Attack Exploits Fully-Patched Linux Kernel" [Goodin 2009] for an example of a code execution exploit that resulted from a null pointer dereference." The page also shows a "Noncompliant Code Example" section.

VM Exercise: Make an Alert Determination

- Examine the diagnostic alert info at line **20** of the file **tutorial/analysis/rosecheckers.txt**
- Examine the source code at line **17** of the file **tutorial/src/file3.c**
- Examine CERT rule [EXP36-C](#) in the SEI Secure Coding Rules PDF, on page **99**.
- Study for 2 minutes, then we'll discuss.

Basic

Unknown

Complex

True

False

Dependent

Supplemental

☐ Dangerous

☐ Ignored

☐ Dead

☐ Inapplicable

Example

The screenshot shows the SCALE Analysis Tool interface. The browser address bar displays `https://scale.cert.org/projects/76`. The page header includes "SCALE Analysis Tool", "Help", "SCALE at CERT", and "Copyright (c) 2007-2015 Carnegie Mellon University".

Below the header, it says "Showing 1 to 2 of 2 | Diagnostics per page: 10" with a "Go" button and a "New Diagnostic" button. There is also a "Set all selected to:" dropdown menu with an "Update" button.

☐	ID	Flag	Verdict	Previous	Path	Line	Message	Checker	Tool	Rule
☐	612	[]	[Unknown]	Unknown	file2.c	9	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C
☐	1312	[]	[Unknown]	Unknown	file3.c	17	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C

Below the table, a code snippet is displayed with line numbers 11 to 20. Line 17 is highlighted in yellow:

```
11 int main(int argc, char **argv) {  
    /*  [+11 file3.c] */  
12     struct FourByteValue val;  
13     val.b1 = 0x25;  
14     val.b2 = 0x4b;  
15     val.b3 = 0xd5;  
16     val.b4 = 0xa4;  
17     int *int_ptr = (int*)&val.b1;  
18     printf("%d\n", *int_ptr);  
19     return 0;  
20 }
```

At the bottom of the code view, there are navigation icons and the filename "file3.c".

CERT Coding Rule EXP36-C

The screenshot shows a web browser window with the address bar displaying <https://www.securecoding.cert.org/confluence/display/c/EXP36-C.+Do+not+cast+p...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Spaces", "People", and a "Create" button. The main content area is titled "EXP36-C. Do not cast pointers into more strictly aligned pointer types" and is part of the "Rule 03. Expressions (EXP)" section. The page was created by cjohns and last modified by Barbara White on Feb 02, 2016. The text explains that converting a pointer value to a more strictly aligned type is undefined behavior, citing the C Standard (6.3.2.3, paragraph 7 [ISO/IEC 9899:2011]). It also mentions that dereferencing a misaligned pointer may cause a loss of information or terminate abnormally.

EXP36-C. Do not cast pointers into more strictly aligned pointer types

Created by cjohns, last modified by Barbara White on Feb 02, 2016

Do not convert a pointer value to a pointer type that is more strictly aligned than the referenced type. Different alignments are possible for different types of objects. If the type-checking system is overridden by an explicit cast or the pointer is converted to a void pointer (`void *`) and then to a different type, the alignment of an object may be changed.

The C Standard, 6.3.2.3, paragraph 7 [ISO/IEC 9899:2011], states:

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined.

See [undefined behavior 25](#).

If the misaligned pointer is dereferenced, the program may [terminate abnormally](#). On some architectures, the cast alone may cause a loss of information even if the value is not dereferenced if the types involved have differing alignment requirements.

Example

```
int main(int argc, char **argv) {  
    struct FourByteValue val;  
    val.b1 = 0x25;  
    val.b2 = 0x4b;  
    val.b3 = 0xd5;  
    val.b4 = 0xa4;  
    int *int_ptr = (int*)&val.b1;  
    printf("%d\n", *int_ptr);  
    return 0;  
}
```

```
struct FourByteValue {  
    char b1;  
    char b2;  
    char b3;  
    char b4;  
};
```

Alert to Audit

```
int main(int argc, char **argv) {  
    struct FourByteValue val;  
    val.b1 = 0x25;  
    val.b2 = 0x4b;  
    val.b3 = 0xd5;  
    val.b4 = 0xa4;  
    → int *int_ptr = (int*)&val.b1;  
    printf("%d\n", *int_ptr);  
    return 0;  
}
```

```
struct FourByteValue {  
    char b1;  
    char b2;  
    char b3;  
    char b4;  
};
```

This code works on Intel x86 architectures, but what about others?

Line	Message	Checker	Tool	Rule
17	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C

Marking the alert

```
int main(int argc, char **argv) {
    struct FourByteValue val;
    val.b1 = 0x25;
    val.b2 = 0x4b;
    val.b3 = 0xd5;
    val.b4 = 0xa4;
    → int *int ptr = (int*)(&val.b1);
    printf("%d\n", *int ptr);
    return 0;
}
```

```
struct FourByteValue {
    char b1;
    char b2;
    char b3;
    char b4;
};
```

Unknown
Complex
True
False
Dependent

Line	Message	Checker	Tool	Rule
17	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C

VM Exercise: Make an Alert Determination

- Examine the diagnostic alert info at lines **6** and **7** of the file **tutorial/analysis/gcc.txt**
- Examine the source code at lines **15** and **16** of the file **tutorial/src/file3.c**
- Examine CERT rule [INT31-C](#) in the SEI Secure Coding Rules PDF, on page **144**.
- Study for 2 minutes, then we'll discuss.

Basic

Unknown

Complex

True

False

Dependent

Supplemental

☐ Dangerous

☐ Ignored

☐ Dead

☐ Inapplicable

Example

```
int main(int argc, char **argv) {  
    struct FourByteValue val;  
    val.b1 = 0x25;  
    val.b2 = 0x4b;  
    val.b3 = 0xd5;  
    val.b4 = 0xa4;  
    int *int_ptr = (int*)&val.b1;  
    printf("%d\n", *int_ptr);  
    return 0;  
}
```

```
struct FourByteValue {  
    char b1;  
    char b2;  
    char b3;  
    char b4;  
};
```

Code unchanged

Alerts to Audit

```
int main(int argc, char **argv) {  
    struct FourByteValue val;  
    val.b1 = 0x25;  
    val.b2 = 0x4b;  
    → val.b3 = 0xd5;  
    → val.b4 = 0xa4;  
    int *int_ptr = (int*)&val.b1;  
    printf("%d\n", *int_ptr);  
    return 0;  
}
```

```
struct FourByteValue {  
    char b1;  
    char b2;  
    char b3;  
    char b4;  
};
```

What happens on platforms where plain **char** is signed?

Line	Message	Checker	Tool	Rule	Title
15	warning: overflow in implicit constant conversion [-Woverflow]	.*overflow in .* conversion	gcc	INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data
16	warning: overflow in implicit constant conversion [-Woverflow]	.*overflow in .* conversion	gcc	INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data

Marking the alerts

```
int main(int argc, char **argv) {
    struct FourByteValue val;
    val.b1 = 0x25;
    val.b2 = 0x4b;
    → val.b3 = 0xd5;
    → val.b4 = 0xa4;
    int *int_ptr = (int*)&val.b1;
    printf("%d", *int_ptr);
    return 0;
}
```

```
struct FourByteValue {
    char b1;
    char b2;
    char b3;
    char b4;
};
```

Line	Message	True	Checker	Tool	Rule	Title
15	val.b3 = 0xd5; [implicit conversion]	False Dependent	.*overflow in .* conversion	gcc	INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data
16	val.b4 = 0xa4; [implicit conversion]	False Dependent	.*overflow in .* conversion	gcc	INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data

Audit Rules

8. When auditing an alert, if a second true violation is discovered, its alert should be marked *True*.

8. When auditing an alert, if a second true violation is discovered, its alert should be marked *True*.

Common occurrence for experienced auditors.

Second violation might be:

- Same line as first alert, but different rule
- Different line as first alert

Resolution

If you have been instructed to focus on the second violation's rule:

- Produce an alert entry associated with the second violation and mark it as *true*.
 - o This requires finding an appropriate alert or creating one.

Otherwise, ignore the second violation.

Producing an Alert for a Violation

First, try to find an existing alert associated with the violation.
A suitable alert should share the violation's:

- Condition
- Path
- Line number

E.g., there are filters in the CERT SCALE web app to help

The screenshot shows the CERT SCALE web application interface. At the top, there is a header bar with "SCALE Analysis Tool", "Help", a lightbulb icon, "SCALE at CERT", and "Copyright (c) 2007-2015 Carnegie Mellon University". Below the header, the main content area has a light blue background. It starts with a text input field containing "jasper.small". Below this are several filter controls: "ID: 206", "Verdict: --", "Previous: --", "Sort by: Priority", and "asc". There are also dropdown menus for "Checker: All Checkers", "Tool: All Tools", and "Rule: All Rules". A "Filter" button is highlighted with a red arrow. Below the filters, there is a section for "Showing 1 to 1 of 1 | Diagnostics per page: 10" with a "Go" button and a "New Diagnostic" button. At the bottom, there is a section for "Set all selected to:" with two dropdown menus and an "Update" button. The bottom of the interface features a table header with columns: ID, Flag, Verdict, Previous, Path, Line, and Message.

ID	Flag	Verdict	Previous	Path	Line	Message
----	------	---------	----------	------	------	---------

Creating an Alert Manually

Useful when you discover a condition without an associated alert.

SCALe Analysis Tool Help  SCALe at CERT Copyright (c) 2007-2015 Carnegie Mellon University

jasper.small

ID: 206 Verdict: -- Previous: -- Sort by: Priority asc

Checker: All Checkers Tool: All Tools

Rule: All Rules Filter

Showing 1 to 1 of 1 | Diagnostics per page: 10 Go [New Diagnostic](#)

Set all selected to: -- -- Update

	ID	Flag	Verdict	Previous	Path	Line	Message
--	----	------	---------	----------	------	------	---------



After Second Alert is Handled

Do not mark the original alert as *true* unless it is independently true, even if the second violation involves the same line of code.

VM Exercise: Make an Alert Determination

- Examine the diagnostic alert info at line **1427** of the file **jasper/analysis/rosecheckers.txt**
- Examine the source code at line **2105** of the file **jasper/src/src/libjasper/jpc/jpc_enc.c**
- Examine CERT rule [MEM35-C](#) in the SEI Secure Coding Rules PDF, on page **279**.
- Study for 2 minutes, then we'll discuss.

Basic

Unknown

Complex

True

False

Dependent

Supplemental

☐ Dangerous

☐ Ignored

☐ Dead

☐ Inapplicable

Example

The screenshot shows the SCALE Analysis Tool web interface. The browser address bar displays `https://scale.cert.org/projects/74`. The page header includes "SCALE Analysis Tool", "Help", "SCALE at CERT", and "Copyright (c) 2007-2015 Carnegie Mellon University".

Below the header, there is a search bar and a "New Diagnostic" button. A dropdown menu shows "10" and a "Go" button. An "Update" button is also present.

The main content area displays a table of diagnostic alerts. The table has columns: Previous, Path, Line, Message, Checker, Tool, Rule, and Title.

Previous	Path	Line	Message	Checker	Tool	Rule	Title
Unknown	/jasper-1.900.1/src/libjasper/jpc/jpc_enc.c	2105	Allocate sufficient memory for an object	MEM35-C	rosecheckers	MEM35-C	Allocate sufficient memory for an object

Below the table, the source code for the file `jasper-1.900.1/src/libjasper/jpc/jpc_enc.c` is displayed. The code is shown with line numbers from 2099 to 2109. Line 2105 is highlighted in yellow, corresponding to the diagnostic alert. The code snippet is:

```
2099     tcmt->cbikheightexpn = cp->tccp.cbikheightexpn;
2100     tcmt->cbiksty = cp->tccp.cbiksty;
2101     tcmt->csty = cp->tccp.csty;
2102
2103     tcmt->numstepsizes = tcmt->numbands;
2104     assert(tcmt->numstepsizes <= JPC MAXBANDS);
2105     memset(tcmt->stepsizes, 0, sizeof(tcmt->numstepsizes *
2106         sizeof(uint fast16 t)));
2107
2108     /* Retrieve information about the various bands. */
2109     jpc tsfb getbands(tcmt->tsfb, jas seq2d xstart(tcmt->data),
```

At the bottom of the code view, there are navigation icons and the file path `jasper-1.900.1/src/libjasper/jpc/jpc_enc.c`.

CERT Coding Rule MEM35-C

The screenshot shows a web browser window displaying the CERT Coding Rule MEM35-C. The browser's address bar shows the URL <https://www.securecoding.cert.org/confluence/display/c/MEM35-C.+Allocate+s...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar is visible, showing "Spaces", "People", and a "Create" button. The page title is "MEM35-C. Allocate sufficient memory for an object". The content area contains the following text:

The types of integer expressions used as size arguments to `malloc()`, `calloc()`, `realloc()`, or `aligned_alloc()` must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer.

Typically, the amount of memory to allocate will be the size of the type of object to allocate. When allocating space for an array, the size of the object will be multiplied by the bounds of the array. When allocating space for a structure containing a flexible array member, the size of the array member must be added to the size of the structure (see [MEM33-C. Allocate and copy structures containing a flexible array member dynamically](#)). Use the correct type of the object when computing the size of memory to allocate.

[STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator is a specific instance of this rule.](#)

Example

```
memset(  
    tcmpt->stepsizes,  
    0,  
    sizeof(tcmpt->numstepsizes * sizeof(uint_fast16_t)));
```

Alert to Audit

```
memset(  
    tcmpt->stepsizes,  
    0,  
    sizeof(tcmpt->numstepsizes * sizeof(uint_fast16_t)));
```

This is a suspicious argument to **sizeof**.
We expect the name of a type, not a product.

Badly-placed
parentheses!

Path	Line	Message	Checker	Tool	Rule	Title
/jasper-1.900.1 /src/libjasper /jpc/jpc_enc.c	2105	Allocate sufficient memory for an object	MEM35-C	rosecheckers	MEM35-C	Allocate sufficient memory for an object

Bad **sizeof** expression

Performing a multiplication within a **sizeof** expression will produce an incorrect size value.

But this is not a violation of MEM35-C, which deals with allocating memory, not zeroing out memory

Sending that size value to **memset ()** must violate some rule, but which one?

CERT Coding Rule ARR38-C

The screenshot shows a web browser window with the address bar displaying <https://www.securecoding.cert.org/confluence/display/c/ARR38-C.+Guar...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Spaces", "People", and a "Create" button. The main content area is titled "ARR38-C. Guarantee that library functions do not form invalid pointers" and is part of the "Rule 06. Arrays (ARR)" section. The page was created by David Svoboda and last modified by Will Snavelly on Feb 10, 2016. The text describes C library functions that make changes to arrays or objects, requiring at least two arguments: a pointer to the array or object and an integer indicating the number of elements or bytes to be manipulated. It states that supplying arguments to such a function might cause the function to form a pointer that does not point into or just past the end of the object, resulting in undefined behavior. A reference to Annex J of the C Standard [ISO/IEC 9899:2011] is provided, stating that it is undefined behavior if the "pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid" (see undefined behavior 109).

ARR38-C. Guarantee that library functions do not form invalid pointers

Created by David Svoboda, last modified by Will Snavelly on Feb 10, 2016

C library functions that make changes to arrays or objects take at least two arguments: a pointer to the array or object and an integer indicating the number of elements or bytes to be manipulated. For the purposes of this rule, the element count of a pointer is the size of the object to which it points, expressed by the number of elements that are valid to access. Supplying arguments to such a function might cause the function to form a pointer that does not point into or just past the end of the object, resulting in [undefined behavior](#).

Annex J of the C Standard [ISO/IEC 9899:2011] states that it is undefined behavior if the "pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid" (see [undefined behavior 109](#)).

Creating the New Alert

Scale

https://scale.cert.or 120%

SCALe Analysis Tool Help SCALe at CERT

Copyright (c) 2007-2015 Carnegie Mellon University

verdict: True *Set to true.*

previous: Unknown

scale id: 99 *Must be unique*

path: /jasper-1.900.1/src/libjasper/jpc/jpc_enc.c

line: 2105 *Same path & line*

message: Badly-placed parentheses

checker: manual *Something useful*

tool: manual

rule: ARR38-C

title: Guarantee that library functions do not form invali

severity: 3

likelihood: 3

remediation: 2 *Fill in details about the CERT rule.*

priority: 18

level: 1

create

Marking the New Alert

```
memset(  
    tcmpt->stepsizes,  
    0,  
    sizeof(tcmpt->numstepsizes * sizeof(uint_fast16_t)));
```

This is a suspicious argument to **sizeof**.
We expect the name of a type, not a product.

Badly-placed
parentheses!

Path	Line	Message	Checker	Tool	Rule	Title
/jasper-1.900.1 /src/libjasper /jpc/jpc_enc.c	2105	Badly-placed parentheses	Unknown Complex	manual	ARR38-C	Guarantee that library functions do not form invalid pointers

True

False

Dependent

Marking the Original Alert

Path	Line	Message	Checker	Tool	Rule	Title
/jasper-1.900.1/src/libjasper/jpc/jpc_enc.c	2105	Allocate object	MEM35-C	rosecheckers	MEM35-C	Allocate sufficient memory for an object

Unknown
Complex
True
False
Dependent

←

Audit Rules

9. Auditors must understand the language and the current alert's condition.

9. Auditors must understand the language and the current alert's condition.

For any codebase, an auditor must understand:

- The codebase language
 - Language specification
 - Compiler and its extensions
- Condition taxonomy
 - CWE or CERT Coding Standard
- Execution Environment
 - Operating System
 - Hardware

To gain experience quickly:

1. Focus on one condition (single item from a taxonomy).
2. Audit many alerts for just that one condition.
3. Move to another condition, repeat.

Language Standards

Language	Specification	Governing Body	SEI/CERT Standard
C	C11 , C99, C90	ISO/ IEC JTC1/SC22/W G14	SEI CERT C Coding Standard
C++	C++11 , C++97	ISO/ IEC JTC1/SC22/W G21	SEI CERT C++ Coding Standard
Java	Java Language Specification	Oracle Corporation	SEI CERT Oracle Coding Standard for Java
Perl	www.perl.org	Larry Wall	SEI CERT Perl Coding Standard

Many languages have a specification, as well as associated secure coding taxonomy

CERT Secure Coding Standards used in these slides.

VM Exercise: Make an Alert Determination

- Examine the diagnostic alert info at line **15** of the file **tutorial/analysis/rosecheckers.txt**
- Examine the source code at line **9** of the file **tutorial/src/file2.c**
- Examine CERT rule [EXP36-C](#) in the SEI Secure Coding Rules PDF, on page **99**.
- Study for 2 minutes, then we'll discuss.

Basic

Unknown

Complex

True

False

Dependent

Supplemental

☐ Dangerous

☐ Ignored

☐ Dead

☐ Inapplicable

Example

The screenshot shows the SCALE Analysis Tool interface in a web browser. The URL is <https://scale.cert.org/projects/76>. The page title is "SCALE Analysis Tool" and the copyright is "Copyright (c) 2007-2015 Carnegie Mellon University".

The interface includes a search bar and a filter button. The search results show a diagnostic for a pointer alignment issue. The diagnostic is listed in a table with columns: ID, Flag, Verdict, Previous, Path, Line, Message, Checker, Tool, and Rule.

ID	Flag	Verdict	Previous	Path	Line	Message	Checker	Tool	Rule
612	[]	[Unknown]	Unknown	file2.c	9	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C

The diagnostic message is: "Do not convert pointers into more strictly aligned pointer types". The checker is "EXP36-C" and the tool is "rosecheckers".

The source code for file2.c is displayed below the table. The code is as follows:

```
5 int main(int argc, char **argv) {
  /* [ +5 file2.c ] */
  // The malloc() and calloc() functions return a pointer to the allocated
  // memory, which is suitably aligned for any built-in type
  uint8_t *bytes = (uint8_t*) malloc(256);
  uint32_t *four_byte_chunks = (uint32_t*)(bytes);
  //...
  free(bytes);
  return 0;
}
```

The code is displayed with a yellow highlight on line 9, which is the line that triggered the diagnostic. The code is also displayed with a yellow highlight on line 9, which is the line that triggered the diagnostic.

CERT Coding Rule EXP36-C

The screenshot shows a web browser window with the address bar displaying <https://www.securecoding.cert.org/confluence/display/c/EXP36-C.+Do+not+cast+p...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Spaces", "People", and a "Create" button. The main content area is titled "EXP36-C. Do not cast pointers into more strictly aligned pointer types" and is part of the "Rule 03. Expressions (EXP)" section. The page was created by cjohns and last modified by Barbara White on Feb 02, 2016. The text explains that converting a pointer value to a more strictly aligned type is undefined behavior, citing the C Standard (6.3.2.3, paragraph 7 [ISO/IEC 9899:2011]). It also mentions that dereferencing a misaligned pointer may cause the program to terminate abnormally.

Pages / ... / Rule 03. Expressions (EXP) [Edit](#) [Watch](#) ...

EXP36-C. Do not cast pointers into more strictly aligned pointer types

Created by cjohns, last modified by Barbara White on Feb 02, 2016

Do not convert a pointer value to a pointer type that is more strictly aligned than the referenced type. Different alignments are possible for different types of objects. If the type-checking system is overridden by an explicit cast or the pointer is converted to a void pointer (`void *`) and then to a different type, the alignment of an object may be changed.

The C Standard, 6.3.2.3, paragraph 7 [ISO/IEC 9899:2011], states:

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined.

See [undefined behavior 25](#).

If the misaligned pointer is dereferenced, the program may [terminate abnormally](#). On some architectures, the cast alone may cause a loss of information even if the value is not dereferenced if the types involved have differing alignment requirements.

Example

```
int main(int argc, char **argv) {  
    uint8_t *bytes = (uint8_t*) malloc(256);  
    uint32_t *four_byte_chunks = (uint32_t*)(bytes);  
    ...  
    free(bytes);  
    return 0;  
}
```

Alert to Audit

```
int main(int argc, char **argv) {  
    uint8_t *bytes = (uint8_t*) malloc(256);  
    → uint32_t *four_byte_chunks = (uint32_t*)(bytes);  
    ...  
    free(bytes);  
    return 0;  
}
```

Path	Line	Message	Checker	Tool	Rule
file2.c	<u>9</u>	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C

More Information

```
int main(int argc, char **argv) {  
    uint8_t *bytes = (uint8_t*) malloc(256);  
    → uint32_t *four_byte_chunks = (uint32_t*)(bytes);  
    ...  
    free(bytes);  
    return 0;  
}
```

C11, s7.22.3 (Memory management functions), p1, says:

The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated).

Path	Line	Message	Checker	Tool	Rule
file2.c	9	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C

Marking the Alert

```
int main(int argc, char **argv) {  
    uint8_t *bytes = (uint8_t*) malloc(256);  
    → uint32_t *four_byte_chunks = (uint32_t*)(bytes);  
    ...  
    free(bytes);  
    return 0;  
}
```

Unknown
Complex
True

False

Dependent

Path	Line	Message	Checker	Tool	Rule
file2.c	<u>9</u>	Do not convert pointers into more strictly aligned pointer types	EXP36-C	rosecheckers	EXP36-C

Audit Rules

10. Do not arbitrarily extend the scope of a condition.

10. Do not arbitrarily extend the scope of a condition.

If a line of code is judged insecure but does not violate the condition in question:

Mark the alert *false*.

The line of code probably violates a different condition. Apply rule #8:

- 8. When auditing an alert, if a second true violation is discovered, its alert should be marked *True*.

Audit Rules

11. Code that behaves as expected might still violate a condition.

11. Code that behaves as expected might still violate a condition.

Do not assume code is secure simply because it seems to be working.

Example

The screenshot shows the SCALE Analysis Tool interface in a web browser. The address bar displays `https://scale.cert.org/projects/76`. The page title is "SCALE Analysis Tool" and the subtitle is "SCALE at CERT". The copyright notice is "Copyright (c) 2007-2015 Carnegie Mellon University".

The interface shows a list of diagnostics. The first diagnostic is selected, showing the following details:

	ID	Flag	Verdict	Previous	Path	Line	Message	Checker	Tool	Rule
☐	12	[]	[Unknown]	Unknown	file1.c	9	Do not use input functions to convert character data if they cannot handle all possible inputs	INT05-C	rosecheckers	ERR34-C

Below the table, the source code for `file1.c` is displayed. The code is as follows:

```
4 void read_integer(char *s, int *val) {  
    /* [ +4 file1.c ] */  
5     int rc;  
6     if(val == NULL) {  
7         return;  
8     }  
9     rc = sscanf(s, "%d", val);  
10    if(rc != 1) {  
11        printf("No integer found!\n");  
12        exit(1);  
13    }  
14 }
```

The line `rc = sscanf(s, "%d", val);` is highlighted in yellow. The interface also includes a search bar, a "Go" button, and a "New Diagnostic" button.

CERT Coding Rule ERR34-C

The screenshot shows a web browser displaying the CERT Coding Rule ERR34-C. The browser's address bar shows the URL <https://www.securecoding.cert.org/confluence/display/c/ERR34-C>. The page header includes the CERT logo and the Software Engineering Institute Carnegie Mellon University. The Confluence navigation bar shows the rule title "ERR34-C. Detect errors when converting a string to a number" and a search bar. The main content area contains the following text:

Created by Confluence Administrator, last modified by Jill Britton on Aug 10, 2017

The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than `INT_MAX`). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a C Standard Library function.

The `strtol()`, `strtoll()`, `strtoimax()`, `strtoul()`, `strtoull()`, `strtoumax()`, `strtof()`, `strtod()`, and `strtold()` functions convert the initial portion of a null-terminated byte string to a long int, long long int, `intmax_t`, unsigned long int, unsigned long long int, `uintmax_t`, float, double, and long double representation, respectively.

Use one of the C Standard Library `strto*()` functions to parse an integer or floating-point number from a string. These functions provide more robust error handling than alternative solutions. Also, use the `strtol()` function to convert to a smaller signed integer type such as `signed int`, `signed short`, and `signed char`, testing the result against the range limits for that type. Likewise, use the `strtoul()` function to convert to a smaller unsigned integer type such as `unsigned int`, `unsigned short`, and `unsigned char`, and test the result against the range limits for that type. These range tests do nothing if the smaller type happens to have the same size and representation for a particular implementation.

Transferring data from www.cert.org...

Example

```
void read_integer(char *s, int *val) {  
    int rc;  
    if (val == NULL) {  
        return;  
    }  
    rc = sscanf(s, "%d", val);  
    if (rc != 1) {  
        printf("No integer found!\n");  
        exit(1);  
    }  
}
```

Alert to Audit

```
void read_integer(char *s, int *val) {  
    int rc;  
    if (val == NULL) {  
        return;  
    }  
    → rc = sscanf(s, "%d", val);  
    if (rc != 1) {  
        printf("No integer found!\n");  
    }  
}
```

What happens if the input is a numeric value outside the range of **signed int**?

Line	Message	Checker	Tool	Rule
9	Do not use input functions to convert character data if they cannot handle all possible inputs	INT05-C	rosecheckers	ERR34-C

Marking the Alert

```
void read_integer(char *s, int *val) {  
    int rc;  
    if (val == NULL) {  
        return;  
    }
```

→ `rc = sscanf(s, "%d", val);`

```
    if (rc != 1) {  
        printf("No integer found!\n");  
    }
```

Line	Message	Checker	Tool	Rule
9	Do not use input function to read character data if the program does not handle all possible inputs	INT05-C	rosecheckers	ERR34-C

Unknown
Complex

True

False

Dependent

Audit Rules

12. Multiple messages help in understanding an alert.

12. Multiple messages help in understanding an alert.

Many alerts produce multiple messages.

- A tool may offer a function trace to justify an alert.
- A tool may have independent checkers that both report an alert.
- Multiple tools may produce the same alert with different messages.

For More Information

Lori Flynn

Email: lflynn@cert.org

David Svoboda

Email: svoboda@cert.org

William Snaveley

Email: wsnaveley@cert.org

U.S. Mail

Software Engineering Institute

Customer Relations

4500 Fifth Avenue

Pittsburgh, PA 15213-2612

USA

Web

www.cert.org/secure-coding

www.securecoding.cert.org

Subscribe to the CERT Secure
Coding eNewsletter

Mail to: info@sei.cmu.edu

