Oracle® Solaris Studio 12.4: コードアナライザユーザーズガイド

Part No: E57230
2014 年 10 月

ORACLE

目次

このドキュメントの使用法

- **概要** – コードアナライザツールを使用してデータを分析および表示する方法を説明します。
- **対象読者** - アプリケーション開発者、システム開発者、アーキテクト、サポートエンジニア
- **必要な知識** - プログラミングの経験、ソフトウェア開発テスト、ソフトウェア製品の構築およびコンパイルの経験

製品ドキュメントライブラリ

この製品の最新情報や既知の問題は、ドキュメントライブラリ (http://docs.oracle.com/cd/E37069_01) に含まれています。

Oracle サポートへのアクセス

Oracle のお客様は、My Oracle Support にアクセスして電子サポートを受けることができます。詳細は、http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info を参照してください。聴覚に障害をお持ちの場合は、http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs を参照してください。

フィードバック

このドキュメントに関するフィードバックを http://www.oracle.com/goto/docfeedback からお聞かせください。

♦ ♦ ♦ 第 1 章

1 概要

Oracle Solaris Studio コードアナライザは、Oracle Solaris 向けの C および C++ アプリケーションの開発者を支援するための統合ツールセットであり、セキュア、堅牢、および高品質なソフトウェアを作成できます。

この章には、次の情報が含まれます。

コードアナライザで分析されるデータ

コードアナライザは 3 種類のデータを分析します。

- コンパイル時に検出される静的コードエラー
- メモリーエラー検出ツールである `discover` ユーティリティーで検出される動的メモリーアクセスエラーおよび警告
- コードカバレージツールである `uncover` ユーティリティーで測定されるコードカバレージデータ

個々の分析へのアクセスを提供するほかに、コードアナライザは静的コード検査と動的メモリーアクセス分析およびコードカバレージ分析を統合して、単独で機能するほかのエラー検出ツールでは検出できない多くの重要なエラーをアプリケーションで検出できます。

また、コードアナライザはコード内の中核となる問題、つまり、それらを修正すればほかの問題も解消される可能性の高い問題を特定します。通常、中核となる問題にはほかのいくつかの問

題が関連しています。たとえば、それらの問題では割り当てポイントが共通であったり、問題が同じ関数の同じデータアドレスで発生したりするためです。

静的コード検査

静的コード検査は、コード内の一般的なプログラミングエラーをコンパイル時に検出します。C および C++ コンパイラの `-xprevise=yes` オプションは、コンパイラの制御およびデータフロー分析フレームワークを活用して、アプリケーションのプログラミングおよびセキュリティー上の潜在的な欠陥を分析します。

注記 - オプションで、`-xanalyze=code` オプションを使用して静的コードエラーを収集できますが、このオプションは EOL です。`-xprevise=yes` オプションを使用することを推奨します。

静的エラーデータの収集については、13 ページの「静的エラーデータの収集」を参照してください。

コードアナライザで分析される静的コードエラーのリストについては、23 ページの「静的コードの問題」を参照してください。

動的メモリーアクセス検査

多くの場合、コード内のメモリー関連のエラーは見つけることが困難です。プログラムを実行する前に `discover` で計測機構を組み込むと、プログラムの実行中に `discover` はメモリーアクセスエラーを動的に検出して報告します。たとえば、プログラムが配列を割り当て、それを初期化せずに、配列内のある場所から読み取ろうとする場合、プログラムは動作が不安定になることがあります。プログラムに Discover で計測機構を組み込んでから実行すると、`discover` はこのエラーを検出します。

動的メモリーアクセスエラーデータの収集については、14 ページの「動的メモリーアクセスデータの収集」を参照してください。

コードアナライザで分析される動的メモリーアクセスの問題のリストについては、29 ページの「動的メモリーアクセスエラー」を参照してください。

コードカバレージ検査

コードカバレージは、テストで実行されるコード領域、および実行されないコード領域に関する情報を提供するので、より多くのコードをテストできるようにテストスイートを改善できます。コードアナライザは、`uncover` によって収集されたデータを使用して、プログラム内のどの関数がカバーされていないか、また、該当する関数をカバーするテストを追加した場合にアプリケーションの合計カバレージが何パーセント増加するかを調べます。

コードカバレージデータの収集については、15 ページの「コードカバレージデータの収集」を参照してください。

コードアナライザを使用するための要件

コードアナライザは、Oracle Solaris Studio 12.3 または 12.4 の C または C++ コンパイラでコンパイルされたバイナリから収集される、静的エラーデータ、動的メモリーアクセスエラーデータ、およびコードカバレージデータを処理します。

コードアナライザは、Solaris 10 10/08 以上のオペレーティングシステム、Oracle Solaris 11、Oracle Enterprise Linux 5.x、または Oracle Enterprise Linux 6.x 以上を実行する SPARC ベースまたは x86 ベースのシステムで実行されます。

コードアナライザ GUI

コンパイラ、Discover、または Uncover でデータを収集したあと、`code-analyzer` コマンドを発行してコードアナライザ GUI を起動し、問題の表示と分析を行うことができます。

コードアナライザは各問題について、問題の説明、問題が見つかったソースファイルのパス名、およびそのファイルの該当するソース行を強調表示したコードスニペットを表示します。

コードアナライザでは次の操作を実行できます。

- 問題の詳細を表示する。静的な問題の場合、詳細にはエラーパスが含まれます。動的メモリーアクセスの問題の場合、詳細には呼び出しスタックが含まれ、データが使用可能であれば割り当てスタックと解放スタックも含まれます。
- 問題が見つかったソースファイルを開く。
- エラーパスまたはスタック内の関数呼び出しから関連するソースコード行に移動する。

- プログラムでの関数の使用箇所をすべて見つける。
- 関数宣言に移動する。
- オーバーライドされるまたはオーバーライドする関数の宣言に移動する。
- 関数のコールグラフを表示する。
- 問題の種類ごとに、コード例や考えられる原因などの詳細を表示する。
- 表示する問題を分析の種類、問題の種類、およびソースファイルによってフィルタリングする。
- すでに確認した問題を非表示にし、関心のない問題を閉じる。

GUI の使用に関する詳細は、GUI のオンラインヘルプと『Oracle Solaris Studio 12.4: コードアナライザチュートリアル 』を参照してください。

コードアナライザのコマンド行インタフェース

コードアナライザのコマンド行インタフェースバージョンである `codean` は、静的コード検査、Discover、および Uncover を使用して、Analytics ファイルを入力として読み取り、テキストまたは HTML 形式で出力を生成します。また、あとで履歴データと新しいデータを比較できるように、履歴アーカイブにデータを保管するメカニズムを提供します。`codean` では、次の操作を実行できます。

- API 形式でのレポートの読み取りと、テキストおよび HTML 形式への情報の変換。`codean` はテキスト出力を *.type*`.html` ファイル (*type* は `static`、`dynamic`、または `coverage`) に保存します。
- `.analyze/`*type*`/latest` レポートでは、各問題のチェックサムが計算され、元の問題情報が `.analyze/history//`*type* ファイル (*type* は `static`、`dynamic`、または`coverage`) に保管されます。
- 最新レポートに新しい問題または修正された問題だけを表示し、以前に保存されているレポートと比較する。
- 収集するデータの種類 (`dynamic`、`static`、`coverage`、または `all`) を指定する。
- フルパス名を表示する。
- 特定のソースファイルの問題を表示する。
- 特定の行数のソースコードを表示する。
- 最新のレポートを保存します。

- 同じタグ名を使用して最後に保存されたレポートを上書きする。
- 新しい問題または修正された問題だけをレポートに表示する。
- レポートを保存するディレクトリを指定する。
- 表示するエラーと警告の種類をフィルタリングする。

詳細については、codean(1) のマニュアルページを参照してください。

リモートデスクトップ配布

ほとんどすべてのオペレーティングシステム上で動作し、リモートサーバー上の Oracle Solaris Studio のコンパイラやツールを使用するコードアナライザのリモートデスクトップ配布を作成できます。インストール時にリモートデスクトップ配布を生成して、「デフォルトのユーザーディレクトリからユーザー設定をエクスポートする」オプションをチェックすると、コードアナライザは、配布の生成元となったサーバーをリモートホストとして認識し、Oracle Solaris Studio インストール内のツールコレクションにアクセスします。デフォルトでは、このオプションは選択されません。

リモートオペレーティングシステムでコードアナライザを起動するには、該当する実行可能ファイルを実行します。

```
./codeanalyzer/bin/codeanalyzer.exe
```

リモートデスクトップ配布のインストール方法については、『Oracle Solaris Studio: 12.4 インストールガイド 』を参照してください。

リモートデスクトップ配布については、コードアナライザ GUI オンラインヘルプを参照してください。

クイックスタート

次に、コードに関する情報を収集するために必要な手順の例と、コードアナライザで結果を表示する方法を、サンプル C プログラムを使用して示します。

▼ クイックスタート

1. **プログラムをコンパイルして静的データを収集します。**

```
% cc -xprevise=yes *.c
```

注記 - 以前は -xanalyze=code オプションを使用してコンパイルできました。このオプションは Oracle Solaris Studio 12.4 でも有効ですが、EOL です。

2. **デバッグ情報付きでプログラムを再コンパイルします。**

```
% cc -g *.c
```

3. **プログラムに `discover` で計測機構を組み込んでからプログラムを実行し、動的メモリーアクセスデータを収集します。**

```
% cp a.out a.out.save
% discover -a a.out
% a.out
```

4. **プログラムに `uncover` で計測機構を組み込んでから、コードカバレージデータを収集します。**

```
% a.out
% cp a.out.save a.out
% a.out
% uncover a.out
```

5. **情報の収集後に、収集データを表示するためにコードアナライザを GUI または `codean` コマンド行ツールで使用するよう選択できます。**

   - **GUI でコードアナライザにアクセスするには、次のコマンドを使用します。**

     ```
     % code-analyzer a.out
     ```

   - **コマンド行ツールでコードアナライザにアクセスするには、次のコマンドを使用します。**

     ```
     % codean a.out
     ```

♦ ♦ ♦ 第 2 章

2 データの収集とコードアナライザの起動

コードアナライザでの分析のために収集されるデータは、ソースコードファイルが入っているディレクトリの *binary-name*.analyze ディレクトリに保存されます。*binary-name*.analyze ディレクトリは、コンパイラ、discover、または uncover により作成されます。

この章には、次のトピックに関する情報が含まれています。

静的エラーデータの収集

C または C++ プログラムの静的エラーデータは、Oracle Solaris Studio 12.3 または 12.4 の C または C++ コンパイラを使用して -xprevise=yes オプションでプログラムをコンパイルします。以前は -xanalyze=code オプションを使用していましたが、このオプションは EOL であるため、代わりに -xprevise=yes オプションを使用することをお勧めします。-xprevise=yes オプションは、以前のリリースの Oracle Solaris Studio ではコンパイラで使用できません。このオプションを使用すると、コンパイラは静的エラーを自動的に抽出し、データを *binary-name*.analyze ディレクトリの static サブディレクトリに書き込みます。

プログラムを -xprevise=yes オプションでコンパイルしたあと、別の手順でリンクする場合は、リンク手順でも -xanalyze=code オプションを指定する必要があります。

Linux では、静的エラーデータを収集するには -xannotate オプションと -xprevise=yes を指定する必要があります。例:

```
% cc -xprevise=yes -xannotate -g t.c
```

コンパイラはコード内の静的エラーをすべて検出できるわけではないことに注意してください。

- 実行時にのみ使用可能になるデータに依存するエラーもあります。たとえば、次のコードの場合、ファイルから読み取られる `ix` の値が [0,9] の範囲外にあることを検出できないため、コンパイラは ABW (配列範囲外への書き込み) エラーを検出しません。

```
void f(int fd, int array[10])
{
  int ix;
  read(fd, &ix, sizeof(ix));
  array[ix] = 0;
}
```

- 一部のエラーはあいまいであり、実際のエラーではないことがあります。コンパイラはこのようなエラーを報告しません。
- 一部の複雑なエラーは、このリリースのコンパイラでは検出されません。

静的エラーデータを収集したあと、コードアナライザ GUI またはコマンド行ツール (`codean`) を起動してデータの分析と表示を行なったり、動的メモリーアクセスまたはコードカバレージのデータを収集できるようにするためにプログラムを再度コンパイルできます。

動的メモリーアクセスデータの収集

C または C++ プログラムの動的メモリーアクセスデータの収集は、`discover` でバイナリに計測機構を組み込む手順と、計測機構付きバイナリを実行する手順の 2 つから成ります。

コードアナライザ用のデータを収集するために `discover` でプログラムに計測機構を組み込むには、プログラムを Oracle Solaris Studio バージョン 12.3 または 12.4 の C または C++ コンパイラでコンパイルしておく必要があります。`-g` オプションでコンパイルするとデバッグ情報が生成され、動的メモリーアクセスエラーおよび警告に関するソースコードおよび行番号情報をコードアナライザで表示できるようになります。

プログラムが最適化なしでコンパイルされている場合に、`discover` はソースコードレベルでもっとも完全なメモリーエラー検出を提供します。最適化を使用してコンパイルした場合、一部のメモリーエラーは検出されません。

Discover で計測機構を組み込むことのできる、またはできないバイナリの種類については、『Oracle Solaris Studio 12.4: Discover および Uncover ユーザーズガイド 』の「バイ

ナリを適切に準備する」および『Oracle Solaris Studio 12.4: Discover および Uncover ユーザーズガイド 』の「プリロードまたは監査を使用するバイナリは互換性がない」を参照してください。

注記 - `discover` と `uncover` の両方で使用するためのプログラムを 1 回で構築できます。ただし、すでに計測機構の付いたバイナリに計測機構を組み込むことはできないため、`uncover` を使用してカバレージデータも収集する予定であれば、`discover`.でバイナリに計測機構を組み込む前に uncover 用のコピーを保存してください。例:

```
% cp a.out a.out.save
```

▼ バイナリから動的メモリーアクセスデータを収集する方法:

1. **Discover で -a オプションを使用してバイナリに計測機構を組み込みます。**

   ```
   % discover -a binary_name
   ```

   注記 - Oracle Solaris Studio バージョン 12.3 または 12.4 のバージョンの Discover を使用する必要があります。以前のバージョンの `discover` では `-a` オプションは使用できません。

   ---

2. **計測機構付きバイナリを実行します。**

   動的メモリーアクセスデータが、*binary_name*`.analyze` ディレクトリの `dynamic` サブディレクトリに書き込まれます。

   注記 - `discover` でバイナリに計測機構を組み込むときに指定できるその他の計測オプションについては、『Oracle Solaris Studio 12.4: Discover および Uncover ユーザーズガイド 』の「計測オプション」または `discover` のマニュアルページを参照してください。

   ---

3. **(オプション) コードアナライザの GUI またはコマンド行ツール (`codean`) を起動し、以前に収集した静的コードデータを使用してデータを分析および表示します。または、計測機構の付いていないバイナリのコピーを使用して、コードカバレージデータを収集できます。**

コードカバレージデータの収集

C または C++ プログラムのコードカバレージデータの収集は、3 段階のプロセスです。

1. `uncover` でバイナリに計測機構を組み込みます。

2. 計測済みバイナリの実行
3. uncover を再度実行してコードアナライザが使用するカバレージレポートを生成します。

バイナリに計測機構を組み込んだあと、その計測機構付きバイナリを複数回実行してすべての実行に関するデータを蓄積してから、カバレージレポートを生成できます。

▼ バイナリからコードカバレージデータを収集する方法

始める前に コードアナライザで使用するデータを収集するために uncover でプログラムに計測機構を組み込むには、プログラムを Oracle Solaris Studio バージョン 12.3 または 12.4 の C または C++ コンパイラでコンパイルしておく必要があります。-g オプションでコンパイルするとデバッグ情報が生成され、ソースコードレベルのカバレージ情報をコードアナライザで使用できるようになります。

注記 - discover で計測機構を組み込むためにプログラムをコンパイルしたときに、バイナリのコピーを保存した場合は、そのコピーの名前を元のバイナリ名に変更し、それを使用して uncover で計測機構を組み込むことができます。例:

```
cp a.out.save a.out
```

---

1. **Uncover でバイナリに計測機構を組み込みます。**

   ```
   % uncover binary-name
   ```

2. **計測機構付きバイナリを 1 回以上実行します。**

   コードカバレージデータが、*binary-name*.uc ディレクトリに書き込まれます。

3. **Uncover で -a オプションを使用して、蓄積されたデータからコードカバレージレポートを生成します。**

   ```
   % uncover -a  binary-name.uc
   ```

   カバレージレポートが、*binary-name*.analyze ディレクトリの coverage サブディレクトリに書き込まれます。

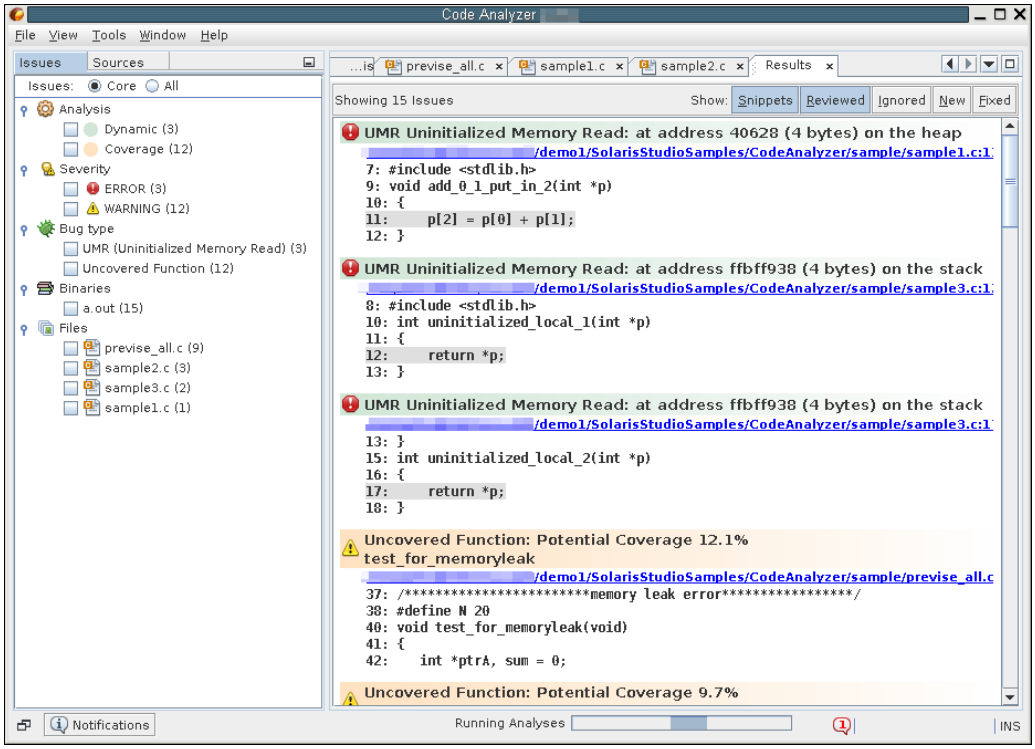
注記 - Oracle Solaris Studio バージョン 12.3 または 12.4 のバージョンの uncover を使用する必要があります。以前のバージョンの uncoverでは -a オプションは使用できません。

コードアナライザ GUI の使用

コードアナライザ GUI を使用して、さまざまな種類のデータを分析できます。GUI を起動するには、`code-analyzer` コマンドと、収集したエラーデータを分析するバイナリへのパスを入力します。

```
% code-analyzer binary-name
```

次の図に示すように、コードアナライザ GUI は、*binary-name*`.analyze` ディレクトリのデータを開き、表示します。



コードアナライザ GUI の実行中に、「開く」->「ファイル」を選択して別のバイナリを指定すると、そのバイナリについて収集されたデータの表示に切り替えることができます。

GUI のオンラインヘルプには、表示する結果のフィルタリング、問題の表示/非表示、および特定の問題に関する詳細の表示を行う、すべての機能の使用方法が説明されています。『Oracle Solaris Studio 12.4: コードアナライザチュートリアル 』では、サンプルプログラムを使用して、データの収集と分析のシナリオ全体を説明します。

コードアナライザコマンド行ツール (codean) の使用

コードアナライザコマンド行ツール codean を使用して、3 種類までのデータを分析できます。codean を起動するには、codean コマンド、オプション、および実行可能ファイルまたはディレクトリのパスを入力します。

codean *options executable-path|directory*

codean ツールは、画面にテキスト出力を表示します。実行可能ファイルと同じ場所にある .*type*.html ファイルで結果を確認することもできます。このセクションでは、コマンドオプションについて説明します。

codean のオプション

以降のセクションでは、codean に使用できる各種オプションについて説明します。

データの種類のオプション

次のオプションは、収集するデータの種類を指定します。

-s　　静的データを処理および表示します。

-d　　動的データを処理および表示します。

-c　　カバレージデータを処理および表示します。

複数のオプションを指定するか、オプションを指定しないでおくことができます。何も選択しないと、デフォルトで、.analyze/*type*/latest ファイル (*type* は static、dynamic、coverage のいずれか) が存在しているかどうかに応じて、すべてのオプションが処理されます。

表示オプション

次のオプションは、結果のテキスト出力の内容を決定します。

--fullpath　　ファイルのフルパス名を表示します。

-f *source-file*　　指定したソースファイルに関する問題だけを表示します。

-n *number*　　指定された行数のソースコードを表示します。

フィルタリングオプション

次のオプションは、結果に報告されるエラーと警告の種類を決定します。

エラーまたは警告の種類は、次のいずれかです。

- 3 文字のエラーコードまたは 3 文字の警告コード。すべてのエラーと警告のリストについては、付録A コードアナライザで分析されるエラーを参照してください。
- MLK または mlk (メモリーリーク)。
- ALL または all (すべての警告またはエラー)。

エラーまたは警告が指定されていない場合、デフォルトは all です。

フィルタリングオプションは次のとおりです。

--showerrors *error-type*　　指定された種類のエラーだけを表示します。

--showwarnings *warning-type*　　指定された種類の警告だけを表示します。

--hideerrors *error-type*　　指定された種類のエラーを表示しません。

--hidewarnings *warning-type*　　指定された種類の警告を表示しません。

結果保存オプション

最新の結果をファイルに保存し、このファイルに特定のタグ名を付けて特定のディレクトリに配置できます。

--save　　最新のレポートを保存します。

--tag *tag-name*　　--save と組み合わせて使用する場合は、保存されるコピーの名前としてタグ名 *tag-name* が使用されます。保存されているコピーに同じタグ名が

使用されている場合、codean は警告メッセージを発行し、ファイルを上書きせずに終了します。タグ名が指定されていない場合、codean は実行可能ファイルの最新レポートの最終変更時間をチェックし、そのタイムスタンプをタグ名として使用します。

-t　同じタグ名で保存されているレポートを上書きします。

-D *directory*　レポートをディレクトリ *directory* に保存します。

結果比較オプション

次のオプションを使用して、以前に生成されたレポートと結果を比較できます。

--whatisnew　新しい問題だけを表示します。このオプションは --whatisfixed と一緒に使用できません。

--whatisfixed　修正された問題だけを表示します。このオプションは --whatisnew と一緒に使用できません。

--tag *tag-name*　--whatisnew または --whatisfixed と組み合わせて使用すると、新しく生成されたレポートとの比較に、タグ名 *tag-name* が付いているレポートの履歴コピーが使用されます。タグ名が指定されていない場合、最新レポートが最終保存コピーと比較されます。

--ref *file|directory*　--whatisnew または --whatisfixed と組み合わせ、後にパス名を指定する必要があります。このオプションは、新しいレポートと比較するファイルまたはディレクトリを指定します。

codean ワークフローの例

このセクションでは、バグ修正の効果をモニターする例を説明します。

1. 修正の前にターゲットソースをコンパイルします。

```
% cc -g *.c
```

2. Discover を使用してバイナリに計測機構を組み込み、Analytics 出力が生成されるようにします。

```
% discover -a a.out
```

3. 計測されたバイナリを実行します。

4. codean を使用して Analytics 出力を格納します。履歴アーカイブが a.out.analyze/history/before_bugfix に作成され、dynamic という名前の履歴ファイルがこのディレクトリ内に作成されます。

   ```
   % codean --save --tag before_bugfix -d a.out
   ```

5. バグを修正します。
6. ターゲットソースをもう一度コンパイルします。

   ```
   % cc -g *.c
   ```

7. discover を使用して再度バイナリに計測機構を組み込みます。

   ```
   % discover -a a.out
   ```

8. 計測されたバイナリを実行します。

   ```
   % a.out
   ```

9. 比較結果を表示し、バグが原因で発生していた無効なメモリーアクセスが修正されたことを確認します。

   ```
   % codean --whatisfixed --tag before_bugfix -d a.out
   ```

   これにより、修正された動的な問題だけが含まれる新しい Analytics 出力ファイルが a.out.analyze/dynamic/fixed_before_bugfix に作成されます。修正された問題を表示するには、codeanまたはコードアナライザ GUI を使用できます。

10. (オプション) codean を実行して、新しいバグが発生していないことを確認します。

    ```
    % codean --whatisnew --tag before_bugfix -d a.out
    ```

    このコマンドにより、新しい動的な問題だけが含まれる新しい Analytics ファイルが a.out.analyze/dynamic/new_before_bugfix に作成されます。

♦ ♦ ♦ 付 録 A

コードアナライザで分析されるエラー

コンパイラ、`discover`、および `uncover` は、コード内の静的コードの問題、動的メモリーアクセスの問題、およびカバレージの問題を検出します。この付録では、これらのツールで検出され、コードアナライザで分析される特有のエラーの種類について説明します。

コードカバレージの問題

コードカバレージ検査では、カバーされていない関数が特定されます。結果では、見つかったコードカバレージの問題に「カバーされていない関数」というラベルが付けられ、潜在的なカバレージの割合が示されます。この割合は、該当する関数をカバーするテストを追加した場合にアプリケーションの合計カバレージが何パーセント増加するかを示しています。

考えられる原因: 関数を実行するテストが行われなかったか、デッドコードまたは古いコードを削除していません。

静的コードの問題

静的コード検査では、次の種類のエラーが検出されます。

- ABR: 配列境界を越える読み取り (beyond array bounds read)
- ABW: 配列境界を越える書き込み (beyond array bounds write)
- DFM: メモリーの二重解放 (double freeing memory)

- ECV: 明示的型キャスト違反 (explicit type cast violation)
- FMR: 解放済みメモリーの読み取り (freed memory read)
- FMW: 解放済みメモリーの書き込み (freed memory write)
- INF: 空の無限ループ (infinite empty loop)
- MLK: メモリーリーク (memory leak)
- MFR: 関数の復帰なし (missing function return)
- MRC: malloc 戻り値の検査なし (missing malloc return value check)
- NFR: 初期化されていない関数の復帰 (uninitialized function return)
- NUL: NULL ポインタ間接参照、リークの可能性があるポインタの検査
- RFM: 解放済みメモリーを返す (return freed memory)
- UMR: 初期化されていないメモリーの読み取り、初期化されていないメモリーの読み取りビット操作 (uninitialized memory read, uninitialized memory read bit operation)
- URV: 使用されていない戻り値 (unused return value)
- VES: スコープ外での局所変数の使用 (out-of-scope local variable usage)

このセクションでは、エラーの考えられる原因と、エラーが発生する可能性があるコードの例を説明します。

配列境界を越える読み取り (ABR)

考えられる原因: 配列境界を越えてメモリーを読み取ろうとしました。

例:

```
int a[5];
. . .
printf("a[5] = %d\n",a[5]);  // Reading memory beyond array bounds
```

配列境界を越える書き込み (ABW)

考えられる原因: 配列境界を越えてメモリーに書き込もうとしました。

例:

```
int a [5];
```

```
. . .
a[5] = 5; // Writing to memory beyond array bounds
```

メモリーの二重解放 (DFM)

考えられる原因: 同じポインタを使用して `free()()` を複数回呼び出しました。C++ では、同じポインタに対して `delete` 演算子を 2 回以上使用しています。

例:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .       // p was not signed a new value between the free statements
 free(p); // Double freeing memory
```

解放済みメモリーの読み取り (FMR)

例:

```
int *p = (int*) malloc(sizeof(int));
free(p);
. . .  // Nothing assigned to p in between
printf("p = 0x%h\n",p); // Reading from freed memory
```

解放済みメモリーの書き込み (FMW)

例:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .        // Nothing assigned to p in between
 *p = 1; // Writing to freed memory
```

空の無限ループ (INF)

例:

```
int x=0;
int i=0;
while (i200) {
  x++; } // Infinite loop
```

メモリーリーク

考えられる原因: メモリーが割り当てられるが、関数の終了またはエスケープの前に解放されていません。

例:

```
int foo()
{
 int *p = (int*) malloc(sizeof(int));
 if (x) {
  p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
  }
}                                     // The 2nd malloc leaked here
```

関数の復帰なし (MFR)

考えられる原因: 終了するパスの一部に戻り値がありません。

例:

```
#include <stdio.h>
int foo (int a, int b)
{
  if (a)
    {
       return b;
    }
}            // If foo returns here, the return is uninitialized
int main ( )
{
    printf("%d\n", foo(0,30));
}
```

malloc 戻り値の検査なし (MRC)

考えられる原因: Cの `malloc` または C++ の new 演算子からの戻り値に `null` 検査なしでアクセスします。

例:

```
#include <stdlib.h>
int main()
{
 int *p3 = (int*) malloc(sizeof(int)); // Missing null-pointer check after malloc.
 *p3 = 0;
```

```
}
```

リークの可能性があるポインタの検査: NULL ポインタ間接参照 (NUL)

考えられる原因: `null` に等しくなる可能性のあるポインタにアクセスしているか、null になることのないポインタに冗長な `null` 検査を行なっています。

例:

```
#include <stdio.h>
#include <stdlib.h>
int gp, ctl;
int main()
{
 int *p = gp;
 if (ctl)
  p = 0;
 printf ("%c\n", *p); // May be null pointer dereference
 if (!p)
  *p = 0; // Surely null pointer dereference

int *p2 = gp;
*p2 = 0; // Access before checking against NULL.
assert (p2!=0);

int *p3 = gp;
if (p3) {
  printf ("p3 is not zero.\n");
}
*p3 = 0; // Access is not protected by previous check against NULL.
}
```

解放済みメモリーを返す (RFM)

例:

```
#include <stdlib.h>
int *foo ()
{
 int *p = (int*) malloc(sizeof(int));
 free(p);
 return p; // Return freed memory is dangerous
}
int main()
{
 int *p = foo();
```

```
 *p = 0;
}
```

初期化されていないメモリーの読み取り (UMR)

考えられる原因: 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
#include <stdio.h>
#include <stdlib.h>
 struct ttt {
    int a: 1;
    int b: 1;
 };

 int main()
 {
   int *p = (int*) malloc(sizeof(int));
   printf("*p = %d\n",*p); // Accessing uninitialized data

   struct ttt t;
   extern void foo (struct ttt *);

   t.a = 1;
   foo (&t); // Access uninitialized bitfield data "t.b"
 }
```

使用されていない戻り値 (URV)

考えられる原因: 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
 int foo();
int main()
{
   foo(); // Return value is not used.
}
```

スコープ外での局所変数の使用 (VES)

考えられる原因: 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
int main()
{
   int *p = (int *)0;
   void bar (int *);
   {
     int a[10];
     p = a;
   } // local variable 'a' leaked out
   bar(p);
 }
```

動的メモリーアクセスエラー

動的メモリーアクセス検査では、次の種類のエラーが検出されます。

- ABR: 配列境界を越える読み取り (beyond array bounds read)
- ABW: 配列境界を越える書き込み (beyond array bounds write)
- BFM: 不正な空きメモリー (bad free memory)
- BRP: 不正な realloc アドレスパラメータ (bad realloc address parameter)
- CGB: 破損したガードブロック (corrupted guard block)
- DFM: メモリーの二重解放 (double freeing memory)
- FMR: 解放済みメモリーの読み取り (freed memory read)
- FMW: 解放済みメモリーの書き込み (freed memory write)
- FRP: 解放済み realloc パラメータ (freed realloc parameter)
- IMR: 無効なメモリーの読み取り (invalid memory read)
- IMW: 無効なメモリーの書き込み (invalid memory write)
- MLK: メモリーリーク (memory leak)
- OLP: 送り側と受け側の重複 (overlapping source and destination)
- PIR: 部分的に初期化された読み取り (partially initialized read)
- SBR: スタック境界を越える読み取り (beyond stack bounds read)
- SBW: スタック境界を越える書き込み (beyond stack bounds write)
- UAR: 割り当てられていないメモリーの読み取り (unallocated memory read)
- UAW: 割り当てられていないメモリーの書き込み (unallocated memory write)
- UMR: 初期化されていないメモリーの読み取り (uninitialized memory read)

このセクションでは、エラーの考えられる原因と、エラーが発生する可能性があるコードの例を説明します。

配列境界を越える読み取り (ABR)

考えられる原因: 配列境界を越えてメモリーを読み取ろうとしました。

例:

```
int a[5];
. . .
printf("a[5] = %d\n",a[5]);  // Reading memory beyond array bounds
```

配列境界を越える書き込み (ABW)

考えられる原因: 配列境界を越えてメモリーに書き込もうとしました。

例:

```
int a [5];
 . . .
 a[5] = 5; // Writing to memory beyond array bounds
```

不正な空きメモリー (BFM)

考えられる原因: `free()()` または `realloc()()` にヒープデータ以外のポインタを渡しました。

例:

```
#include <stdlib.h>
int main()
{
 int *p = (int*) malloc(sizeof(int));
 free(p+1); // Freeing wrong memory block
}
```

不正な Realloc アドレスパラメータ (BRP)

例:

```
#include <stdlib.h>
int main()
{
 int *p = (int*) realloc(0,sizeof(int));
 int *q = (int*) realloc(p+20,sizeof(int[2])); // Bad address parameter for realloc
}
```

破損したガードブロック (CGB)

考えられる原因: 動的に割り当てられた配列の末尾を越えて「レッドゾーン」に書き込んでいます。

例:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
 int *p = (int *) malloc(sizeof(int)*4);
 *(p+5) = 10; //  Corrupted array guard block detected (only when the code is not
annotated)
 free(p);

 return 0;
}
```

メモリーの二重解放 (DFM)

考えられる原因: 同じポインタを使用して `free()()` を複数回呼び出しました。C++ では、同じポインタに対して `delete` 演算子を 2 回以上使用しています。

例:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .       // p was not assigned a new value between the free statements
 free(p); // Double freeing memory
```

解放済みメモリーの読み取り (FMR)

例:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .  // Nothing assigned to p in between
 printf("p = 0x%h\n",p); // Reading from freed memory
```

解放済みメモリーの書き込み (FMW)

例:

```
int *p = (int*) malloc(sizeof(int));
 free(p);
 . . .        // Nothing assigned to p in between
 *p = 1; // Writing to freed memory
```

解放済み Realloc パラメータ (FRP)

例:

```
#include <stdlib.h>

int main() {
  int *p = (int *) malloc(sizeof(int));
  free(0);
  int *q = (int*) realloc(p,sizeof(it[2])); //Freed pointer passed to realloc
}
```

無効なメモリーの読み取り (IMR)

考えられる原因: ハーフワード境界、ワード境界、またはダブルワード境界に整列していないアドレスから、それぞれ 2、4、または 8 バイトを読み取っています。

例:

```
#include <stdlib.h>
int main()
{
  int *p = 0;
  int i = *p;   // Read from invalid memory address
}
```

無効なメモリーの書き込み (IMW)

考えられる原因: ハーフワード境界、ワード境界、またはダブルワード境界に整列していないアドレスに、それぞれ 2、4、または 8 バイトを書き込んでいます。テキストアドレスに書き込んでいるか、読み取り専用データセクション (`.rodata`) に書き込んでいるか、`mmap` によって読み取り専用にされているページに書き込んでいます。

例:

```
  int main()
{
  int *p = 0;
```

```
  *p = 1;  // Write to invalid memory address
}
```

メモリーリーク

考えられる原因: メモリーが割り当てられるが、関数の終了またはエスケープの前に解放されていません。

例:

```
    int foo()
{
  int *p = (int*) malloc(sizeof(int));
  if (x) {
   p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
  }
}                                     // The 2nd malloc leaked here
```

送り側と受け側の重複 (OLP)

考えられる原因: 正しくないソース、宛先、または長さが指定されました。ソースと宛先が重複している場合、プログラムの動作は不定になります。

例:

```
#include <stlib.h>
#include <string.h>
int main() {
  char *s=(char *) malloc(15);
  memset(s, 'x', 15);
  memcpy(s, s+5, 10);
  return 0;
}
```

部分的に初期化された読み取り (PIR)

例:

```
 #include <stdio.h>
#include <stdlib.h>
 int main()
{
  int *p = (int*) malloc(sizeof(int));
  *((char*)p) = 'c';
```

```
  printf("*(p = %d\n",*(p+1)); // Accessing partially initialized data
}
```

スタック境界を越える読み取り (SBR)

考えられる原因: ローカル配列の末尾よりあと、または先頭より前を読み取っています。

例:

```
 #include <stdio.h>

int main() {
  int a[2] = {0, 1};
  printf("a[-10]=%d\n",a[-10]); //  Read is beyond stack frame bounds

  return 0;
}
```

スタック境界を越える書き込み (SBW)

考えられる原因: ローカル配列の末尾よりあと、または先頭より前に書き込んでいます。

例:

```
 #include <stdio.h>

 int main() {
  int a[2] = {0, 1};
  a[-10] = 2; //  Write is beyond stack frame bounds

  return 0;
}
```

割り当てられていないメモリーの読み取り (UAR)

考えられる原因: ストレイポインタ (不正な値を持つポインタ)、ヒープブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例:

```
 #include <stdio.h>
 #include <stdlib>
int main()
{
```

```
  int *p = (int*) malloc(sizeof(int));
  printf("*(p+1) = %d\n",*(p+1)); // Reading from unallocated memory
}
```

割り当てられていないメモリーの書き込み (UAW)

考えられる原因: ストレイポインタ (不正な値を持つポインタ)、ヒープブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例:

```
#include <stdio.h>
#include <stdlib>
int main()
{
 int *p = (int*) malloc(sizeof(int));
  *(p+1) = 1; // Writing to unallocated memory
}
```

初期化されていないメモリーの読み取り (UMR)

考えられる原因: 初期化されていないローカルデータまたはヒープデータの読み取り。

例:

```
#include <stdio.h>
#include <stdlib>
int main()
{
 int *p = (int*) malloc(sizeof(int));
 printf("*p = %d\n",*p); // Accessing uninitialized data
}
```

動的メモリーアクセスの警告

動的メモリーアクセス検査では、次の種類の警告が検出されます。

- AZS: 0 サイズの割り当て (allocating zero size)
- メモリーリーク
- SMR: 投機的な非初期化メモリーからの読み取り (speculative uninitialized memory read)

このセクションでは、警告の考えられる原因と、警告が発生する可能性があるコードの例を説明します。

サイズ 0 の割り当て (AZS)

例:

```
#include <stdlib>
int main()
{
  int *p = malloc(); // Allocating zero size memory block
}
```

メモリーリーク (MLK)

考えられる原因: メモリーが割り当てられるが、関数の終了またはエスケープの前に解放されていません。

例:

```
int foo()
{
 int *p = (int*) malloc(sizeof(int));
 if (x) {
  p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
 }
}                                     // The 2nd malloc leaked here
```

投機的なメモリーの読み取り (SMR)

例:

```
int i;
if (foo(&i) != 0)  /* foo returns nonzero if it has initialized i */
printf("5d\n", i);
```

コンパイラは、前述のソースに対して次の同等のコードを生成する可能性があります。

```
int i;
int t1, t2'
t1 = foo(&i);
t2 = i; /* value in i is loaded. So even if t1 is 0, we have uninitialized read due to
speculative load */
if (t1 != 0)
```

```
    printf("%d\n", t2);
```

索引

G

X