



Static Security Analysis (SSA)





What is Static Security Analysis (SSA)?

Static security analysis (SSA) available with Intel® Parallel Studio XE, attempts to identify errors and security weaknesses through deep analysis of source code. SSA is primarily aimed at developers and QA engineers who wish to harden their application against security attack. Since many security weaknesses are also bugs, SSA provides an effective way to discover defects, especially in code that is hard to exercise thoroughly with tests. SSA can also detect race conditions resulting from misuse of parallel programming frameworks such as OpenMP and Intel Cilk Plus.

While the variety of security attacks is practically unlimited, a number of common patterns have emerged. Probably the single biggest root cause of security weaknesses is the failure to adequately screen input data, especially data obtained from a remote user. Once an attacker has found an application weakness, they may be able to exploit that weakness to inject and execute malicious code. This often involves corrupting memory through a buffer overflow or other kind of bounds violation. Thus two broad themes emerge: protecting the integrity of memory under all circumstances and avoiding unsafe use of unchecked input.

Many patterns of unsafe usage can be discovered through static analysis. The main advantage of static analysis over dynamic analysis is that it examines all possible execution paths and variable values, not just those that are provoked during testing. This aspect of static analysis is especially valuable in security assurance, since security attacks often exercise an application in unforeseen and untested ways. SSA can detect over 250 different error conditions.

This document is an introductory tutorial describing the static security analysis feature of the Intel® Parallel Studio XE. It provides a brief description of the goals of the product feature and walks through an end-to-end example showing how it is used.

Topic
How Does SSA Work?
Setting Up for SSA
C++ Tutorial
Running SSA and Opening the Results in Intel® Inspector XE
Interacting with the Analysis Results
Investigating a Problem
Reducing Clutter with Filtering
Fixing a Problem and Rescanning
Fortran Tutorial
Additional Resources

How Does SSA Work?

Static security analysis is performed by the Intel® C++ and the Intel® Fortran Compilers operating in a special mode. In this mode the compiler dedicates more time to error analysis and bypasses the instruction generation process entirely. This allows it to find errors that go undetected during ordinary compilation. SSA requires your code to compile without serious errors using the Intel Compiler, but you do not execute the results of this compilation. You do not need to use Intel compiler to create your production binaries in order to take advantage of SSA. We will begin the tutorial with showing how to prepare an application for SSA.



SSA can be done on a partial program or a library, but it works best when operating on an entire program. Each individual file is compiled into an object module, and the analysis results are produced at the link step. The results are then viewed in Intel® Inspector XE.

The results of static analysis are often inconclusive. The tool results are best thought of as *potential* problems deserving investigation. You will have to determine whether a fix is required or not. The Intel Inspector XE GUI is designed to facilitate this process. You indicate the results of your analysis by assigning a state to each problem in the result. Intel Inspector XE saves the state information in its result files.

Over time the source will change and you will want to repeat the analysis. The first time you open your new analysis result in Intel Inspector XE, it automatically calculates a correspondence between the previous result and the new result. It uses this correspondence to initialize the state information for the new result. This means you do not have to investigate the same issues over and over again.

When you decide a problem does require fixing you should report it into your normal bug tracking system, just as you would report a defect detected by an executable test. Intel Inspector XE is not a bug tracking system. The state information tracks your progress in investigating the result of analysis. What you do with your conclusions is outside the scope of SSA and the Intel Parallel Studio XE.

Setting Up for SSA

The set up process is usually fairly simple, but it can be quite complicated, depending on the situation. We recommend you modify your build procedure (whatever it is) to create a new **build configuration** for SSA. The term build configuration refers to a mode of building your application, using specific compiler options and directing the intermediate files to specific directories. Most applications have at least two build configurations: debug and release. You will want to create one more for SSA. The SSA configuration must build with the Intel compiler with additional options set to enable SSA.

Please note the distinction between creating a new build configuration and building an existing configuration with additional options. If you build, say, your debug configuration with the additional options that enable SSA then you will get analysis results (as long as your debug configuration builds with the Intel compiler). This is a perfectly good way to do an initial product evaluation. However, it's awkward to work this way on an ongoing basis, because the object modules produced by SSA will overwrite your debug-mode object modules every time you run SSA. Just as you want to keep debug object modules separate from release object modules, you will want to keep debug object modules separate from SSA object modules. If you are going to use SSA on an ongoing basis, you will want to get set up properly.

The process for creating a new build configuration will be different for each application. If your application is built under an IDE, such as Microsoft* Visual Studio* or Eclipse*, the IDE makes it very easy to add new build configurations. The sample applications used in this tutorial can be built under Visual Studio on Microsoft* Windows* OS or with a make file on Linux* OS. We will walk through the steps needed to update this make file for SSA.

If your application build is very complex and you don't feel confident that it can be modified safely, there is an alternative set up method. You can execute your normal build under a "watcher" utility called `inspxe-inject`. This application intercepts process creations and recognizes all the compilation and link steps performed during your build. It records this information in a **build specification** file. This file can be supplied as input to another utility, `inspxe-runsc`, which invokes the Intel compiler to repeat the same build steps as your original build did. These utilities are not explained in this tutorial.

Starting the Tutorial



This tutorial can be executed on Windows or on Linux OS. It can also be done using a C++ or Fortran sample application. You should pick the configuration(s) you prefer. The C++ and Fortran tutorials are described separately. The differences between Linux OS and Windows OS are small and these cases are described together. When we come to a step that is done differently on Windows and Linux OS, we will say something like “On Windows OS, do ...”

To follow the C++ tutorial, continue in the following section. To follow the Fortran tutorial, go to the Fortran Tutorial section.

C++ Tutorial

We will be using a sample application called “tachyon_ssa”. You can find it below the Intel Inspector XE root install directory, below the “samples” subdirectory. Unzip this application to some directory on your disk.

We will start by getting set up for SSA. This is done differently on Windows and Linux OS. For Windows OS read the following section. For Linux OS, goto Setting up for SSA Using a make File on Linux* OS.

Setting up for SSA Using Microsoft* Visual Studio* on Windows* OS

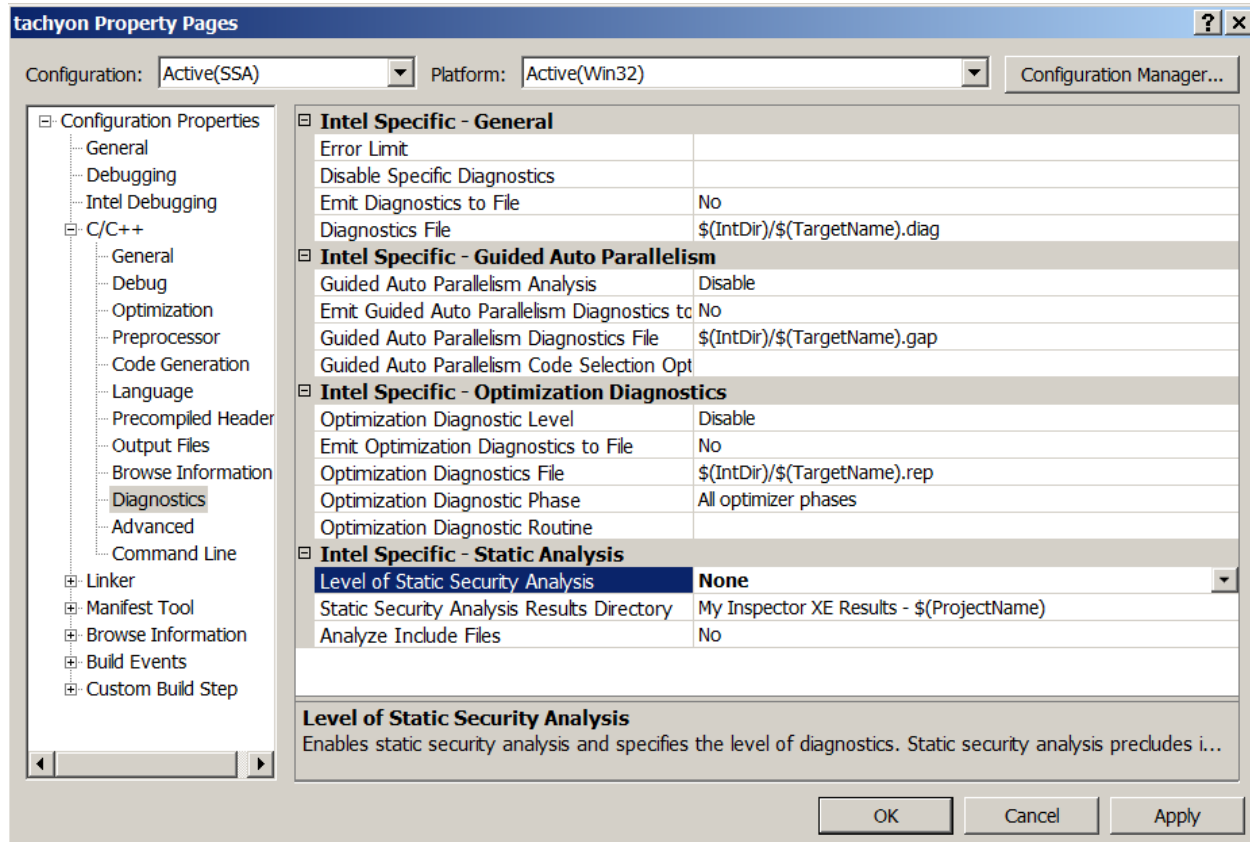
We will go through the setup process using Visual Studio solution tachyon.sln that is supplied with the sample program. Start by opening this solution in Visual Studio. If you are using Visual Studio 2008 or Visual Studio 2010 then Visual Studio will need to convert for solution file from Visual Studio 2005 format. Let it go ahead and do this conversion.

Create a new build configuration as follows:

1. Select the **Build > Configuration Manager** menu item to open the configuration manager
2. In the upper left drop-down control, select **New...**
3. In the **New Solution Configuration** dialog box, set the name to **SSA** and set to **Copy setting from** to **Debug**.
4. Click **OK** to dismiss the New Solution Configuration dialog and then **Close** to dismiss the Configuration Manager.

Before you can set the options on your new build configuration, you need to set the project to build with the Intel compiler. Follow these steps:

1. Right click the **tachyon** project in the solution explorer and select the **Intel C++ Composer XE 2011 > Use Intel C++...** menu item and click **OK** on the confirmation dialog. You will see that the icon for the project in the solution explorer has changed to indicate it will be built using the Intel compiler.
Now you are ready to set the properties for SSA. Right click the **tachyon** project and select **Properties...** Open the **C++ > Diagnostics** property page. You will see the following dialog:



Set the **Level of Static Security Analysis** option to **All Errors and Warnings (/Qdiag-enable:sc3)** and click **OK**.

That's it; you are now set up for SSA.

For information on setting up applications built using a command line script or make file, see the following section explaining the setup process on Linux OS.

Setting up for SSA Using a make File on Linux* OS

The sample application comes with a make file that can be used to build the application on Linux OS. To save time, we have included two versions of the make file. One is the original make file, **tachyon.mk**. This make file builds the application using the gcc compiler. The other is the updated make file, **tachyon_ssa.mk**, that adds a new build target, SSA. The SSA build target is just like the debug build target except for these changes:

- It builds using the Intel compiler
- It adds the additional option "-diag-enable:sc3"
- It puts the intermediate files in the SSA subdirectory.

You can compare the two make files to see the differences. This illustrates the changes that are needed to update a make file to prepare for SSA.



Running SSA and Opening the Results in Intel® Inspector XE

Now that you are set up, all you have to do is build your SSA build configuration to perform analysis. On Windows OS, you can do this by right-clicking the **tachyon** project in the solution explorer and choosing **Build**. On Linux OS, you can do this by performing the following steps:

1. Open a command shell
2. Set the environment variables for the Intel compiler by executing the `ic1vars.sh` script in the compiler bin directory, supplying the `ia32` option.
3. Execute `make -f tachyon_ssa.mk`

TIP: keep this command window open for later operations.

On Windows OS, the Intel® Inspector XE automatically opens a new result as soon as the build completes. On Linux OS, type `inspxe-gui` to invoke the Intel Inspector XE and then use the **File > Open** menu to open the result. By default, the file you want to open is called `r000sc.inspxe`, and is contained in a directory named `r000sc` below the root directory of the tachyon project.

Interacting with the Analysis Results

The remainder of this tutorial is almost identical for Windows and Linux OS. The main difference is that on Windows OS, the Intel Inspector XE GUI is embedded within Visual Studio, while on Linux OS the GUI runs as a stand-alone program. The look of the individual Intel Inspector XE windows is almost identical on Windows and Linux OS.

The initial window you will see will look something like this. (Note: you might have to drag the scrollbar up to the top to get the right line on top.)



Source Code Security Errors Intel Inspector XE 2011

Summary

ID	Problem	Sources	State	Weight	Category
P89	Uninitialized variable	vol.cpp	New	100	Initialization
vol.cpp(99): error #12143: "tex->opacity" is uninitialized					
P70	Format to arg count mismatch	getargs.cpp	New	95	Format
getargs.cpp(76): error #12068: number of format specifiers does not match number of arguments					
P1	Bad free	find_and_fix_memory_errors.cpp	New	80	Memory
find_and_fix_memory_errors.cpp(175): error #12375: referenced memory allocated through "operator new" is illegally deallocated through "free"					
P69	Divide by zero (possible)	cylinder.cpp	New	75	Other
cylinder.cpp(131): error #12062: possible divide by zero					
P73	Unsafe format specifier	parse.cpp	New	70	Format
parse.cpp(187): error #12329: specify field width in format specifier to avoid buffer overflow on argument 3 in call to "fscanf"					
P74	Unsafe format specifier	parse.cpp	New	70	Format
parse.cpp(244): error #12329: specify field width in format specifier to avoid buffer overflow on argument 3 in call to "fscanf"					
P75	Unsafe format specifier	parse.cpp	New	70	Format
parse.cpp(305): error #12329: specify field width in format specifier to avoid buffer overflow on argument 3 in call to "fscanf"					

Code Locations Observations / Timeline

ID	Description	Source	Function	State
X93	Uninitialized read	vol.cpp:99	void * newscalarvol(void *,struct vector,struct vector,int,int,int,char *,struct ...	New

Filters Sort

Severity

- Error 72 item(s)
- Warning 73 item(s)

Problem

- Bad free 1 item(s)
- Big arg passed by value 42 item(s)
- Bounds violation on string 4 item(s)
- Dead assignment 18 item(s)
- Dead statement 28 item(s)
- Default initialization 1 item(s)
- Divide by zero (possible) 1 item(s)

Source

- api.cpp 17 item(s)
- apigeom.cpp 8 item(s)
- apitrigeom.cpp 6 item(s)
- bndbox.cpp 1 item(s)
- box.cpp 1 item(s)
- coordsys.cpp 2 item(s)

State

- New 145 item(s)

Category

- C++ 12 item(s)
- Format 14 item(s)
- Initialization 4 item(s)
- Memory 4 item(s)
- Other 45 item(s)
- Pointer 10 item(s)

Suppressed

- Not suppressed 145 item(s)

Investigated

- Not investigated 145 item(s)

This window consists of three main areas. The upper left pane is the table of Problem Sets. This is your "to do" list, the things you need to investigate. The lower left pane shows the code locations corresponding to the currently selected problem set. The right pane shows the filters. It controls what problem sets are displayed and which are hidden.

You can sort the table of problem sets by clicking on any of the column headers. By default the problem set table is sorted by **weight**. The weight is a value between 1 and 100 which reflects how interesting a problem is. Problems that can do more damage have higher weight. Problems that are more likely to be true problems (as opposed to false positives) are also given higher weight. So the weight provides a natural guidance for your order of investigation.

Investigating a Problem

Let's start with an easy one. P73, the fourth one on the list, is an unsafe format specifier. Let's see what we can learn about this problem. Initially, all we know about the problem is summarized in the table entry:

P73 Unsafe format specifier parse.cpp New 70 Format

parse.cpp(187): error #12329: specify field width in format specifier to avoid buffer overflow on argument 3 in call to "fscanf"

Unsafe format specifier is the short description of the problem. The full description is shown in the shaded area. The “New” entry in the state column indicates that this problem was discovered for the first time in this analysis run and has not yet been investigated. The 70 weight value indicates the weight. The category of problem is “Format”, which means it is related to misuse for printf-style format specifications.

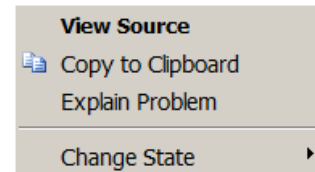
Click on this problem to select it. The lower pane refreshes itself to show the source code locations related to this problem. Here’s what it says:

ID	Description ▲	Source	Function	State
X74	Format mismatch	parse.cpp:187	unsigned int GetString(struct _jobuf *,char const *)	New

Here we see the source location (file parse.cpp, line 187). We also see the role that this source reference plays in the problem.

Format mismatch indicates that this is a place where a format string was used. It also shows the name of the function that contains the line (GetString) and its parameter signature.

One way we could get more information about a problem is to read an explanation of what that problem type means. Some SSA errors are pretty technical and require explanation. To see the explanation for this problem type, right click the problem. That brings up this pop-up menu:



Select **Explain Problem** to open a help topic that explains this problem in detail. This is the help topic for this particular problem type:

Unsafe format specifier

Some forms of formatted input can cause buffer overflow and should not be used.

Care must be taken on formatted input to avoid buffer overflow. In particular, the "%s" input format is inherently unsafe. A better alternative is "%dds" or "%*s", where *ddd* is the sized of the destination buffer, for example "%24s". If the buffer size is not a compile time constant, then "%*s" can be used, where "*" obtains the maximum size from the next input argument, for example, `scanf("%*s", sizeof(buffer), buffer);`

ID	Observation	Description
1	Format mismatch	The unsafe formatted input statement

Example

```
#include <stdio.h>

char buffer[1024];

int main(int argc, char **argv)
{
    scanf("%s", buffer); // unsafe: could overflow buffer
    // better is scanf("%*s", sizeof(buffer), buffer);
    printf("read %s\n", buffer);
    return 0;
}
```

As you can see, the problem type reference material explains more fully what the precise error condition is and its potential consequences. It explains more fully the role the various code locations play in creating the problem, and provides an example that demonstrates the problem. Where possible, it also provides better alternatives to avoid this issue

The next thing we need to do is determine whether this problem is really present in our application. To do that, we need to look at the source code. The fastest way to do that is to expand the code reference in the lower pane to expose a small snippet at the referenced location. There are two ways to do this. One is to click the plus sign in the ID column. The other is to right-click on the item in the lower pane and select "Expand All Code Snippets" from the pop-up menu. After you do this, you will see this:

ID	Description ▲	Source	Function	State
⊞X74	Format mismatch	parse.cpp:187	unsigned int GetString(struct _iobuf *,char const *)	New

```

185     char data[255];
186
187     fscanf(dfile,"%s",data);
188     if (strcmp(data, string) != 0) {
189         fprintf(stderr, "parse: Expected %s, got %s \n",string, data);
    
```

This is pretty clear. The highlighted call to fscanf has a format string with a "%s" format specifier. This reads input characters up to the next newline and stores the data in the array "data". There is no guarantee that the number of characters read will not overflow the bounds of the array, so this statement could corrupt memory. We got lucky here, because the code snippet contained all we needed to know about this problem. We will see later how we see more of the source when we need to.

Since we have confirmed that this is a real error, let's record our conclusion. Right click on the problem and select **Change State > Confirmed** from the pop-up menu.

View Source

Copy to Clipboard

Explain Problem

Change State ▶

- Not fixed
- Confirmed**
- Fixed
- Not a problem

Now we're done with that problem. You can see the state is updated in the problem set table:

ID	Icon	Description	Source	State	Count	Category
P73	⊞	Unsafe format specifier	parse.cpp	Confirmed	70	Format

```

parse.cpp(187): error #12329: specify field width in format specifier to avoid buffer overflow on argument 3 in call to "fscanf"
    
```

Reducing Clutter with Filtering

Filters allow you to focus on the problems you are interested in and hide the problems you want to ignore.

Once a problem has been investigated there is usually no reason to look at it again. One of the nicest uses for filters is to hide all the problems you are finished investigating. Go all the way to the bottom of the filter window and click on the "Not investigated" item inside the **Investigated** filter.

Investigated	
Investigated	1 item(s)
Not investigated	144 item(s)

When you do that, the filter item redraws to indicate that it is active:

Investigated	
Not investigated	144 item(s)

Notice how that problem we marked as **Confirmed** disappeared from the table of problem sets when we did that. It's a good idea to keep this filter set like this while you work on analyzing results.

The first several filters, Severity, Problem, Source, State, and Category, correspond to columns in the table of problem sets. You can hide all rows in the table that do not match a specific value in some column. Click on the second line in the Source filter (apigeom.cpp). It will look like this:

Source	All
apigeom.cpp	8 item(s)

Notice how the problem set table has also redrawn to show only problems in this source file. You can turn off a filter by clicking on the **All** box, but leave it turned on for now.

Investigating a Second Problem

Take a look at another problem in the solution. This time pick problem P94, which should be the second one you now see in the table of problem sets:

P94		Null pointer dereference (possible)	apigeom.cpp; vector.cpp	New	60	Pointer
<p>apigeom.cpp(165): error #12172: dereference of pointer "normals" which is possibly set to null at (file:apigeom.cpp line:143) vector.cpp(77): error #12172: dereference of pointer "a" which is possibly set to null at (file:apigeom.cpp line:143)</p>						

This problem is a little more interesting. As you can see, it has two source locations in different files. The problem type is "Null pointer dereference (possible)". The full description describes two places where a pointer is dereferenced that could possibly be set to null. In both cases the place where the pointer was possibly set to null is the same (apigeom.cpp, line 143).

This illustrates why we call this the table of **problem sets** instead of **problems**. Here SSA has combined two related problems into one problem set because it is likely that both problems could have a common solution.

Let's investigate to see what is happening here. As before, select this problem set and look at the lower pane to see the related source code:

ID	Description ▲	Source	Function	State
X106	Memory write	apigeom.cpp:143	void rt_sheightfield(void *,struct vector,int,int,double *,double,double)	New
X108	Null dereference	apigeom.cpp:165	void rt_sheightfield(void *,struct vector,int,int,double *,double,double)	New
X123	Null dereference	vector.cpp:77	void VNorm(struct vector *)	New

Here we see three code locations. One is the place where the pointer was assigned (Memory write) and the other two places are where a null pointer value could be dereferenced. Take a closer look by right-clicking one of these and selecting **Expand All Code Snippets** from the pop-up menu:

```

X106 Memory write apigeom.cpp:143 void rt_sheightfield(void *,struct vector,int,int,double *,double,double) New
141
142 vertices = (vector *) malloc(m*n*sizeof(vector));
143 normals = (vector *) malloc(m*n*sizeof(vector));
144
145 offset.x = ctr.x - (wx / 2.0);

X108 Null dereference apigeom.cpp:165 void rt_sheightfield(void *,struct vector,int,int,double *,double,double) New
163 /* build normals from vertex list */
164 for (x=1; x<m; x++) {
165 normals[x] = normals[(n - 1)*m + x] = rt_vector(0.0, 1.0, 0.0);
166 }
167 for (y=1; y<n; y++) {

X123 Null dereference vector.cpp:77 void VNorm(struct vector *) New
75 flt len;
76
77 len=sqrt((a->x * a->x) + (a->y * a->y) + (a->z * a->z));
78 if (len != 0.0) {
79 a->x /= len;

```

Now the problem is starting to become apparent. The memory write assigned a value from the routine “malloc”. Malloc returns a null pointer when an application runs out of memory. Apparently this application didn’t check for that case before using the pointer. The second snippet indicates that we are using what looks like the same pointer variable here (“normals”), 20 lines below the assignment in the same file and subroutine. What about the third snippet? We can see this is using a pointer (“a”), but how is it related to the normals pointer variable? And this is a completely different source file (vector.cpp). How is that related to that malloc call?

Here you must remember that SSA is doing whole-program, cross file analysis. It can analyze the flow of data values through procedure calls, even across files. But how did the value that was received from malloc get into the pointer named “a”?

SSA helps you answer questions like this by providing “traceback” information. To see the traceback information, go to the **Sources** view. Right-click one of these code references and select **View Source** from the pop-up menu. This is what you will see:

ID	Description	Source	Function	State
X113	Memory write	apigeom.cpp:142	void rt_sheightfield(void *,struct vector,int,int,double *,double,double)	New
X115	Null dereference	apigeom.cpp:164	void rt_sheightfield(void *,struct vector,int,int,double *,double,double)	New
X130	Null dereference	vector.cpp:77	void VNorm(struct vector *)	New

This is the Sources view. It contains two pairs of windows, each of which is showing a section of source code and (on the right) a Traceback. At the top left you see a highlighted box that says **Sources**. To the left of that you see another box that says **Summary**. If you click the Summary box it takes you back to the summary view we were looking at earlier.

Neither of these code windows shows the source location we are interested in. However, down at the bottom is the same table of code locations we saw in the summary view. If you look closely you will see a little red tag on one, and a blue tag on another. This tells you which code locations are on display. The same red and blue tags are shown in the upper left in the code windows.

We are interested in the code location in `vector.cpp`. Double-click it. Now the source for that code reference appears. You could also right click on it and choose **Set as Related Observation** or **Set as Focus Observation** from the pop-up menu. The red tag is for the focus observation and the blue tag is for the related observation. Here's what you see:

ID	Description	Source	Function	State
X113	Memory write	apigeom.cpp:142	void rt_sheightfield(void *,struct vector,int,int,double *,double,double)	New
X115	Null dereference	apigeom.cpp:164	void rt_sheightfield(void *,struct vector,int,int,double *,double,double)	New
X130	Null dereference	vector.cpp:77	void VNorm(struct vector *)	New

Look to the right of the middle pane, at the windows that says **Traceback** on it. This contains three lines. Go ahead and click on these lines one after another. You can see the left code window refresh itself with different source positions. The one on the end of the traceback is in fact the same source location you see in the top screen, the place where `malloc` was called.

The traceback is showing you the connections between where you started (the `malloc`) and where you ended up (the possibly null dereference of "a"). The interesting place is the middle point in the traceback.

```

Related Observation: apigeom.cpp:178 - Null dereference
173     normals[addr] = rt_vector(
174         -(field[addr + 1] - field[addr - 1]) / (2.0 * xinc),
175         1.0,
176         -(field[addr + m] - field[addr - m]) / (2.0 * yinc));
177
178     MyVNorm(&normals[addr]);
179 }
180 }
181
182 /* generate actual triangles */
183 for (y=0; y<(n-1); y++) {

```

```

Traceback
void VNorm(struct vector *) - vector.cpp ...
void rt_sheightfield(void *,struct vector,i...
void rt_sheightfield(void *,struct vector,i...

```

Here we see a call to MyVNorm. If you look back at the place where the dereference happened, you will see that it is in a subroutine called VNorm. MyVNorm and VNorm do not look like the same thing, but they are. This is the line in the apigeom.cpp that proves it:

```
#define MyVNorm(a)          VNorm ((vector *) a)
```

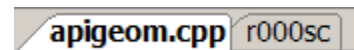
This call ties the two things together. We started in the routine called `rt_sheightfield`, we malloc-ed some storage and assigned it to a pointer called `normals`. Then we called `VNorm`, passing `normals` (actually `&normals[addr]`, but that could be null too). If we look back at the code in `VNorm`, you can see that “a” is the name of the formal parameter. That is how the value from malloc got into “a”: it was passed at this call.

Fixing a Problem and Rescanning

Now go ahead and fix this problem. To fix this problem you need to add some code after the call to malloc to test the result for null. If the result is null, then you would perform some kind of error recovery. In this case we can recover from the error by simply returning from the subroutine in question.

To modify the source we need to get into a real source editor on line 143 in `apigeom.cpp`, right after the malloc call. You can enter your normal source editor by double-clicking the line in the sources view, or by right-clicking that line and selecting **Edit Source** from the pop-up menu. On Linux OS this will activate whatever editor is selected by the `EDITOR` environment variable. On Windows OS this opens the Visual Studio source editor.

On Windows OS the editor window will open a new tab in the same tab group window that contains the results. So opening the source for editing will hide the result. To view the results again, click the `r000sc` tab.



If you have a big screen, you can put the result and the source files you are editing in different tab groups.

You can see the following in the editor:

```
vertices = (vector *) malloc(m*n*sizeof(vector));
normals = (vector *) malloc(m*n*sizeof(vector));
```

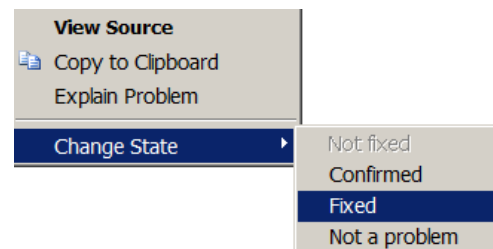
Rewrite this code like this:

```
vertices = (vector *) malloc(m*n*sizeof(vector));
if (vertices == NULL) {
    return;
}
normals = (vector *) malloc(m*n*sizeof(vector));
```

Type or copy-paste these three code lines into the source exactly as shown, save the result, and close the editor but do not rebuild the application yet. In fact this change does not really fix the problem you intended to fix, but make the change anyway.

Now go back into the summary view. In Visual Studio, click on the **r000sc** tab to get back to the Intel Inspector XE GUI. Click on the box that says **Summary** at the upper left of the Sources view.

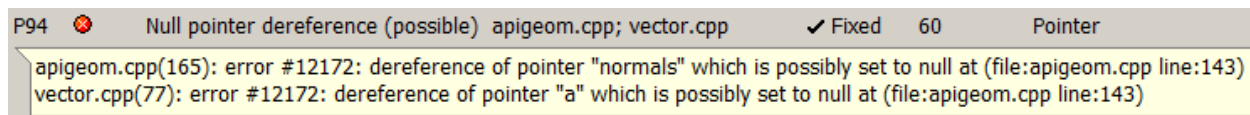
Look at the problem set table. There are actually two problems related to this code: P94 (the one we investigated), and P93, which is a similar problem with the line above it. In fact the fix we just typed in has actually corrected P93, but it did not fix P94. Still, go ahead and use the right-click pop-up menu to change the state of P94 to **Fixed**.



Notice that as soon as you do this, the problem disappears, since you set the filter set to show only uninvestigated problems. To see that problem again, go over to **Filters** pane, go down to the bottom and turn off the Investigated filter by clicking on **All**:



Now you can see the problem again, and see that it really is marked as **Fixed**.




Now, go ahead and rebuild the application to create a new analysis result and open the new result, **r001sc**. On Windows OS, you can do all this by hitting the F7 key. You will want to close the Visual Studio Output window when the build completes. On Linux OS you should repeat the steps you used earlier to build (type `make -f "tachyon_ssa.mk"` in your command window, then use **File > Open** in the Intel Inspector GUI to open **r001sc**).

The first thing you will notice is that most problems now say "Not fixed" instead of "New" in their state. This demonstrates the way that Intel Inspector XE automatically initializes the state in the new result, **r001sc**, from the previous result, **r000sc**. Problems that were seen before are no longer considered "New". They are simply not investigated yet, which is what the "Not fixed" state indicates.

You will also notice that the problem we investigated earlier, the unsafe format problem, is still in a "Confirmed" state, just as we marked it in **r000sc**.

Use the filter to select only the problems in file `apigeom.cpp`. You will notice that there are only 7 problems in **r001sc** (there were 8 in **r000sc**). This reflects the fact that our source change did fix one problem. Look at the problem that is the top of the list. Here's what it looks like now:

P93  Null pointer dereference (possible) apigeom.cpp; vector.cpp Regression 60 Pointer

```
apigeom.cpp(168): error #12172: dereference of pointer "normals" which is possibly set to null at (file:apigeom.cpp line:146)
vector.cpp(77): error #12172: dereference of pointer "a" which is possibly set to null at (file:apigeom.cpp line:146)
```

This is the problem that used to be number P94 in the old result. Now it is P93, but it is the same problem that we marked as “Fixed” in r000sc. Intel Inspector XE correctly determined that this is the same problem, and since it is still present it is set to **Regression** instead **Fixed**. This indicates that a fix did not really make the problem go away. **Regression** is considered an uninvestigated state, so this problem would still be seen if you set the filter to show only problems whose investigation state is **Not investigated**

Summary and Review

To review, we have seen:

1. The table of problem sets summarizes the problems found by SSA.
2. Selecting a problem set shows the associated code locations in the lower pane.
3. Dig into a problem by viewing code snippets, going to the **Sources** view (where you can see tracebacks), or going to your source editor.
4. Get a full explanation of what a problem type means by using the “Explain Problem” menu item to access the problem type reference material.
5. Set the state of a problem set to record the results of your investigation.
6. Use filters to hide problems that you are done investigating, or to focus on particular source files or particular classes of problems.
7. SSA sometimes combines related problems into problem sets to reduce investigation time.
8. SSA automatically carries state information forward from result to result. It keeps track of what problems are new, what problems were investigated, and verifies that fixed problems are really fixed.

Fortran Tutorial

We will be using a sample application called “FortranDemo”. You can find it below the Intel Inspector XE root install directory, below the “samples” subdirectory. Unzip this application to directory on your disk.

We will start by getting set up for SSA. This is done differently on Windows and Linux OS. For Windows OS read the following section. For Linux OS go to Setting up for SSA Using a make File on Linux* OS.

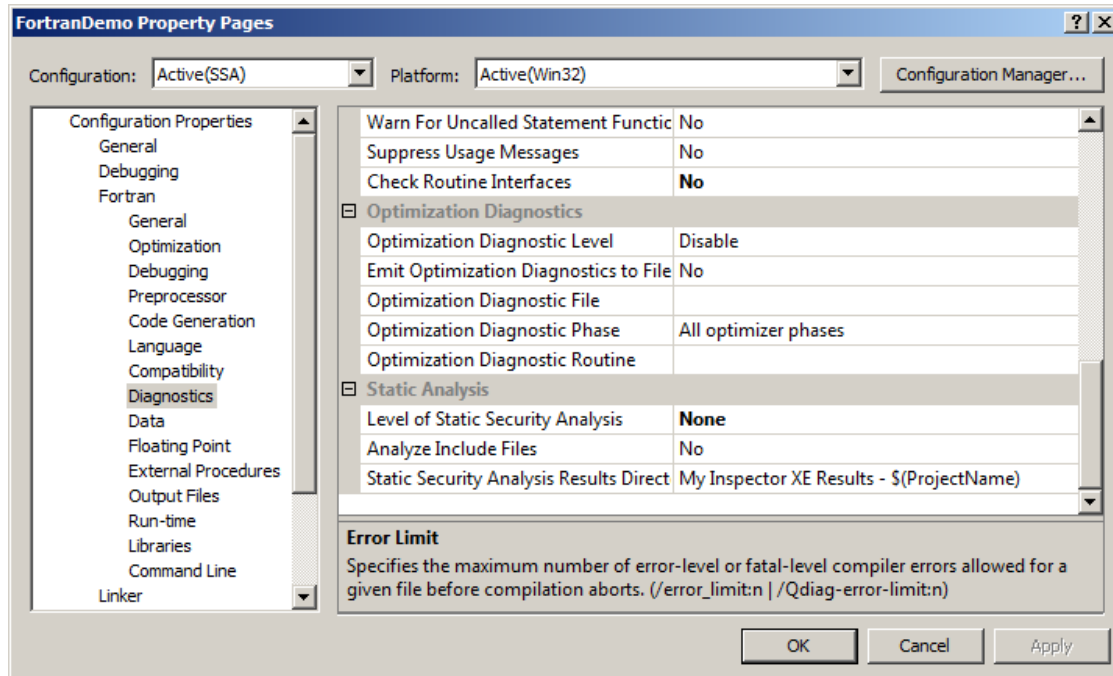
Setting up for SSA Using Visual Studio on Windows OS

We will go through the setup process using Visual Studio solution FortranDemo.sln that is supplied with the sample program. Start by opening this solution in Visual Studio. If you are using Visual Studio 2008 or Visual Studio 2010 then Visual Studio will need to convert for solution file from Visual Studio 2005 format. Let it go ahead and do this conversion.

Create a new build configuration as follows:

1. Select the **Build > Configuration Manager** menu item to open the configuration manager.
2. In the upper left drop-down control, select **<New...>**.
3. In the **New Solution Configuration** dialog, set the name to “SSA” and set to **Copy setting from** to **Debug**.
4. Click **OK** to dismiss the **New Solution Configuration** dialog and then **Close** to dismiss the **Configuration Manager**.

Now you are ready to set the properties for SSA. Right click the **FortranDemo** project and select **Properties...** Open the **Fortran > Diagnostics** property page. You will see the following dialog:



Set the **Level of Static Security Analysis** option to **All Errors and Warnings (/Qdiag-enable:sc3)** and click **OK**.

You are now set up for SSA.

For information on setting up applications built using a command line script or make file, see the following section explaining the setup process on Linux OS.

Setting up for SSA using a make file on Linux OS

The sample application comes with a make file that is used to build the application on Linux OS. The included make file will build the application using the Intel Fortran compiler and includes a build target, SSA. The SSA build target is just like the debug build target except for these changes:

1. It adds the additional option "-diag-enable sc3"
2. It puts the intermediate files in the SSA subdirectory.

This illustrates the changes that are needed to update a make file to prepare for SSA.

Running SSA and Opening the Results in Intel Inspector XE

Now that you are set up, all you have to do is build your SSA build configuration to perform analysis. On Windows OS, you can do this by right-clicking the **FortranDemo** project in the solution explorer and choosing **Build**. On Linux OS, you can do this by performing the following steps:

1. Open a command shell

2. Set the environment variables for the Intel compiler by executing the `ifortvars.sh` script in the compiler bin directory, supplying the `ia32` option.
3. Execute `make ssa`

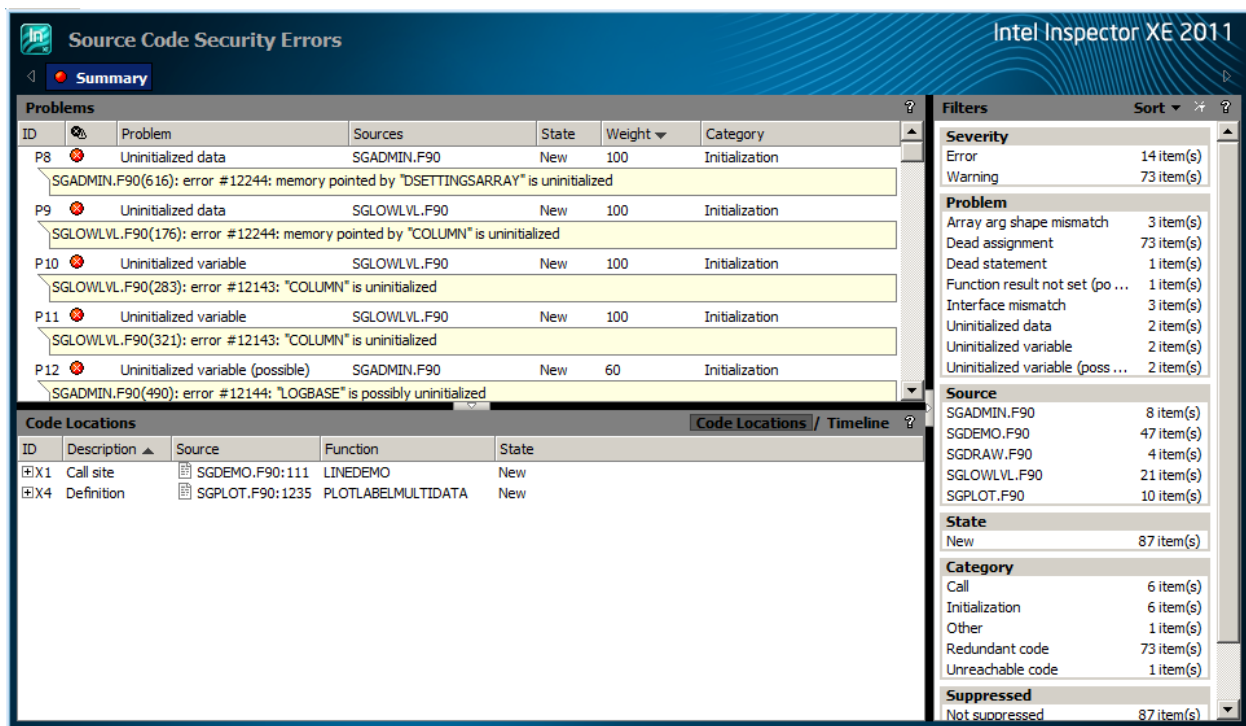
TIP: keep this command window open for later operations.

On Windows OS, the Intel Inspector XE automatically opens a new result as soon as the build completes. On Linux OS, type `inspxe-gui` to invoke Intel Inspector XE and then use the **File > Open** menu to open the result. By default, the file you want to open is called `x000sc.inspxe`, and is contained in a directory named `x000sc` below the root directory of the **FortranDemo** project.

Interacting with the Analysis Results

The remainder of the Fortran demo is almost identical for Windows and Linux OS. The main difference is that the Intel Inspector XE GUI is embedded within Visual Studio on Windows OS, while on Linux OS the GUI runs as a stand-alone program. The look of the individual Intel Inspector XE windows is almost identical on Windows and Linux OS.

The initial window you will see will look something like this. (Note: you might have to drag the scrollbar up to the top to get the right line on top.)



This window consists of three main areas. The upper left pane is the table of Problem Sets. This is your "to do" list, the things you need to investigate. The lower left pane shows the code locations corresponding to the currently selected problem set. The right pane shows the filters. It controls what problem sets are displayed and which are hidden.

You can sort the table of problem sets by clicking on any of the column headers. By default the problem set table is sorted by **weight**. The weight is a value between 1 and 100 which reflects how interesting a problem is. Problems that can do more damage

have higher weight. Problems that are more likely to be true problems (as opposed to false positives) are also given higher weight. So the weight provides a natural guidance for your order of investigation.

Look at P1, Array arg shape mismatch, and see what we can learn about this problem.

Investigating a Problem

So far all we know about the problem is summarized in the table entry:

P1		Array arg shape mismatch	SGDEMO.F90; SGPLOT.F90	New	25	Call
SGDEMO.F90(111): error #12028: shape of actual argument 2 in call to "PLOTLABELMULTIDATA" doesn't match the shape of formal argument "LABELS"; "PLOTLABELMULTIDATA" is defined at (file:SGPLOT.F90 line:1235)						

Array arg shape mismatch is the short description of the problem. The full description is shown in the shaded area. The **New** entry in the state column indicates that this problem was discovered for the first time in this analysis run and has not yet been investigated. The 25 weight value indicates the weight. The category of problem is **Call**, which means it is related to the assignment of an actual array argument to a dummy argument of a different size.

Click on this problem to select it. The lower pane refreshes itself to show the source code locations related to this problem:

Code Locations					Code Locations / Timeline ?
ID	Description ▲	Source	Function	State	
X1	Call site	SGDEMO.F90:111	LINEDEMO	New	
X4	Definition	SGPLOT.F90:1235	PLOTLABELMULTIDATA	New	

Here we see two source locations. The first is where the function PLOTLABELMULTIDATA was called by the subroutine LINEDEMO ("Call site") and the other is where PLOTLABELMULTIDATA is defined.

One way we could get more information about a problem is to read an explanation of what that problem type means. Some SSA errors are pretty technical and require explanation. To see the explanation for this problem type, right click on the problem. That brings up this pop-up menu:

View Source
Copy to Clipboard
Explain Problem
Change State ▶

Select **Explain Problem**. This brings up a help topic that explains this problem in detail. This is the help topic this particular problem type:

Array parameter shape mismatch

The type of an actual argument does not match the corresponding formal parameter at a subroutine call.

Specifically, this error occurs when both the actual and formal parameter types are arrays, but the arrays have different shape.

This same kind of error can also happen when a FORTRAN dummy argument of type subroutine is invoked. That is, the subroutine that is invoked through a dummy argument might exhibit the same problem as can occur in a direct call. In this case, the problem may or may not occur, depending on what subroutine was passed to the dummy argument of subroutine type. There will be an additional observation in such cases that identifies the call site where the subroutine argument was passed in.

ID	Observation	Description
1	Call site	The actual argument that was passed
2	Definition	The definition of the called procedure and shows the declaration of the formal parameter

Example

```
subroutine mysub(m)
type mytype1
  integer f1
  real f2
end type mytype1
type (mytype1), dimension(2,3) :: m
print *, m
end

type mytype2
  integer g1
  real g2
end type mytype2
type (mytype2), dimension(3,2) :: n
n%g1 = 1
call mysub(n)
! shapes of dummy argument and actual argument are different.
print *, n%g1
end
```

Copyright © 2010, Intel Corporation. All rights reserved.

As you can see, the problem type reference material explains more fully what the precise error condition is, its potential consequences, and provides an example that demonstrates the problem.

Next, determine whether this problem is really present in our application. To do that, look at the source code. The fastest way to do that is to expand the code reference in the lower pane to expose a small snippet at the referenced location. There are two ways to do this. One is to click the plus sign in the ID column. The other is to right-click on the item in the lower pane and select **Expand All Code Snippets** from the pop-up menu. You will see this:



Code Locations					Code Locations / Timeline ?
ID	Description ▲	Source	Function	State	
X1	Call site	SGDEMO.F90:111	LINEDEMO	New	
	109	retcode=PlotGraph(lineGraph,2,lineAxes,4)			
	110				
	111	retcode=PlotLabelMultiData(lineGraph,lineLabels,lineData, &			
	112	4,lineDataSets,lineAxes(1),lineAxes(2))			
	113	END SUBROUTINE			
X4	Definition	SGPLOT.F90:1235	PLOTLABELMULTIDATA	New	
	1233 !	Plots multiple label/numeric data sets. Must happen after PlotGraph			
	1234				
	1235	INTEGER FUNCTION PlotLabelMultiData(graph,labels,data, &			
	1236	numSettings,dSettingsArray, &			
	1237	axis1,axis2)			

In this case, the code snippets do not really show enough as to what issue is. Double click the first snippet to open the Sources view.

ID	Description ▲	Source	Function	State
X1	Call site	SGDEMO.F90:111	LINEDEMO	New
X4	Definition	SGPLOT.F90:1235	PLOTLABELMULTIDATA	New

You can see in the bottom source window that the second argument of PlotLabelMultiData, labels, is a single element array. If you were to scroll up in the top source window you would see that the actual argument, lineLabels, is defined as a six element array, hence the error.

Since you have confirmed that this is a real error, record our conclusion. Right click the problem and select **Change State > Confirmed** from the pop-up menu.

- View Source
- Copy to Clipboard
- Explain Problem
- Change State**
 - Not fixed
 - Confirmed**
 - Fixed
 - Not a problem

You can see the state is updated in the problem set table:

P1		Array arg shape mismatch	SGDEMO.F90; SGPLOT.F90	Confirmed	25	Call
SGDEMO.F90(111): error #12028: shape of actual argument 2 in call to "PLOTLABELMULTIDATA" doesn't match the shape of formal argument "LABELS"; "PLOTLABELMULTIDATA" is defined at (file:SGPLOT.F90 line:1235)						

Reducing Clutter with Filtering

Filters allow you to focus on the problems you are interested in and hide the problems you want to ignore.

Once a problem has been investigated there is usually no reason to look at it again. One of the nicest uses for filters is to hide all the problems you are finished investigating. Go all the way to the bottom of the filter window and click the **Not investigated** item inside the **Investigated** filter.

Investigated	
Investigated	1 item(s)
Not investigated	86 item(s)

When you do that, the filter item redraws to indicate that it is active:

Investigated	All
Not investigated	86 item(s)

Notice how that problem we marked as **Confirmed** disappeared from the table of problem sets when we did that. It a good idea to keep this filter set like this while you work on analyzing results.

The first several filters, Severity, Problem, Source, State, and Category, correspond to columns in the table of problem sets. You can hide all rows in the table that do not match a specific value in some column. Click on the second line in the Source filter (SGDEMO.F90). It will look like this:

Source	All
SGDEMO.F90	46 item(s)

Notice how the problem set table has also redrawn to show only problems in this source file. You can turn off a filter by clicking the **All** box, but leave it turned on for now.

Investigating a Second Problem

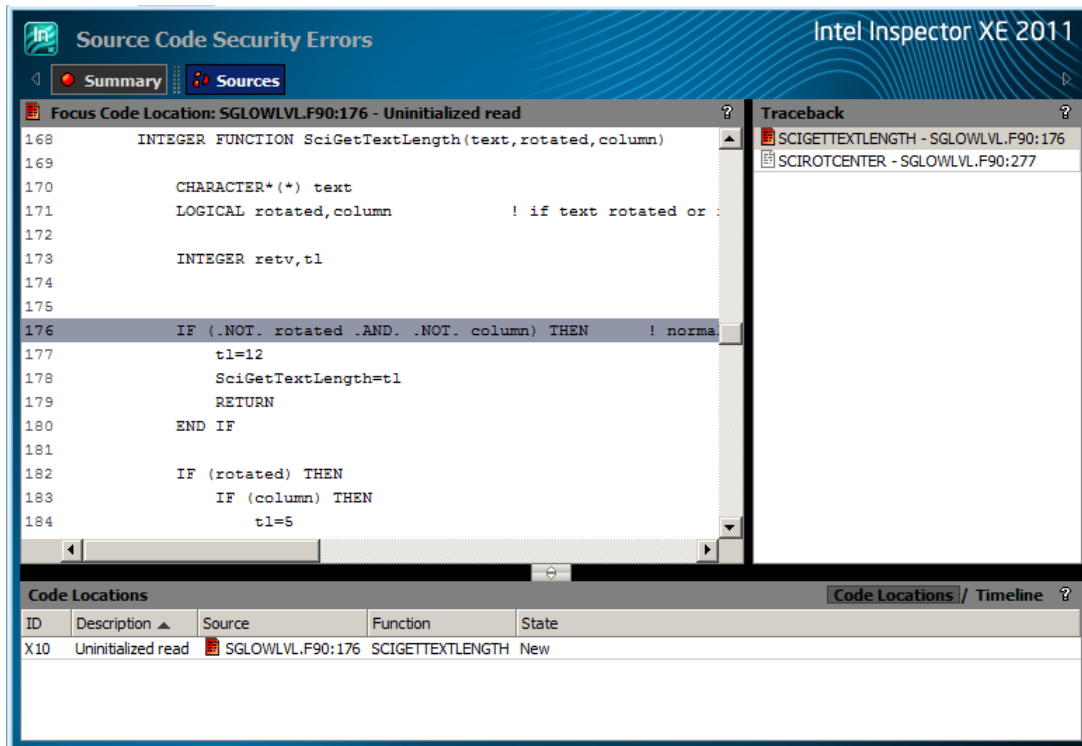
Take a look at another problem in the solution. Pick problem P9, which should be the second one you now see in the table of problem sets:

P9		Uninitialized data	SGLOWLVL.F90	New	100	Initialization
SGLOWLVL.F90(176): error #12244: memory pointed by "COLUMN" is uninitialized						

To investigate this issue, select this problem set and look at the lower pane to see the related source code:

ID	Description ▲	Source	Function	State
X10	Uninitialized read	SGLOWLVL.F90:176	SCIGETTEXTLENGTH	New
<pre> 174 175 176 IF (.NOT. rotated .AND. .NOT. column) THEN ! normal text 177 t1=12 178 SciGetTextLength=t1 </pre>				

This shows us the use of column, but it doesn't tell us enough to understand the problem. To see the actual source, right click the code snippet and select **View source**. This is what you will see:



This is the **Sources** view. It contains one pair of windows, showing a section of source code and (on the right) a Traceback. Up at the top left you see a highlighted box that says **Sources**. To the left of that you see another box that says **Summary**. You can click this box to go back to the summary view we were looking at earlier.

The source shows that “column” is an input argument to the function `SciGetTextLength`. So where is `SciGetTextLength` being called? If you remember, SSA performs a whole-program, cross file analysis. It analyzes the flow of data values through procedure calls, even across files. Traceback information answers this question.

If you look at the Traceback, you will see that `SciGetTextLength` is called by `SciRotCenter`. Click `SCIROTCENTER` in the Traceback window and you will see the call to `SciGetTextLength` in the source code window. Scrolling up through the source code window you can see that ‘column’ has not been initialized prior to calling `SciGetTextLength`.

Fixing a Problem and Rescanning

Let’s fix this problem by entering the statement “column = .true.” at source line 276. To bring up your normal source editor, you can either double click that line the sources view window, or right click that line and select **Edit Source** from the pop-up menu. On Linux OS this will activate whatever editor is selected by the `EDITOR` environment variable. On Windows OS this opens the Visual Studio source editor.

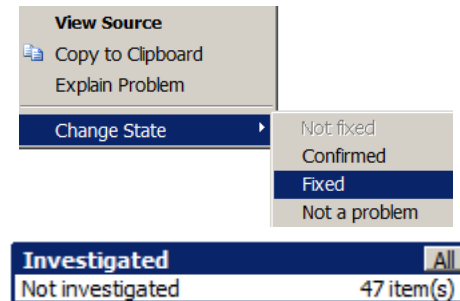
On Windows OS, the editor window will open a new tab in the same tab group window that contains the results. So opening the source for editing will hide the result. To view the results again, click the `r000sc` tab.



After making the edit, save the result, and close the editor but do not rebuild the application yet.

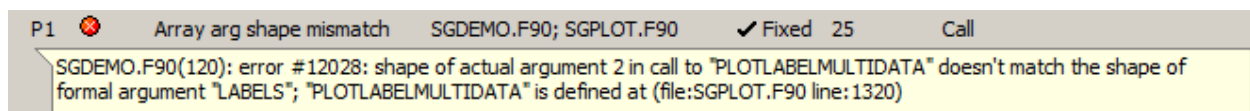
Now, go back into the summary view. On Visual Studio you may need to click on the **r000sc** tab to get back to the Intel Inspector GUI. To get from the **Sources** view to the **Summary** view, click on the box that says **Summary** at the upper left of the **Sources** view.

Use the right-click pop-up menu to change the state of P9 to say **Fixed**.



Notice that as soon as you do this, the problem disappears, since you set the filter set to show only uninvestigated problems. To see that problem again, go to **Filters** pane, go down to the bottom and turn off the Investigated filter by clicking on **All**:

Now you can see the problem again, and see that it really is marked as **Fixed**.



Now go ahead and rebuild the application to create a new analysis result and open the new result, **r001sc**. (P9, P10, and P11 were all related issues and generally you would have wanted to investigate all three prior to rescanning, but for the sake of this example go ahead and rescan now). On Windows OS you can do all this by hitting the F7 key. You will want to close the Visual Studio Output window when the build completes. On Linux OS you should repeat the steps you used earlier to build (rerun the make file in your command windows, then use **File > Open** in the Intel Inspector GUI to open **r001sc**).

The first thing you will notice is that most problems now say **Not fixed** instead of **New** in their state. This demonstrates the way that Intel Inspector XE automatically initializes the state in the new result, **r001sc**, from the previous result, **r000sc**. Problems that were seen before are no longer considered **New**. They are simply not investigated yet, which is what the **Not fixed** state indicates.

You will also notice that the problem we investigated earlier, the array parameter shape mismatch problem, is still in a **Confirmed** state, just as we marked it in **r000sc**.

You will see that the number of errors has been reduced from 14 to 13, however, the error eliminated was P10, one of the related errors pointed out earlier. P9 which was marked as **Fixed** in **r000sc** is now marked as a **Regression**. This indicates that the problem still exists. **Regression** is considered an uninvestigated state, so this problem would still be seen if you set the filter to show only problems whose investigation state is **Not investigated**. If you were to investigate P10 (P11 in **r000sc**) you would see that there is another instance of `SciGetTextLength` being called without "column" being initialized. Fixing this issue will then also resolve P9.

Summary and Review







To review, we have seen:

1. The table of problem sets summarizes the problems found by SSA.
2. Selecting a problem set shows the associated code locations in the lower pane.
3. Dig into a problem by viewing code snippets, going to the Sources view (where you can see tracebacks), or going to your source editor.



4. Get a full explanation of what a problem type means by using the “Explain Problem” menu item to access the problem type reference material.
5. Set the state of a problem set to record the results of your investigation.
6. Use filters to hide problems that you are done investigating, or to focus on particular source files or particular classes of problems.

SSA automatically carries state information forward from result to result. It keeps track of what problems are new, what problems were investigated, and verifies that fixed problems are really fixed.

Additional Resources	
Intel® Software Network Forums	
Intel® Software Products Knowledge Base	
Intel Software Network Blogs	
Intel Parallel Studio Website	
Intel® Threading Building Blocks Website	
Parallelism blogs, papers and videos	



Optimization Notice

Intel® Compiler includes compiler options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel® Compiler are reserved for Intel microprocessors. For a detailed description of these compiler options, including the instruction sets they implicate, please refer to “Intel® Compiler User and Reference Guides > Compiler Options.” Many library routines that are part of Intel® Compiler are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® Compiler offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® Compiler, with respect to Intel’s compilers and associated libraries as a whole, Intel® Compiler may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other compilers to determine which best meet your requirements.

Performance measurements are made using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing

© 2010, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Xeon and Intel Cilk are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.