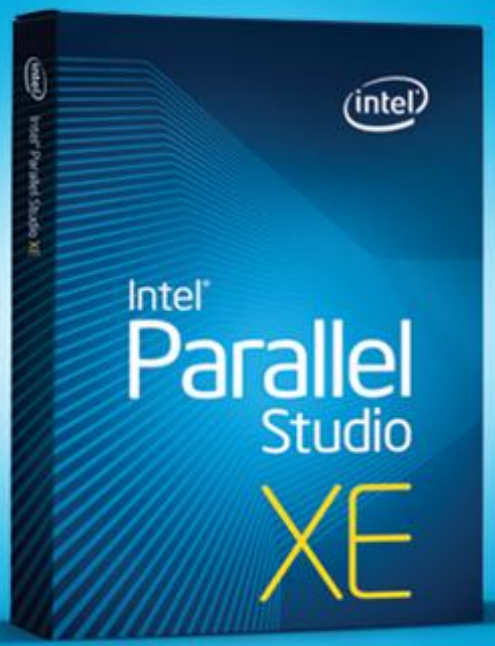
Improve C++ Code Quality with Static Security Analysis (SSA)

with Intel® Parallel Studio XE



This document is an introductory tutorial describing how to use static security analysis (SSA) on C++ code to improve software quality, either by eliminating bugs or making the code more secure, or both. 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.

Static Security Analysis is available in the Intel® Parallel Studio XE and Intel® Cluster Studio XE suites.

Introduction

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.

What is Static Security Analysis (SSA)?

Static security analysis (SSA) attempts to identify errors and security weaknesses through deep analysis of source code. SSA is primarily aimed at developers and QA engineers who wish detect software defects early in the development cycle in order to reduce time, cost and increase ROI. It also assists developers in hardening their application against security attack. 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.

Many coding errors and 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. They include buffer overflows and boundary violations, misuse of pointers and heap storage, memory leaks, use of uninitialized variables and objects, unsafe/incorrect use of C/C++ or Fortran features and libraries.

How does Static Security Analysis 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 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. For Microsoft* Windows* when using Microsoft* Visual Studio*, set up is automated with a simple menu/dialog driven approach that automatically configures your solution for SSA. Details on how to use this and how it works are in the tutorial that follows.

In Linux* OS or when building with a make file or command line script in Windows 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. 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.

Start of C++ Tutorial

This tutorial can be executed on Windows or on Linux OS. It can also be done using a C++ sample application. 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 ..."

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.

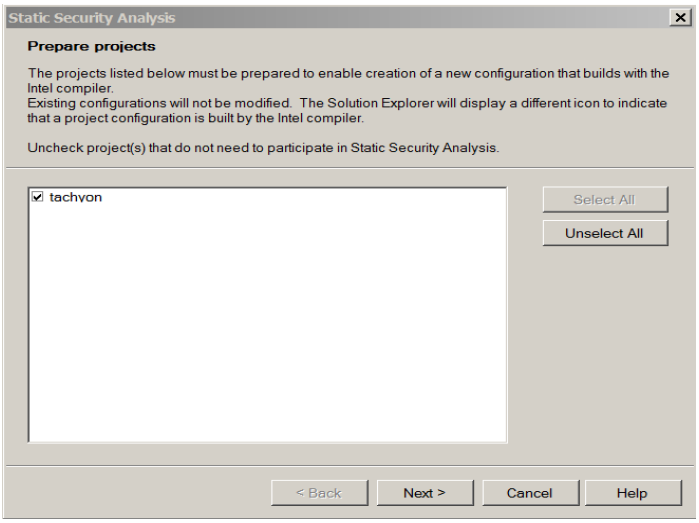
We will start by getting set up for SSA. This is done differently on Windows and Linux OS. For Windows OS read the following section. For Linux OS, go to Setting up and Running SSA Using a make File on Linux* OS.

Setting up and Running SSA Using Microsoft* Visual Studio* on Windows* OS

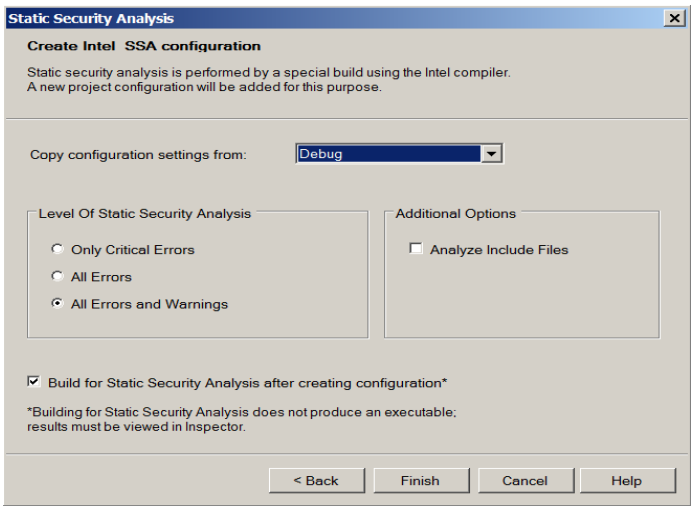
We will go through the setup process using Visual Studio solution tachyon.sln that is supplied with the sample program. Start by opening this solution in Visual Studio. The process for setting up for SSA is quite simple. If you are using Visual Studio* 2008 or Visual Studio 2010 then Visual Studio will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

Set up for SSA by selecting the **Build > Build Solution for Intel Static Security Analysis** menu item.

1. For Visual Studio 2005 and 2008 the 'Prepare projects' dialog appears as shown in the figure below.



2. Click on 'Next'. When we're finished, this will convert the project from Visual C++ format to Intel format. You can tell what projects are in Visual C++ versus Intel format by looking at their icon in the solution explorer. Intel format projects have the purple Composer icon. This conversion is done in such a way that existing configurations continue to build with the Visual C++ compiler. This step is not needed for Visual Studio 2010.
3. The 'Create configuration' dialog is used to choose an existing baseline configuration and specify the SSA options.



Click 'Finish' to accept the default settings. This creates a new Intel_SSA configuration. It copies the properties from the baseline (Debug) configuration and then changes it to 1) build with the Intel compiler and 2) enable the selected SSA-related options. Since the checkbox next to "Build for Static Security Analysis after creating configuration" is checked, it will immediately build this configuration, which is equivalent to running SSA.

That's it; you are now set up for SSA and have launched your first analysis.

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 and Running 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:

1. It builds using the Intel compiler
2. It adds the additional option "-diag-enable:sc3"
3. 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.

Now that you are set up, all you have to do is build your SSA build configuration to perform analysis. You can do this by performing the following steps:

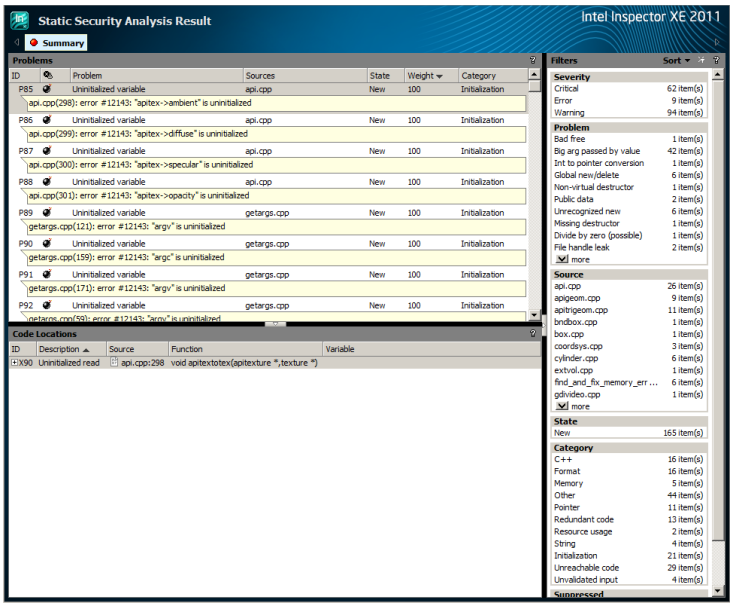
1. Open a command shell
2. Set the environment variables for the Intel compiler by executing the iclvars.sh script in the compiler bin directory, supplying the ia32 option.
3. Execute make –f tachyon_ssa.mk.

TIP: keep this command window open for later operations.

Interacting with the Analysis Results

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. 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.



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.

Let's start with an easy one. P66, the fourth one on the list, is an unsafe format specifier. Let's see what we can learn about this problem.

Investigating a Problem

Initially, all we know about the problem is summarized in the table entry:

P66	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"					

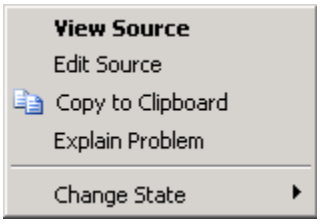
Unsafe format specifier is the short description of the problem. The full description is shown in the shaded area. The "New" entry in the state column indicates that this problem was discovered for the first time in this analysis run and has not yet been investigated. The 70 weight value indicates the 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	Variable
X69	Format mismatch	parse.cpp:187	unsigned int GetString(_iobuf *,char const *)	

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 "%*ddd*s" 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

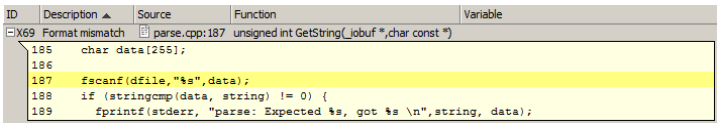
```
#include <stdio.h>

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;
}
```

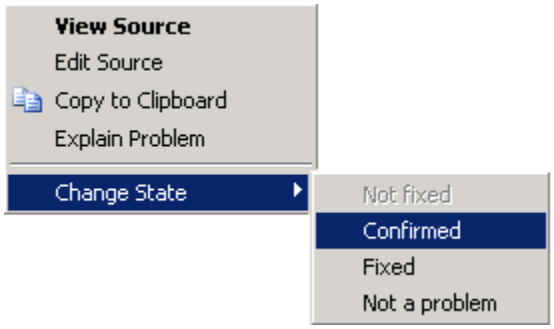

As you can see, the problem type reference material explains more fully what the precise error condition is and its potential consequences. It explains more fully the role the various code locations play in creating the problem, and provides an example that demonstrates the problem. Where possible, it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

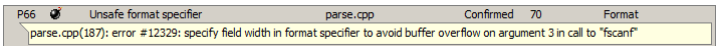


This is pretty clear. The highlighted call to fscanf has a format string with a "%s" format specifier. This reads input characters up to the next newline and stores the data in the array "data". There is no guarantee that the number of characters read will not overflow the bounds of the array, so this 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.



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



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 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.

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

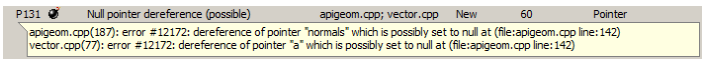
Notice how that problem we marked as "Confirmed" disappeared from the table of problem sets when we did that. It a good idea to keep this filter set like this while you work on analyzing results.

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 (apigeom.cpp). It will look like this:

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

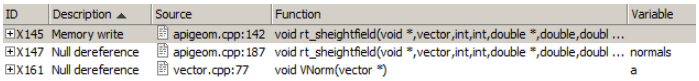
Take a look at another problem in the solution. This time pick problem P131, which should be the second one you now see in the table of problem sets:



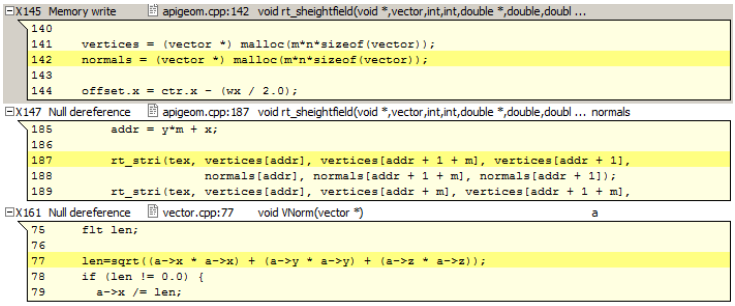
This problem is a little more interesting. As you can see, it has two source locations in different files. The problem type is "Null pointer dereference (possible)". The full description describes two places where a pointer is dereferenced that could possibly be set to null. In both cases the place where the pointer was possibly set to null is the same (apigeom.cpp, line 142).

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:



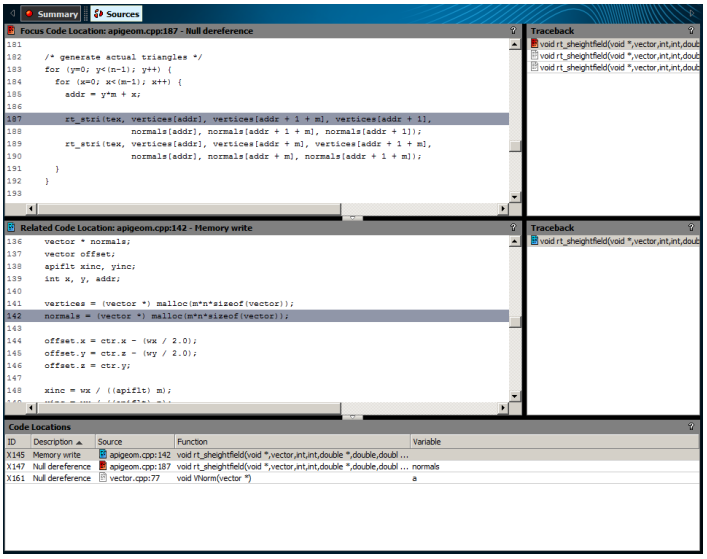
Here we see three code locations. One is the place where the pointer was assigned (Memory write) and the other two places are where a null pointer value could be dereferenced. Take a closer look by right-clicking one of these and selecting **Expand All Code Snippets** from the pop-up menu:



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"), 45 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"?

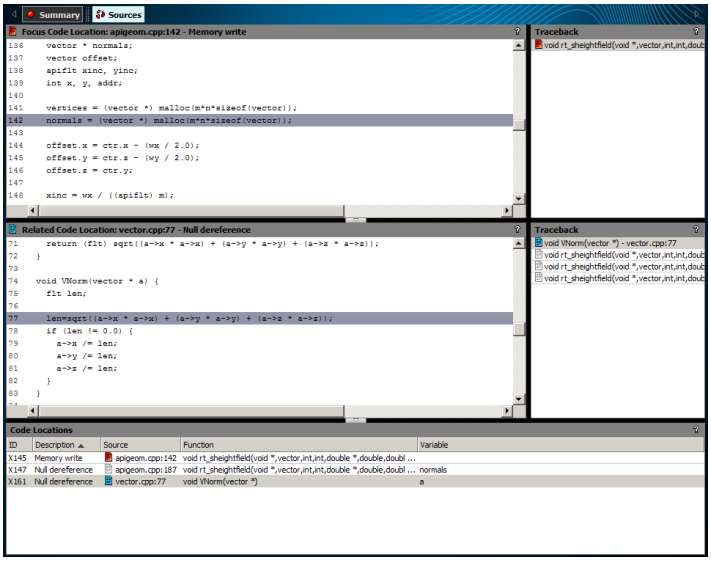
SSA helps you answer questions like this by providing "traceback" information. To see the traceback information, go to the Sources view. Right-click one of these code references and select **View Source** from the pop-up menu. This is what you will see:



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.

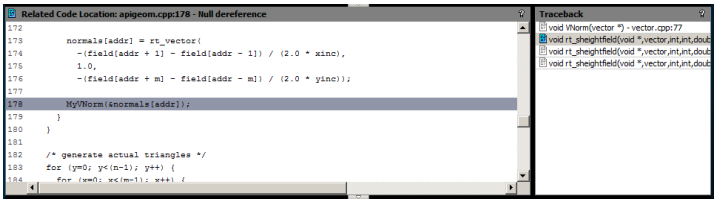
Neither of these code windows shows the source location we are interested in. However, down at the bottom is the same table of code locations we saw in the summary view. If you look closely you will see a little red tag on one, and a blue tag on another. This tells you which code locations are on display.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



Look to the right of the middle pane, at the windows that says **Traceback** on it. This contains four 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.



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 142 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));
```

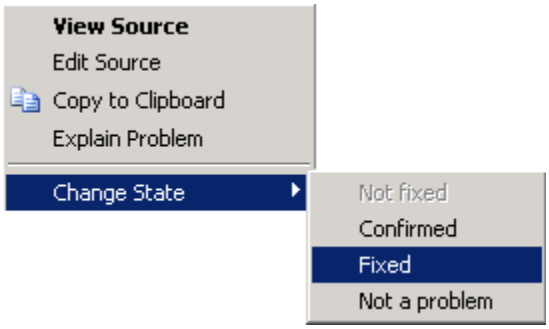
Rewrite this code like this:

```
  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.

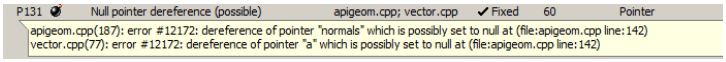
Now go back into the summary view. In Visual Studio, click on the **r000sc** tab to get back to the Intel Inspector XE GUI. Click on the box that says **Summary** at the upper left of the Sources view.

Look at the problem set table. There are actually two problems related to this code: P131 (the one we investigated), and P130, which is a similar problem with the line above it. In fact the fix we just typed in has actually corrected P130, but it did not fix P131. Still, go ahead and use the right-click pop-up menu to change the state of P131 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.

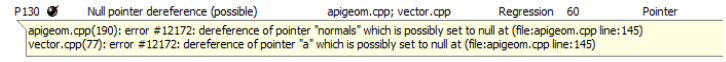


Now, go ahead and rebuild the application to create a new analysis result and open the new result, r001sc. On Windows OS, you can do all this by selecting the **Build>Rebuild solution for Static Security Analysis...** menu item. You will want to close the Visual Studio Output window when the build completes. 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 8 problems in r001sc (there were 9 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:



This is the problem that used to be number P131 in the old result. Now it is P130, 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.

Additional Resources

Intel Learning Lab – Technical videos, whitepapers, webinar replays and more.

Intel Parallel Studio XE product page – How to videos, getting started guides, documentation, product details, support and more.

Evaluation Guide Portal – Additional evaluation guides that show how to use various powerful capabilities.

Intel® Software Network Forums – A community for developers.

Intel® Software Products Knowledge Base – Access to information about products and licensing,

Download a free 30 day evaluation

Improve C++ Code Quality with Static Security Analysis (SSA)

Purchase Options: Language Specific Suites

Several suites are available combining the tools to build, verify and tune your application. All the suites listed below offer Static Security Analysis. Single or multi-user licenses and volume, academic, and student discounts are available.

	Suites >>	Intel® Parallel Studio XE	Intel® C++ Studio XE	Intel® Fortran Studio XE	Intel® Cluster Studio XE	Intel® Cluster Studio
Components	Intel® C / C++ Compiler	●	●		●	●
	Intel® Fortran Compiler	●		●	●	●
	Intel® Integrated Performance Primitives[3]	●	●		●	●
	Intel® Math Kernel Library[3]	●	●	●	●	●
	Intel® Cilk™ Plus	●	●		●	●
	Intel® Threading Building Blocks	●	●		●	●
	Intel® Inspector XE	●	●	●	●	
	Intel® VTune™ Amplifier XE	●	●	●	●	
	Static Security Analysis	●	●	●	●	
	Intel® MPI Library				●	●
	Intel® Trace Analyzer & Collector				●	●
	Rogue Wave IMSL* Library[2]					
	Operating System[1]	W, L	W, L	W, L	W, L	W, L

Note: (1)[1] Operating System: W=Windows, L= Linux, M= Mac OS* X. (2)[2] Available in Intel® Visual Fortran Composer XE for Windows with IMSL*
(3)[3] Not available individually on Mac OS X, it is included in Intel® C++ & Fortran Composer XE suites for Mac OS X

Notices

Optimization Notice

Notice revision #20110804

Intel's compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.