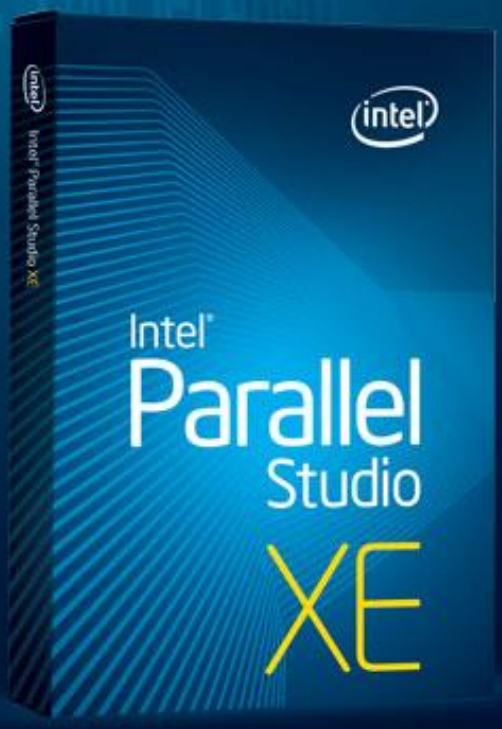
スタティック解析による C++ コード品質の向上



本ガイドは、スタティック解析 (SSA) により問題を排除し、C++ コードの安全性を高めて、ソフトウェアの品質を向上する方法について説明します。機能の概要を紹介し、エンドツーエンドの例を用いて使用方法を示します。

このスタティック解析は、インテル® Parallel Studio XE スイートでのみ利用可能です。

はじめに

本ガイドは、インテル® Parallel Studio XE のスタティック解析機能について説明する入門チュートリアルです。機能の概要を紹介し、エンドツーエンドの例を用いて使用方法を示します。

スタティック解析 (SSA) とは?

スタティック解析 (本ガイドでは SSA と表記します) は、ソースコードの詳細な解析を通じて、エラーとセキュリティーの脆弱性を発見します。SSA は、開発者と QA エンジニアが開発サイクルの初期にソフトウェアの問題を検出することで、時間とコストを節約し、投資収益率 (ROI) を改善できるように支援します。また、セキュリティー攻撃に対してアプリケーションを強化する手助けもします。SSA は不具合 (特にテストで完全に発見することが困難な不具合) を発見する効率的な方法を提供します。また、OpenMP* やインテル® Cilk™ Plus などの並列プログラミング・フレームワークの誤用による競合状態も検出します。

コーディング・エラーや安全ではない使用の多くのパターンは、スタティック解析で発見することが可能です。ダイナミック解析と比較して、スタティック解析の主な利点は、テスト中に発生するものだけではなく、可能性のあるすべての実行パスと変数値が検証されることです。通常、セキュリティー攻撃は予期しない方法やテストを実施していない方法で行われるため、この利点は、安全性の保証において特に重要でしょう。

SSA は 250 以上のエラー条件を検出することができます。検出されるエラーには、バッファー・オーバーフロー、境界違反、ポインターとヒープストレージの誤用、メモリーリーク、初期化されていない変数とオブジェクトの使用、C/C++ または Fortran 言語拡張とライブラリーの安全ではない使用/誤用などがあります。

SSA はどのように動作するか

SSA はインテル® C++ コンパイラーおよびインテル® Fortran コンパイラーの特別なモードで実行します。このモードでは、コンパイラーは解析により時間をかけ、命令生成プロセス全体を省略します。これにより、通常のコンパイルでは検知されないエラーを発見することができます。SSA を利用するには、インテル® コンパイラーで重大なエラーなしにコードをコンパイルできる必要がありますが、そのコンパイルの結果として作成されるバイナリーは実行しません。SSA を利用するために、インテル® コンパイラーを使用して製品版のバイナリーを作成する必要はありません。本ガイドでは、SSA を使用するためのアプリケーションの準備方法から説明します。

SSA は、プログラムやライブラリーの一部だけにも使用できますが、プログラム全体に使用することで最も効果を発揮します。各ファイルは 1 つのオブジェクト・モジュールにコンパイルされ、解析結果はリンク時に生成されます。そして、その結果はインテル® Inspector XE を使用して表示されます。

スタティック解析の結果は、多くの場合、決定的ではありません。ツールの結果は、調査に値する、潜在的な問題の最良の考察です。修正が必要かどうかを決定するのは開発者自身です。インテル® Inspector XE GUI は、このプロセスを円滑に行うために設計されています。報告された結果の各問題に対してステート (状態) を割り当てることで、解析結果を開発者が決定します。インテル® Inspector XE はそのステート情報を結果ファイルに保存します。

時間が経つにつれて、ソースも変更され、この解析を再度行うことがあるでしょう。インテル® Inspector XE で初めて新しい解析結果を開くと、以前の結果と新しい結果が自動的に対応します。この対応により、古い結果のステート情報が新しい結果に反映されます。つまり、これで同じ問題について再度調査する必要がなくなります。

修正の必要があると判断した問題については、テストで検出された不具合をレポートする場合と同じように、通常のバグ・トラッキング・システムにレポートしてください。インテル® Inspector XE はバグ・トラッキング・システムではありません。ステート情報は、解析結果の調査の進捗を追跡するものです。結果に対して開発者が何を行うかは、SSA やインテル® Parallel Studio XE の範囲外です。

SSA の設定

SSA の設定プロセスは通常、比較的簡単ですが、状況によっては非常に複雑になる場合もあります。Microsoft* Windows* で Microsoft* Visual Studio* を使用する場合は、メニュー/ダイアログから SSA 用のソリューション構成を自動的に作成できます。詳しい操作方法については、後述のチュートリアルで説明します。

Linux* または Windows* で makefile やコマンドライン・スクリプトを使用してビルドする場合は、SSA 用の新しいビルド構成を作成する必要があります。「ビルド構成」とは、特定のコンパイラー・オプションを指定し、中間ファイルを特定のディレクトリーに配置してアプリケーションをビルドするモードを指します。ほとんどのアプリケーションでは少なくとも、デバッグとリリースの 2 つの構成があります。もう 1 つを SSA 向けに作成してください。SSA 構成は、インテル® コンパイラーで SSA を有効にするオプションを指定してビルドする必要があります。

新しいビルド構成を作成することと、既存の構成を追加オプションでビルドすることは異なりますので、注意してください。例えば、デバッグ構成で SSA を有効にするオプションを使ってビルドすることで解析結果を得られます (インテル® コンパイラーでビルドした場合)。これは、最初の製品評価としては非常に良い方法です。しかし、この方法を継続的に行うことは不便でしょう。なぜなら、SSA で生成されるオブジェクト・モジュールは、SSA を実行するたびにデバッグモード・オブジェクト・モジュールを上書きするからです。デバッグ・オブジェクト・モジュールをリリース・オブジェクト・モジュールと分けておきたいのと同様に、SSA オブジェクト・モジュールも分けておくと良いでしょう。SSA を継続的に使用する場合は、このように設定を適切に行ってください。

新しいビルド構成を作成するプロセスは、各アプリケーションによって異なります。本ガイドで使用するサンプル・アプリケーションは、Microsoft* Windows* で Visual Studio* を使用するか、Linux* で makefile を使用してビルドできます。ここでは、この makefile を SSA 向けに更新する手順を示します。

アプリケーションのビルドが非常に複雑で、安全に変更できるか分からない場合は、別の設定方法があります。inspxe-inject と呼ばれる「ウォッチャー」ユーティリティーで通常のビルドを実行できます。このアプリケーションはプロセスの作成を中断して、ビルド中に行われたすべてのコンパイルとリンク段階を認識し、この情報をビルド仕様ファイルに記録します。このファイルは、別のユーティリティー inspxe-runsc の入力ファイルとして使用します。inspxe-runsc はインテル® コンパイラーを起動して、オリジナルのビルドと同じビルドステップを繰り返します。これらのユーティリティーについては、ここでは説明していません。

C++ チュートリアル

このチュートリアルは Windows* または Linux* で、C++ のサンプル・アプリケーションを使用します。Linux* と Windows* の違いはあまりないため、両者は一緒に説明しています。Windows* と Linux* で操作が異なる個所では、「Windows* では...」のように説明しています。

ここでは、「**tachyon_ssa**」というサンプル・アプリケーションを使用します。このサンプルは、インテル® Inspector XE のインストール・ディレクトリー以下の「**samples**」サブディレクトリーにあります。このサンプル・アプリケーションを展開してください。

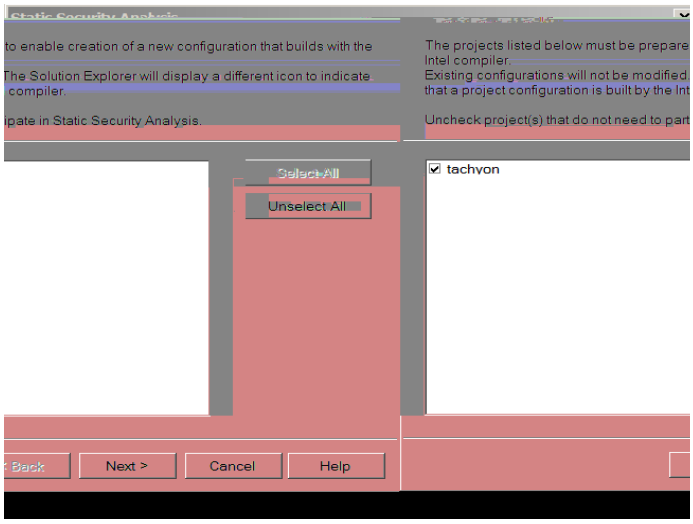
まず、SSA の設定から開始します。これは、Windows* と Linux* では操作が異なります。Windows* については、以下を参照してください。Linux* については、「Linux* で makefile を使用して SSA を設定する」へ進んでください。

Windows* で Microsoft* Visual Studio* を使用して SSA を設定する

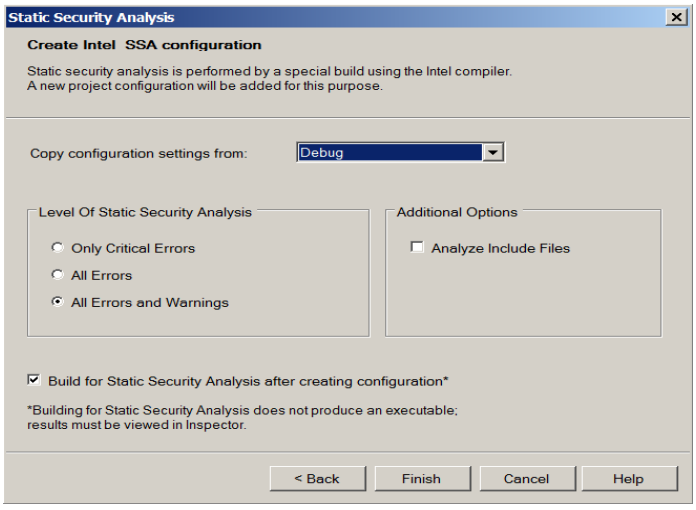
サンプル・アプリケーションに含まれている Visual Studio* ソリューション tachyon.sln を使用して設定を行います。まず、このソリューションを Visual Studio* で開きます。SSA の設定は簡単です。このソリューションは Visual Studio* 2008 形式なので、Visual Studio* 2010 以降を使用している場合は変換する必要があります。指示に従って、変換を行ってください。

SSA の設定を行うには、**[Build (ビルド)] > [Build solution for Intel Static Analysis (インテル・スタティック解析用にソリューションをビルド)]** メニューを選択します。

1. Visual **Studio*** 2008 の場合、以下の **[Prepare Projects (プロジェクトの準備)]** ダイアログ ボックスが表示されます。



2. **[Next (次へ)]** をクリックします。完了すると、プロジェクトが Visual C++* 形式からインテル形式に変換されます。Visual C++* 形式か、インテル形式かは、**[Solution Explorer (ソリューション エクスプローラ)]** のアイコンで判別できます。インテル形式のプロジェクトの場合、紫色の Composer アイコンが表示されます。この変換を行っても、既存の構成は引き続き Visual C++* コンパイラーでビルドできます。このステップは、Visual Studio* 2010 以降では必要ありません。
3. **[Create configuration (構成を作成)]** ダイアログで、必要に応じて、既存のベースライン構成を選択し、SSA オプションを指定します。



このチュートリアルでは、デフォルト設定のまま **[Finish (完了)]** をクリックします。すると、ベースラインの (Debug) 構成からプロパティーがコピーされ、1) インテル® コンパイラーでビルドし、2) 選択した SSA 関連のオプションを有効にするように変更された新しい Intel_SSA 構成が作成されます。そして、**[Build for Static Analysis after creating configuration (構成を作成後にスタティック解析用にビルド)]** チェックボックスがオンになっていたため、この構成のビルドが直ちに実行されます (つまり、SSA が実行されます)。

以上で SSA の設定と最初の解析は終了です。

コマンドライン・スクリプトや makefile を使用してアプリケーションのビルドを設定する方法については、次に説明する、Linux* での設定手順のセクションを参照してください。

Linux* で makefile を使用して SSA を設定する

サンプル・アプリケーションには Linux* でアプリケーションをビルドするのに使用できる makefile が含まれています。この makefile は、2 つのバージョンが用意されています。1.つはオリジナルの makefile である **tachyon.mk** です。この makefile は gcc* コンパイラーを使用してアプリケーションをビルドします。もう 1 つはアップデートされた makefile、**tachyon_ssa.mk** で、新しいビルドターゲット SSA が追加されています。SSA ビルドターゲットは、次の変更点を除いて、デバッグ・ビルド・ターゲットのようなものです。

1. インテル® コンパイラーを使用してビルドする
2. 追加オプション「-diag-enable:sc3」が含まれている
3. 中間ファイルを SSA サブディレクトリーに置く

2 つの makefile を比較して、違いを確認してみましょう。SSA 用に必要な makefile ファイルへの変更点が分かります。

設定が終わったので、あとは SSA をビルド構成をビルドし、解析を行うだけです。次の操作を行います。

1. コマンドシェルを開きます。
2. インテル® コンパイラーの bin ディレクトリー以下にある iclvars.sh スクリプトを ia32 オプションを付けて実行し、インテル® コンパイラーの環境変数を設定します。
3. make -f tachyon_ssa.mk を実行します。

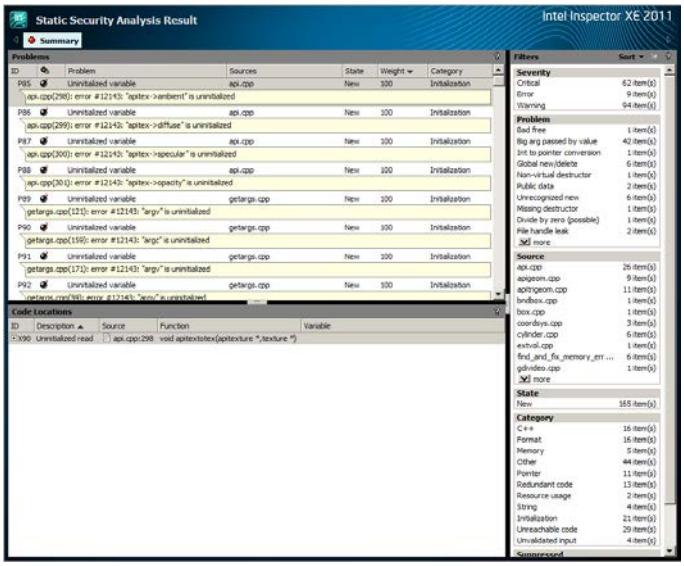
ヒント: 後の操作で使用できるように、このコマンドウィンドウを開いたままにしておきます。

解析結果の確認

Windows* ではビルドが終了すると、新しい結果がインテル® Inspector XE で自動的に開きます。Linux* では `inspxe-gui` と入力してインテル® Inspector XE を起動し、**[File (ファイル)]** > **[Open (開く)]** メニューから結果を開きます。デフォルトでは、このファイルは「`r000sc.inspxe`」という名前になります。これは、tachyon プロジェクトのルート・ディレクトリー以下の r000sc ディレクトリーにあります。

このチュートリアルの残りの内容は、Windows* と Linux* でほとんど同じです。主な違いは、Windows* ではインテル® Inspector XE GUI が Visual Studio* に統合されますが、Linux* の GUI は、スタンドアロンのプログラムとして実行されます。インテル® Inspector XE の各ウィンドウの外観も Windows* と Linux* でほとんど同じです。

最初に表示されるウィンドウは次のようになります。



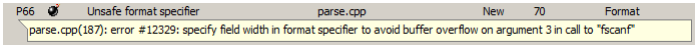
このウィンドウは、3 つの主なエリアで構成されています。左上のペインは **[Problem Sets (問題セット)]** の表です。これが「やること」リスト、つまり調査が必要なものです。左下のペインは、現在選択されている問題セットに対応するコードの場所を示しています。右のペインはフィルターを示しています。どの問題セットを表示/非表示にするかをこのフィルターで制御します。

問題セットの表は、列ヘッダーをクリックして、ソートすることができます。デフォルトでは、**[weight (ウェイト)]** 順にソートされます。ウェイトとは 1 から 100 までの値で、問題のインパクトの程度を表しています。ダメージが大きい可能性があるほど、大きなウェイトの値になります。また、(誤診ではなく) 実際に問題になりそうなものほど高い値になります。つまり、ウェイトは検証する順序のためのガイダンスを提供します。

まずは、簡単なものから開始しましょう。リストの 4 番目、P66 は安全ではない書式指定子です。この問題から何が得られるか見てみましょう。

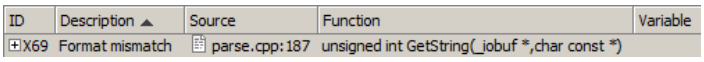
問題の検証

最初に、この問題について分かっていることは、表エントリーにまとめられています。



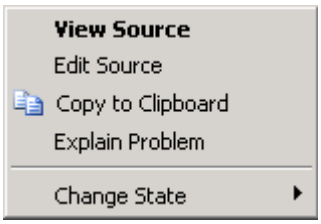
「**Unsafe format specifier (安全でない書式指定子)**」がこの問題の内容で、説明はその下のシェードエリアに表示されています。ステート列の「New」エントリーは、この問題がこの解析で初めて発見され、まだ調査されていなことを示しています。70 がウェイトの値です。問題のカテゴリーは「Format」です。これは、printf 形式の書式指定子の誤用に関連していることを意味します。

この問題をクリックして選択します。下のペインは更新され、この問題に関連するコードの場所が表示されます。

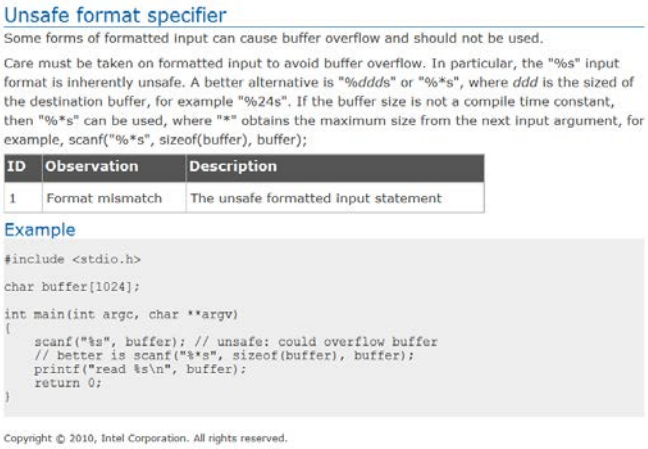


上記のようにソース個所が表示されます (parse.cpp ファイル、行 187)。このソース個所が問題を引き起こしていることが分かります。「**Format mismatch (書式の不一致)**」は、書式文字列が使用された場所を示します。その行 (GetString) と引数シグネチャーが含まれる関数名も分かります。

問題に関してより多くの情報を得る 1 つの方法は、問題タイプが何かをチェックすることです。いくつかの SSA エラーは非常にテクニカルで、詳細情報が必要です。この問題タイプの詳細情報を見るには、問題を右クリックして、次のようなポップアップ・メニューを表示します。



[Explain Problem (問題の説明)] を選択して、この問題の詳細が説明されているヘルプトピックを開きます。



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

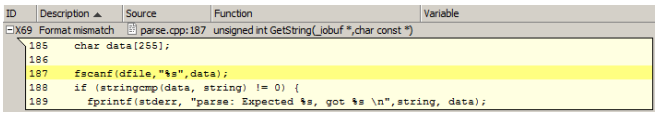
```
#include <stdio.h>

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;
}
```

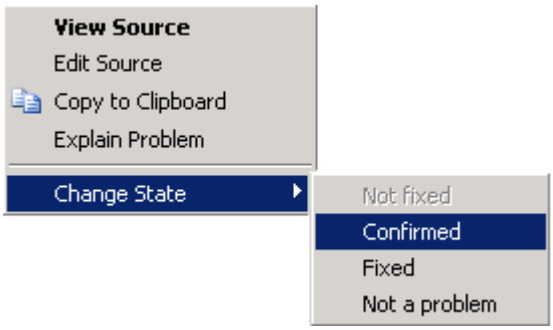

このように、問題タイプのリファレンスでは正確なエラー状態と潜在的な結果について詳しく説明しています。問題を引き起こす一因となるさまざまなコードの関与についてさらに詳細情報を示し、また問題を示すサンプルも提供しています。可能であれば、この問題を回避するより良い代替案も示します。

次に、この問題が本当にアプリケーションに存在しているかどうかを判断しなければなりません。そのためには、ソースコードを確認します。最も早い方法は、下のペインにあるコード参照を展開して、該当する場所の小さな断片を表示します。これを行うには、ID 列のプラスサインをクリックするか、下のペインのアイテムを右クリックしてポップアップ・メニューから **[Expand All Code Snippets (すべてのコード片を展開)]** を選択します。すると、以下のような画面が表示されます。

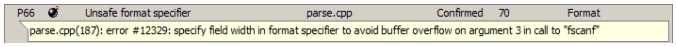


これでソースの該当箇所がよく分かります。ハイライトされた fscanf への呼び出しに、「%s」書式指定子による書式文字列があります。これは、入力文字列を次の行まで読み込み、そのデータを配列「data」に格納します。読み込まれる文字数が配列境界をオーバーフローしないという保証はありません。そのため、このステートメントはメモリーを壊す可能性があります。幸いなことに、この問題で知らなければならないことがすべてこのコード片に含まれています。ソースをさらに表示する方法については後述します。

この問題は (誤診ではなく) 本当のエラーであることが確認できたので、結論を記録しましょう。問題を右クリックし、ポップアップ・メニューから **[Change State (ステートを変更)] > [Confirmed (確認済み)]** を選択します。



これで、この問題については終了です。問題セットの表のステートが更新されます。



フィルターで整理

フィルターを使って、開発者は関心のある問題に注目し、無視する問題は非表示にできます。

いったん問題が調査されたら、通常は、それを再度確認する必要はありません。調査が終了したすべての問題を非表示にできるのがフィルターの良い点です。フィルターウィンドウの下部にある **[Investigated (調査済み)]** フィルターの **[Not investigated (未調査)]** をクリックします。

すると、フィルター項目が更新され、アクティブであることが示されます。

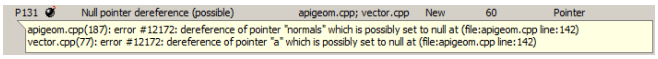
ここで、**[Confirmed (確認済み)]** とマークした問題が、問題セットの表からなくなりました。解析結果の調査中は、このようにフィルターをセットしておくと良いでしょう。

最初の Severity、Problem、Source、State、Category フィルターは問題セットの表の列と対応しています。列の特定の値と一致しない表の行はすべて非表示にすることができます。Source フィルター の 2 行目 (apigeom.cpp) をクリックすると、次のように表示されます。

問題セットの表も更新され、このソースファイルの問題のみが表示されます。**[All (すべて)]** ボックスをクリックして、フィルターをオフにすることもできますが、ここではオンのままにしておきます。

2 つ目の問題の調査

次に、ソリューションの別の問題を見てみましょう。今度は、問題セットの表の P131 を選びます。



これは興味深い問題です。上記で分かるように、2 つのソースの場所が別々のファイルにあります。問題タイプは「**Null pointer dereference (possible) (NULL ポインターの逆参照 (可能性あり)**」です。詳細説明では、NULL である可能性のあるポインターの逆参照が設定されている 2 つの場所を言及しています。ポインターが NULL に設定されている可能性のある場所は、両方のケースで同じです (apigeom.cpp、行 142)。

ここで、SSA は 2 つの関連する問題を 1 つの問題セットにまとめています。これは、両方とも同じソリューションで解決する可能性が高いためです。このように問題がまとめられることから、この表は、問題の表ではなく、問題セットの表と呼びます。

ここで起こっている問題について調査してみましょう。前回と同様、この問題セットを選択して、下のペインで対応するソースコードを開きます。

ID	Description	Source	Function	Variable
⊞X145	Memory write	apigeom.cpp:142	void rt_sheightfield(void *,vector,int,int,double *,double,doubl ...	
⊞X147	Null dereference	apigeom.cpp:187	void rt_sheightfield(void *,vector,int,int,double *,double,doubl ...	normals
⊞X161	Null dereference	vector.cpp:77	void VNorm(vector *)	a

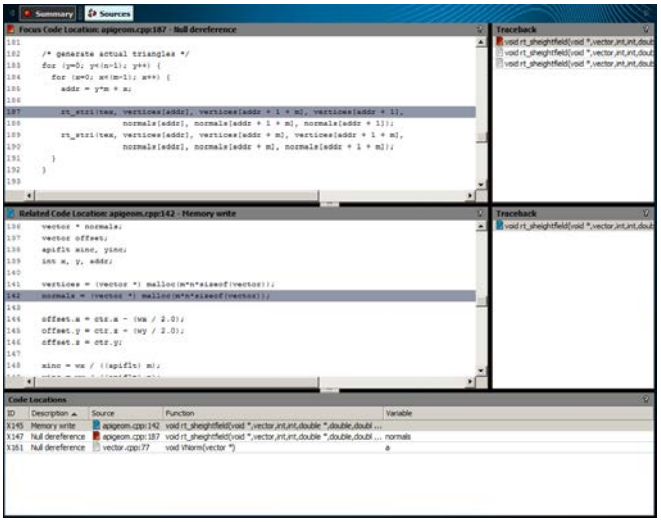
上記のように、3 つのコード個所があります。1 つはポインターが割り当てられた個所 (Memory write)、他の 2 つは NULL ポインター値が逆参照された可能性のある個所です。3 つのいずれかを右クリックし、ポップアップ・メニューから **[Expand All Code Snippets (すべてのコード片を展開)]** を選択してさらに詳しく見てみましょう。

```
⊟X145 Memory write    apigeom.cpp:142  void rt_sheightfield(void *,vector,int,int,double *,double,doubl ...
  140
  141     vertices = (vector *) malloc(m*n*sizeof(vector));
  142     normals = (vector *) malloc(m*n*sizeof(vector));
  143
  144     offset.x = ctr.x - (wx / 2.0);
⊟X147 Null dereference    apigeom.cpp:187  void rt_sheightfield(void *,vector,int,int,double *,double,doubl ... normals
  185         addr = y*m + x;
  186
  187         rt_stri(tex, vertices[addr], vertices[addr + 1 + m], vertices[addr + 1],
  188                 normals[addr], normals[addr + 1 + m], normals[addr + 1]);
  189         rt_stri(tex, vertices[addr], vertices[addr + m], vertices[addr + 1 + m],
⊟X161 Null dereference    vector.cpp:77  void VNorm(vector *)  a
   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;
```

これで、問題はさらに明らかになります。メモリーへの書き込みは、ルーチン「malloc」からの値を割り当てました。アプリケーションがメモリー不足になると、malloc は NULL ポインターを返します。このアプリケーションは明らかにポインターを使用する前にこのようなケースのチェックを行っていません。2 つ目の断片は、同じポインター変数であるように見えるもの (「normals」) (同じファイルのサブルーチンで割り当て後の 45 行下) を使っていることが分かります。3 つ目の断片はどうでしょう? これはポインター (「a」) を使用していることが分かりますが、normals ポインター変数とどのように関連しているのでしょうか? さらに、これは全く別のソースファイル (vector.cpp) です。malloc 呼び出しとどのような関係にあるのでしょうか?

ここで、SSA はプログラム全体のクロスファイル解析を行うことを思い出してください。SSA は、ファイル間をまたがっていても、プロシージャー・コールを通じたデータ値の流れを解析できます。しかし、malloc から受け取った値がどのようにポインター「a」になったのでしょうか?

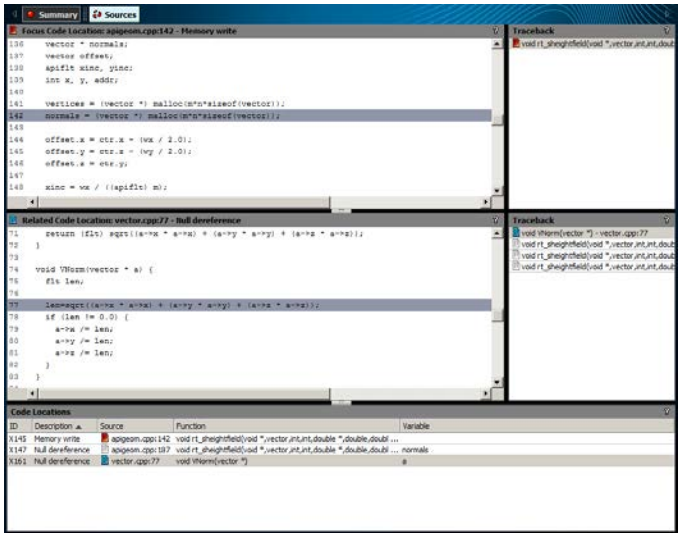
SSA は、この疑問への解決を支援するべく、「traceback (トレースバック)」情報を提供します。トレースバック情報を表示するには、**[Sources (ソース)]** ビューを開きます。**[Sources (ソース)]** ビューは、コード参照のいずれかを右クリックし、ポップアップ・メニューから **[View Source (ソースの表示)]** を選択すると表示されます。



これが **[Sources (ソース)]** ビューです。2 ペアのウィンドウがあり、それぞれ、ソースコードのセクションと、右側に **[Traceback (トレースバック)]** が表示されます。左上に **[Sources (ソース)]** ボックスがあり、その左に **[Summary (サマリー)]** ボックスがあります。**[Summary (サマリー)]** ボックスをクリックすると、**[Summary (サマリー)]** ビューに戻ります。

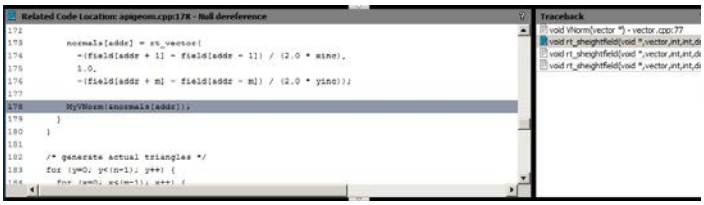
どちらのコードウィンドウも、ここで見るべきソース個所を示していません。しかし、下部には、サマリービューで見たものと同じ Code Locations (コード場所) の表があります。注意して見てみると、ここに小さな赤と青のタグがあることに気付くでしょう。これは、表示されているコード場所を示しています。同じ赤と青のタグが、左上のコードウィンドウに表示されています。

vector.cpp のコードの場所に注目してみましょう。これをダブルクリックしてください。これで、コード参照のソースが開きます。右クリックして、ポップアップ・メニューから **[Set as Related Observation (関連観測結果として設定)]** または **[Set as Focus Observation (重点観測結果として設定)]** を選択することもできます。赤のタグは重点観測結果で、青のタグは関連観測結果です。以下のような画面が表示されます。



右側の **[Traceback (トレースバック)]** ウィンドウに表示されている 4 行をそれぞれクリックしてみてください。左のコードウィンドウが更新され、異なるソースの位置が示されます。トレースバックの最後のものは、実際、トップの画面で表示されるソース個所 (malloc が呼び出された場所) と同じです。

トレースバックは、調査を開始した個所 (malloc) と終了した個所 (「a」の NULL 逆参照の可能性) の関連性を示しています。ここで注目すべきは、トレースバックの 2 行目です。



ここで、MyVNorm への呼び出しがあります。逆参照が発生した個所を再度確認すると、VNorm というサブルーチン内にあることが分かります。MyVNorm と VNorm は同じようには見えませんが、実際は同じです。それを証明するのが次の行です。

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

この呼び出しは、これら 2 つを結び付けています。プログラムは `rt_sheightfield` というルーチンから開始して、ストレージのメモリー割り当てを行い、それを `normals` というポインターに割り当てます。そして、`VNorm` を呼び出し、`normals` (実際は `&normals[addr]` ですが、これは NULL にもなり得ます) を渡しました。`VNorm` のコードを再度確認すると、「a」が仮引数の名前であることが分かります。これが、malloc の値が「a」になった経緯です。

問題を修正してリスキャン

次に、この問題を修正します。この問題を修正するには、malloc への呼び出しの後にコードを追加して、NULL の結果をテストする必要があります。結果が NULL の場合は、エラー回復処理を行います。このケースでは、問題のサブルーチンから単に戻るだけでエラーを回復できます。

ソースの修正は、ソースエディターを使って、apigeom.cpp の malloc 呼び出し直後の行 142 を変更します。**[Sources (ソース)]** ビューの行をダブルクリックするか、行を右クリックしてポップアップ・メニューから **[Edit Source (ソースの編集)]** を選択し、通常のソースエディターを開くことができます。Linux* では、EDITOR 環境変数で指定されているエディターが開きます。Windows* では Visual Studio* ソースエディターが開きます。

Windows* では、エディターウィンドウは、結果が含まれる同じタブ・グループ・ウィンドウで新しいタブに開きます。ソースを編集用に開くと、結果は隠れます。結果を再度表示するには、r000sc タブをクリックします。

大きなディスプレイをお使いであれば、結果とソースファイルを並べて表示しても良いでしょう。

エディターで次のコードを確認します。

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

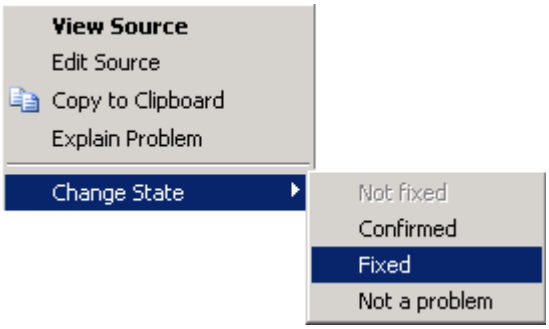
このコードを以下のように書き換えます。

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

if からの 3 行をコピーして、ソースに貼り付けてください。そして、保存し、エディターを閉じます。ただし、まだアプリケーションはリビルドしないでください。実際、この変更は問題を解決するわけではありませんが、まずは変更を行います。

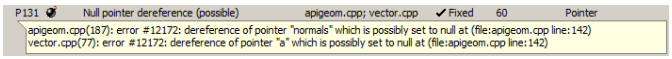
ここで、**[Summary (サマリー)]** ビューに戻りましょう。Visual Studio* で **r000sc** タブをクリックし、インテル® Inspector XE GUI へ戻ります。**[Sources (ソース)]** ビューの左上にある **[Summary (サマリー)]** ボックスをクリックします。

問題セットの表を見てください。このコードに関連する問題が 2 つあります。P131 (調査済みのもの) と P130 (その上の行と類似する問題) です。実際、上記で入力した修正は P130 を修正していますが、P131 は直していません。それでも、ポップアップ・メニューを右クリックして、P131 のステートを **[Fixed (修正済み)]** に変更してください。



[Fixed (修正済み)] に変更するとすぐに、この問題は表示されなくなります。これは、未調査の問題のみを表示するようにフィルターを設定しているためです。この問題を再度表示するには、**[Filters (フィルター)]** ペインの下部で **[All (すべて)]** をクリックしてフィルターを変更してください。

これで、再度この問題が表示され、**[Fixed (修正済み)]** とマークされていることが確認できます。

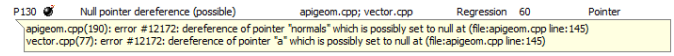


ここで、アプリケーションをリビルドし、新しい解析結果、r001sc を生成して開いてみましょう。Windows* では、**[Build (ビルド)] > [Rebuild solution for Intel Static Analysis (インテル・スタティック解析用にソリューションをリビルド)]** を選択するだけです。ビルドが終了したら、Visual Studio* 出力ウィンドウを閉じてかまいません。Linux* では、前に行ったビルド手順を繰り返します (コマンドウィンドウで `make –f “tachyon_ssa.mk”` と入力し、インテル® Inspector XE の GUI で **[File (ファイル)] > [Open (開く)]** を選択して **r001sc** を開きます)。

ここで、ほとんどの問題が「New」ではなく、「Not fixed」となっていることに気付くでしょう。これは、インテル® Inspector XE により、以前の結果 r000sc のステートが新しい結果 r001sc に自動的に反映されるためです。いったん表示された問題は「New」ではなくなり、まだ調査はされていないので、「Not fixed」ステートになります。

また、前に調査した問題 （安全でない書式問題） が r000sc でマークしたように、「Confirmed」ステートになっています。

フィルターを使って apigeom.cpp ファイルにある問題のみを選択してください。r001sc には 8 つの問題があることが分かります (r000sc では 9 つでした)。つまり、ソースを変更したことにより問題が 1 つ解決されたことを示しています。リストのトップにある問題を見てください。次のような行が表示されています。



これは、以前の結果で P131 だった問題で、今は P130 になっています。しかし、r000sc で 「Fixed」とマークしたものと同じ問題です。インテル® Inspector XE により、これが同じ問題であることが判断され、まだ問題が存在するため、**[Fixed (修正済み)]** ではなく、**[Regression (リグレッション)]** に設定されました。これは、行った変更が問題を解決したわけではないことを示しています。**[Regression (リグレッション)]** は未調査のステートと見なされるため、フィルターを **[Not investigated (未調査)]** ステートのみの表示に設定すると、この問題は表示されます。.

サマリーとレビュー

これまで行ったことを振り返ってみましょう。

1) 問題セットの表は、SSA で見つかった問題をまとめています。
2) 問題セットを選択すると、関連するコードの場所が下のペインに表示されます。
3) コード片を表示したり、(トラックバックを確認できる) **[Sources (ソース)]** ビューを開いたり、ソースエディターを開いたりして、問題を掘り下げます。
4) **[Explain Problem (問題の説明)]** メニューで問題タイプのリファレンスを表示し、問題タイプの意味について詳細説明を確認します。
5) 問題のステートを設定して、調査の結果を記録します。
6) フィルターを使用して、調査の終わった問題を非表示にしたり、特定のソースファイルや問題のクラスに注目します。
7) SSA は、調査時間を短縮するため、関連する問題を問題セットにまとめることがあります。
8) SSA はステート情報を 1 つの結果から別の結果へと自動的に維持します。新規の問題、調査され修正が確認された問題も記録します。

関連情報

ラーニングラボ - テクニカルビデオ、ホワイトペーパー、Webinar など

インテル® Parallel Studio XE 製品ページ - HOW TO ビデオ、入門ガイド、ドキュメント、製品の詳細情報、サポートなど

評価ガイド - さまざまな機能の使用法を紹介する評価ガイド

インテル® ソフトウェア・ネットワーク・フォーラム - デベロッパー・コミュニティー

インテル® ソフトウェア製品ナレッジベース - 製品およびライセンスに関する情報

30 日間の評価版のダウンロード

開発者向け日本語最新技術情報: http://www.isus.jp/

スタティック解析による C++ コード品質の向上

購入方法:言語別のスイート

アプリケーションをビルド、検証、チューニングする複数のツールが組み合わされた次のスイートがご利用になれます。本資料で説明している製品は青でハイライトされています。ライセンスは、シングルユーザー・ライセンス、フローティング・ライセンス、アカデミック・ライセンスが用意されています。

	スイート>>	インテル® Cluster Studio XE	インテル® Parallel Studio XE	インテル® C++ Studio XE	インテル® Fortran Studio XE	インテル® Composer XE	インテル® C++ Composer XE	インテル® Fortran Composer XE
コンポーネント	インテル® C/C++ コンパイラー	●	●	●		●	●	
	インテル® Fortran コンパイラー	●	●		●	●		●
	インテル® IPP	●	●	●		●	●	
	インテル® MKL	●	●	●	●	●	●	●
	インテル® Cilk™ Plus	●	●	●		●	●	
	インテル® TBB	●	●	●		●	●	
	インテル® Inspector XE	●	●	●	●			
	インテル® VTune™ Amplifier XE	●	●	●	●			
	インテル® Advisor XE	●	●	●	●			
	スタティック解析	●	●	●	●			
	インテル® MPI ライブラリー	●						
	インテル® Trace Analyzer & Collector	●						
	Rogue Wave IMSL* ライブラリー[2]							●
	オペレーティング・システム [1]	W、L	W、L	W、L	W、L	W、L	W、L、O	W、L、O

注: [1] オペレーティング・システム:W = Windows*、L = Linux*、O = OS X*
[2] インテル® Visual Fortran Composer XE Windows* 版 IMSL* 同梱で利用可能

インテル® Parallel Studio XE の詳細:

- 以下の Web サイトをご覧ください。 http://intel.ly/parallel-studio-xe
- あるいは、左の QR コードをスキャンしてください。

30 日間の評価版:

- http://intel.ly/sw-tools-eval の Web サイトで、「Product Suites」をクリックしてください。

著作権、商標、注意事項について

最適化に関する注意事項 改訂 #20110804

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。これには、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項の適用対象である特定の命令セットに関する詳細は、該当する製品のユーザー・リファレンス・ガイドを参照してください。