「EW-25E808 セキュアプログラミング評価手順」補足資料

目次

1-目的

本資料は、「EW-25E808_セキュアプログラミング評価手順」から割愛した環境の設定やコマンドを追記・補足し、運用時の手引書となることを目的とする。

2-適用範囲

「EW-25E808_セキュアプログラミング評価手順」による。

3-Coverity実施手順

Coverity Analysisのワークフローには次の４つのフェーズがある。

- コンパイラ設定フェーズ（コンパイラの設定）
  ビルド中に呼び出されるネイティブコンパイラを識別し、コンパイラの動きをエミュレートする。　プロジェクトに対して一度だけ設定する。
- ビルドフェーズ（ソースコードをビルドし、中間表現にコンパイル）
  Covertyビルド監視下で、ネイティブビルドを監視・実行し、ソースコードを収集、解析のために中間言表現にコンパイルする。
- 解析フェーズ（中間表現を解析し不具合を検出）
  メトリクスデータの収集と不具合の検出を行うためにビルドフェーズの成果物を解析する。
- コミットフェーズ（Coverity Connectに解析結果を登録）
  不具合とソースコードの解析結果をCoverity Connect のデータベースに登録する。

3-1-準備

事前に以下の情報を調査しておくこと。

(1) **ビルド環境**
（例）Eclipse4.6 neon3、RX65N/CC-RX V2.07.00、Android Studio など

(2) **言語**
（例）Java、C/C++　など

(3) **マイコン情報**
（例）M16C、RX65N　など

(4) **コンパイラ情報**
コンパイラのバイナリ名が必要。マイコンの場合はマイコン型式とコンパイラのコマンドライン実行形式ファイル名。IDEの場合はIDE名。

（例）コンパイラおよび（コンパイラのバイナリ名）

- gcc（arm-linux-androideabi-gcc）
- gcc 4.6.2 (arm-none-linux-gnueabi-gcc)
- gcc 4.9.2 (arm-linux-gnueabi-gcc)
- gcc 4.9.2 (arm-linux-gnueabi-g++)
- nc30.exe (renesascc)
  など

(5) **Coverity Connectログイン用アカウント**
予めCoverity Connect管理者からログイン用アカウントを取得しておくこと

3-2-インストール

Coverity2018.09をインストールする。

- cov-analysis-win32-2018.09.exe : Windows 32bit用
- cov-analysis-win64-2018.09.exe : Windows 64bit用
- cov-analysis-linux-2018.09.sh : Linux 32bit用
- cov-analysis-linux64-2018.09.sh : Linux 64bit用

※2018年4月現在のインストーラ保管場所は、

- \\150.35.20.125\Coverity\Windows\Analysis
- \\150.35.20.125\Coverity\Linux\Analysis

バージョンアップがあった場合、Synopsys Software Integrity Communityのアカウント保有者は、このコミュニティの PREMIUM DOWNLOADS https://community.synopsys.com/s/downloads から最新版をダウンロードできる。

(例)

```
Release: Coverity 2017.07-SP2
Operating System: Windows
Packages: Analysis

Download するファイル：
Windows (32-bit) Installer、あるいは、Windows (64-bit) Installer
```

※Linux版は実行権限を付加すること

ライセンスタイプはCoverityを指定する
ライセンスの指定画面ではコミュニティサイトからダウンロードしたライセンスファイルlicense.zipを解凍したlicense.datを指定する

※2018年4月現在、**license.dat** は、

- \\150.35.20.125\Coverity\license\00057310-SAVE-026(Coverity)\license.zip

を使用する

3-3-解析方法

　解析方法には、Coverity Wizard、コマンドライン、および、Visual Studio、Eclipse、Jenkinsなどのプラグインを使う 3 つの方法があるが、本手順ではコマンドラインを使った手順を示す。

3.3.1. プロジェクトとストリームの作成（Coverity Connect管理者）
　Coverity Connectに予めプロジェクトとストリームを作成する。ここでは簡単のために、hello.cプロジェクトを例に説明する。

「+プロジェクト」でプロジェクトを作成後、「+ストリーム」でストリームを作成する。

※ストリームは、プロジェクトの下に作成する解析結果の登録先のこと

3.3.2. コンパイラ設定（コンフィギュア）
(1) ビルドするためのバッチファイルまたはシェルを開いて、コンパイラに必要な環境変数を全て設定する。
　開いたシェル内で、hello.c が使用するNative Compilerで正常にコンパイルされるか確認する。
　Linkまで行い実行ファイルを生成する必要はない。(hello.o が 生成されればよい)

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

gcc,shc,clなどを対応するコンパイラを指定する

```
$ gcc –c hello.c
```

(2) PATH環境変数に <インストール先>\binを追加する。

- Linux/UNIXの場合

```
$ export PATH=/home/fujimoto/coverity/static-analysis/bin:$PATH
```

- Windowsの場合

```
> set PATH="C:\Program Files\Coverity\Coverity Static Analysis\bin";%PATH%
```

(3) **一度だけ**、cov-configure コマンドを実行する。
マイコンのコンパイラ(`--comptype` で指定する compiler type)については、`cov-configure --help` でタイプ一覧を参照する。

- Microsoft C/C++ コンパイラ cl.exe の場合:

```
    cov-configure --msvc
```

- Microsoft C# コンパイラ csc.exe の場合:

```
   cov-configure --csc
```

- Renesas　shコンパイラの場合:

```
   cov-configure --compiler shc --comptype renesascc --template
```

- gcc および g++ コンパイラの場合:

```
   cov-configure --gcc
```

- その他の C/C++ コンパイラの場合:

```
   cov-configure --compiler <path_to_compiler> --comptype <compiler_type> --
template
```

- Javaの場合:

```
   cov-configure --java
```

- JavaScriptの場合:

```
    cov-configure --javascript
```

3.3.3 ビルド
(1) 下記コマンドでビルドを行う。

```
cov-build --dir <intermediate_directory=中間ファイルのフォルダ指定> --encoding
<Shift_JIS/ EUC-JP/ UTF-8 etc.> <ビルドのコマンド>
```

※ encodingタイプについては、cov-build --helpで出力されるタイプ一覧を参照する

- Windowsで、Shift_JISでエンコードされているソースファイルを、Visual Studio IDEでビルドする場合

```
cov-build --dir C:\coverity\covinter\project1 --encoding Shift_JIS "C:\Program
Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe"
C:\work\demo\IDE\testprj2\testprj2.sln
```

※上記のコマンドから Visual Studio IDEが起動し、Solutionファイル(プロジェクトファイル)がロードされる。　リビルドを行い、IDEを終了する。

- Renesasで、Shift_JISでエンコードされているソースファイルを、HEW IDEでビルドする場合

```
cov-build --dir C:\coverity\covinter\project1 --encoding Shift_JIS "C:\Program
Files (x86)\Renesas\Hew\Hew2.exe"
```

※上記のコマンドから HEW IDEが起動するので、Solutionファイル(プロジェクトファイル)をオープンし、クリーン、ビルドを行う。

- Linux/UNIXで、EUCでエンコードされているソースファイルを、makeコマンドでビルドする場合

```
cov-build --dir /home/covinter/project1 --encoding EUC-JP make
```

- Javaで、antコマンドでビルドする場合

```
cov-build --dir /home/covinter/project1 ant
```

※ビルド時間はネイティブビルドの3倍程度かかる。

(2) ビルドを開始すると、<intermediate_directory>/build-log.txt にログが出力される。

※　ビルド完了後、「build-log.txt」ファイルを参照する。ログの最後の部分に、ビルド結果 (全ソースの何パーセントがビルドされたかを表す数値) が出力される。　この数値はコンソール画面にも表示される。　状況にもよるが、90%以上のビルド結果が得られた場合には、解析の段階に進むことができる。

(3) Java Web APIのキャプチャーを行う (Security Advisorのみ実施)

```
cov-emit-java --dir <intermediate_directory=中間ファイルのフォルダ指定> --webapp-
archive <WARファイルの存在するパス>
```

3-4-解析

下記のコマンドで解析を行う。

```
cov-analyze --dir <intermediate_directory=中間ファイルのフォルダ指定> --all
```

解析が完了すると画面上にテキストで結果のサマリーが出力される (検出された不具合の情報)

- Security Advisorを含む解析の場合
  下記のコマンドで解析を行う。

```
cov-analyze --java --webapp-security-aggressiveness-level high --distrust-
database --distrust-filesystem
```

解析が完了すると画面上にテキストで結果のサマリーが出力される (検出された不具合の情報)

3-5-コミット

　検出された不具合を閲覧するためには、解析結果をCoverity Connect のデータベースに登録する必要がある。この作業をコミットと呼ぶ。
　コミットは、下記のコマンドで行う。

```
cov-commit-defects --dir <intermediate_directory>  --host <CoverityConnectのホスト名>    --user admin --password <管理者のパスワード> --stream <ストリーム名>
```

コミットが完了すると、ブラウザから [http://<CoverityConnectのホスト名>:8080/] にアクセスし、Coverity Connectで不具合を見ることが出来る。

※2018年12月現在、<CoverityConnectのホスト名>:は、150.35.5.147　である

3-6-実施例

(1) ルネサスM16C用コンパイラ

　0. コンパイラのコンフィグレーション（一度だけ実行する）

```
C:\Users\kde\Documents\HEW\M5D_INXX>cov-configure --template --comptype
renesascc --compiler nc30.exe
```

１．IDEキャプチャの場合（HEWによるビルド）
1-1. 解析するワークスペースを以下のコマンドで開く

```
C:\Users\kde\Documents\HEW\M5D_INXX>cov-build --dir cov-im "C:\Program Files
(x86)\Renesas\Hew\hew2.exe" "ow M5D_INXX.hws"
```

1-2. IDEでワークスペースが開くことを確認し、フルビルドを行う。(手作業)

1-3. ビルドが終わればIDEを終了させる(手作業)
コンソールを確認するとcov-buildの結果が表示される。

```
Coverity Build Capture (64-bit) version 8.7.1 on Windows 7 Professional, 64-bit,
 Service Pack 1 (build 7601)
Internal version numbers: a59584cec3 p-lodi1-push-28091.44.570

Emitted 51 C/C++ compilation units (100%) successfully

51 C/C++ compilation units (100%) are ready for analysis
The cov-build utility completed successfully.
```

1-4. 解析

- MISRA有効の場合

```
C:\Users\kde\Documents\HEW\M5D_INXX>cov-analyze --dir cov-im --all --enable-
constraint-fpp --force --misra-config "C:\Program Files (x86)\Coverity\Coverity
Static Analysis\config\MISRA\MISRA_c2012_7.config"
```

- MISRA無効の場合

```
C:\Users\kde\Documents\HEW\M5D_INXX>cov-analyze --dir cov-im --all --enable-
constraint-fpp --force
```

1-5. コミット

```
C:\Users\kde\Documents\HEW\M5D_INXX>cov-commit-defects --dir cov-im --stream
PGL17A051 --host 150.35.6.125 --port 8080 --user shimatani --password 170522
```

２．コマンドラインからビルドする場合
2-1.　IDEからmakeファイルをエクスポートする。
(例) M5D_INXX.mak を出力する

2-2.　クリーン

```
C:\Users\kde\Documents\HEW\M5D_INXX\make>del /s /q /f ..\M5D_INXX\Debug\*.R30
..\M5D_INXX\Debug\*.mot ..\M5D_INXX\Debug\*.map
```

2-3. ビルド

```
C:\Users\kde\Documents\HEW\M5D_INXX\make>cov-build --dir cov-im2 "C:\Program
Files (x86)\Renesas\Hew\hmake" M5D_INXX.mak CONFIG=Release
```

あるいは、

```
C:\Users\kde\Documents\HEW\M5D_INXX\make>cov-build --dir cov-im2 "C:\Program
Files (x86)\Renesas\Hew\hmake" M5D_INXX.mak CONFIG=Debug
```

2-4. 解析

- MISRA有効の場合

```
C:\Users\kde\Documents\HEW\M5D_INXX>cov-analyze --dir cov-im2 --all --enable-
constraint-fpp --force --misra-config "C:\Program Files (x86)\Coverity\Coverity
Static Analysis\config\MISRA\MISRA_c2012_7.config"
```

- MISRA無効の場合

```
C:\Users\kde\Documents\HEW\M5D_INXX>cov-analyze --dir cov-im2 --all --enable-
constraint-fpp --force
```

2-5. コミット

```
C:\Users\kde\Documents\HEW\M5D_INXX\make>cov-commit-defects --dir cov-im2 --
stream PGL17A051 --host 150.35.6.125 --port 8080  shimatani --password 170522
```

(2) Visual Studi(MSBUILD)

0. コンパイラのコンフィギュレーション（一度だけ実行する）
   Coverityのコンフィグレーションで、Visual Studioのコンパイラ用のコンフィグレーションを行う

```
c:\work>cov-configure --msvc
```

１. MSBUILDコマンドを使い、コンソールからソリューションファイルをビルドする場合
1-1. クリーン

```
c:\work>msbuild "C:\Users\kde\Documents\Visual Studio
2013\Projects\Win32Project1\Win32Project1.sln" /p:Configuration=Debug
/p:Platform="Win32" /t:Clean
```

1-2. ビルド
(例１)

```
c:\work>msbuild "C:\Users\kde\Documents\Visual Studio
2013\Projects\Win32Project1\Win32Project1.sln" /p:Configuration=Debug
/p:Platform="Win32"
```

※/p:Platform=""　この中をVCのビルドターゲットに合わせる

(例２)

```
c:\work>msbuild "C:\Users\kde\Documents\Visual Studio
2013\Projects\caffe\windows\caffe.sln" /p:Configuration=Release
/p:Platform="x64"
```

1-3. リビルド（クリーン込み）

```
c:\work>msbuild "C:\Users\kde\Documents\Visual Studio
2013\Projects\Win32Project1\Win32Project1.sln" /p:Configuration=Debug
/p:Platform="Win32" /t:Rebuild
```

※Coverityでの解析手順(cov-imフォルダにビルド結果が格納される)

２．Coverityコマンドを使い、コンソールからソリューションファイルをビルドする場合

2-1. ビルド
(例１)

```
c:\work>cov-build --dir cov-im msbuild "C:\Users\kde\Documents\Visual Studio
2013\Projects\Win32Project1\Win32Project1.sln" /p:Configuration=Debug
/p:Platform="Win32" /t:Rebuild
```

(例２)

```
c:\work>cov-build --dir cov-im msbuild "C:\Users\kde\Documents\Visual Studio
2013\Projects\caffe\windows\caffe.sln" /p:Configuration=Release
/p:Platform="x64" /t:Rebuild
```

2-2. 解析

```
c:\work>cov-analyze --dir cov-im --all
```

2-3. コミット
(例１)

```
c:\work>cov-commit-defects --dir cov-im --host 10.1.64.166 --port 8888 --stream
vc-test --user admin
```

(例２)

```
c:\work>cov-commit-defects --dir cov-im --host 150.35.6.125 --port 8080 --stream
d-bips --user ninomiyak
```

3-7-MISRA解析

通常の解析とMISRA用の解析を同時に実行することができる。

- オプション `--misra-config file` を指定してMISRAチェックを行う場合に、通常のCoverityのチェッカーも同時に実行できるようになる。

(例)

```
cov-analyze --dir i5 --misra-config "c:\Program Files\Coverity\Coverity Static
Analysis\config\MISRA\MISRA_c2012_7.config“ --all
```

MISRAルールをルール毎にコントロール（有効化/無効化）できる。

- ルールは7種類のグループに分類され、グループ単位でのルールの有効化を行う。
- ルール毎にDeviationを指定でき、これによって、独自のレベルの定義も可能である。
- MISRA 2012の場合には、Mandatory 以外のルールで Deviations を指定できる。

3-8-解析結果の確認

解析結果は、Coverity Connectサーバにログインして確認する。

※2018年12月現在、Coverity Connectサーバは、http://150.35.5.147:8080/

左上の(S)マークの右側の灰色プルダウンからプロジェクトを選択し、次に(S)マーク下のハンバーガーメニュー（≡）をクリックして、「問題：スナップショット別」の配下に、

「影響度：高」未修正（※）
「未修正」
　：
　：
「CERT-C_C++」
「CERT-Java」
「High Impact Outstanding」※と同じ
「MISRA-C」
「OWASP Top-10 2013」

等があることを確認し、指摘事項を修正していく。

3-9-参照先

詳細はCoverity Connect右上のヘルプ～Coverity ヘルプセンター http://150.35.5.147:8080/doc/ja/index.html
あるいは、
\\150.35.15.209\k\開発g\セキュリティ\0203.セキュリティツール\シノプシス\Coverity\Coverity マニュアル
を参照のこと。

4-ワークフロー例

　開発者のビルドマシンに、Jenkinsの新しいプロジェクトを作成し、CIサイクルを定期的に実施する場合のワークフローを以下に示す。

4.1 ソースコードの更新
　GitBlit/GitBucketから最新のソースコードを取り出し、開発環境のソースコードを最新の状態に更新する。

4.2 ビルドの確認
　ビルドを実行しエラーが出ないことを確認する。

4.3 Coverity Analysisのコンパイラ設定（コンパイラ設定フェーズ）
　Coverity Analysisにコンパイラを識別させて正しくエミュレートさせるために、コンパイラの設定を行う。　作成したプロジェクトに対しては一度だけ実行する。

4.4 Coverity Analysisのビルド（ビルドフェーズ）
　コマンドプロンプトあるいはターミナル上でCoverity Analysisコマンドを実行し、ビルドと並行してCoverity Analysisが動作する（Coverity Analysisのビルドコマンドからターゲットのビルドコマンドを呼び出す）こと、およびCoverity Analysisのビルドが90%以上成功することを確認する。

4.5 バージョン管理情報の取得
　GitBlit/GitBucketからソースコードの管理情報（改訂者、変更日、バージョン）を取得する

4.6 Coverity Analysisの解析検査（解析フェーズ）
　ソースコードの脆弱性検査を実行する

4.7 Coverity Analysis解析検査結果の登録（コミットフェーズ）
　脆弱性検査結果をCoverity Connectに登録する

4.8 Coverity Connectで解析検査結果をレビューする
　WebブラウザからCoverity Connectに接続し、問題個所を確認する。検出された問題が本当のバグか誤検知かを判断し、重要度、アクション、担当者などの設定を行う。（選別の項参照）

4.9 ソースコードの修正
　検出された問題がバグの場合、ソースコードの修正を行う。

4.10 リビルド
　修正したソースコードをリビルドし動作を確認する。

4.11 ソースコードの登録
　動作を確認したソースコードをGitBlit/GitBucketに登録する。

4.12 選別
　開発者は、Coverity Connectにログイン後、解析結果をレビューし、検出された問題が本当のバグなのか誤検知なのかを判断し、重要度、アクション、担当者などの設定を行う。　Coverity Connectではこれらの設定行為を選別と呼んでいる。　選別内容は次の通りである。

- 分類
  保留・誤検知・意図的・バグ　のいずれかから選択する。（選択前は未分類） Coverity Analysisが影響度を高と判定した問題は、バグを選択すること。
- 重要度
  高・中・低　のいずれかから選択する。（選択前は未指定） Coverity Analysisが判定した影響度の高・中・低を参考にする。　レビューの結果を反映して変更してもよい。
- アクション
  要修正・修正完・無視　のいずれかから選択する。（選択前は未確定） Coverity Analysisが影響度を高と判定した問題は、要修正を選択すること。
- レガシー
  既存の不具合をレガシーと呼び、真・偽　のいずれかから選択する。
- 担当者
  担当者名　をアサインする。

※選別項目を追加・削除・変更することができる

5-Coverity用語集

5.1 Coverity Static Analysis
　ソースコードの解析を行うソフトウェア。　ビルドを行うPCにインストールする

5.2 Coverity Connect
　Coverity Analysisが解析した結果を登録しているサーバ側のソフトウェア

5.3 共通脆弱性タイプ一覧　(CWE: Common Weakness Enumeration)
　報告されたソフトウェアの脆弱性についてまとめたリストです。脆弱性にはそれぞれ番号が割り当てられています (たとえば、CWE-476 は http://cwe.mitre.org/data/definitions/476.html に詳しく説明されています)。　Coverity では、多くの不具合のカテゴリ (「NULL ポインタの間接参照」など) に CWE 番号を割り当てています。

5.4 静的解析 (static analysis)
　コンパイルされたプログラムを実行することなく実施されるソフトウェアコードの解析。

5.5 動的解析 (dynamic analysis)
　コンパイルされたプログラムを実行することで実施されるソフトウェアコードの解析。

5.6 検知漏れ (false negative)
　Coverity Analysis の解析で検出されないソースコード内の不具合。

5.7 誤検知 (false positive)
　Coverity Analysis によって特定された潜在的な不具合で、ユーザが不具合でないと判断したもの。Coverity Connect では、このような問題を誤検知として無視できる。　また、ソースコードで注釈 (コード注釈とも呼ばれます) を使用して、解析結果を Coverity Connect に送信する前のソースコード解析段階でこのような問題を [意図的 (Intentional)] として識別することも可能である。

5.8 サニタイズ (sanitize)
　データが有効であることを確認するために、汚染されたデータを無害化または検証すること。　汚染されたデータのサニタイズは、セキュアなコーディングを実施し、システムのクラッシュや破損、権限昇格、サービス拒否を排除する上で重要な役割を果たす。

5.9 汚染されたデータ (tainted data)
　ユーザからの入力としてプログラムに与えられるデータ。　プログラムには入力された値を制御する機能がないため、このデータを使用する前にデータをサニタイズし、システムのクラッシュや破損、権限昇格、サービス拒否を排除する必要がある。

5.10 スナップショット (snapshot)
　開発プロセスのある時点でのコード ベースのコピー。　スナップショットは、開発中に発生した不具合を分離する際に役立つ。
　スナップショットは解析結果を示すもので、問題に関する情報と、問題が検出されたソース コードの両方が含まれている。　Coverity Connect では、不正なデータを登録した場合やテスト目的でデータを登録した場合、スナップショットを削除できる。

5.11 選別 (triage)
　特定のストリーム内の問題または複数のストリームで発生した問題の状態を設定するプロセス。これらユーザ定義の状態は、問題の重要度、予想される結果 (誤検知) かどうか、問題に必要なアクション、問題の割り当て先などの項目を反映している。　これらの詳細情報によって、プロダクトの追跡情報を把握できるようになる。　Coverity Connect には、1 つまたは複数のストリームにわたって存在する問題の情報を個別に更新したり、まとめて更新できるメカニズムが搭載されている。

5.12 中間ディレクトリ (intermediate directory)
　--dir オプションでさまざまなコマンドに指定されているディレクトリ。　このディレクトリの主な機能は、ビルドと解析の結果がスナップショットとして Coverity Connect データベースに登録される前に、これらのデータを記録することである。
　--dir オプションをサポートするその他の特殊なコマンドも、このディレクトリでデータの書き込みと読み取りを実行する。
　ビルドの中間表現は <intermediate_directory>/emit ディレクトリに、解析結果は <intermediate_directory>/output に保存される。
　このディレクトリには複数の言語 (C/C++、C#、Java コード ベース) のビルドおよび解析結果を含めることができる。

5.13 中間表現 (intermediate representation)
　解析を実施し、不具合をチェックするために Coverity Analysis が使用するCoverity コンパイラの出力。コードの中間表現は中間ディレクトリにある。

5.14 問題 (issue)
　Coverity Connect には、品質問題、潜在的セキュリティ脆弱性、テスト ポリシー違反の 3 つのタイプのソフトウェア問題が表示される。　チェッカーの中には、品質問題と潜在的セキュリティ脆弱性の両方を検出するものもあれば、1つのタイプの問題に焦点を当てたものもある。　Coverity Connect の品質、 セキュリティ、テスト アドバイザのダッシュボードはそれぞれ、各タイプの問題にハイレベルのメトリクスを提供する。

5.15 ネイティブ ビルド (native build)
　ソフトウェア開発環境における通常のビルド処理。　Coverity 製品は関係しない。

5.16 影響度 (impact)
　主にソフトウェアの品質とセキュリティに対する影響とともに、チェッカーの精度も考慮して判断された、問題修正の緊急度を示すことを意図した用語。　影響度は必然的に、主観的かつ確率的であるため、これのみを基に優先順位付けを行うことは推奨されない。

5.17 ストア (store)
　抽象構文ツリーから整数値およびイベント シーケンスへのマップ。　このマップは、フロー センシティブな解析に使用される抽象インタプリタの実装に使用される。

5.18 ストリーム (stream)
　スナップショットのシーケンシャル コレクション。　ストリームは、経時的および開発プロセスにおける特定時点のソフトウェアの問題に関する情報を提供する。

5.19 混在 (various)
　Coverity Connect では次の 2 つの場合に混在 (Various) という用語を使用する。

- チェッカーが品質チェッカーおよびセキュリティ チェッカーの両方として分類されている場合。　たとえば、USE_AFTER_FREE および UNINIT はビューペインの [問題の種類 (Issue Kind)]

列に [混在 (Various)] として表示される。

- CID が同じインスタンスの選別結果が異なる場合。　プロジェクトの範囲内で、指定の CID が別々のストリームに発生するインスタンスは、ストリームが異なる に割り当てられている場合、特定の選別属性で異なる値を持つ可能性がある。
  たとえば、高度な選別を使用して、ある選別ストアでは CID を [バグ (Bug)] として分類し、別のストアではデフォルトの [未分類 (Unclassified)]設定のままにする場合がある。　このような場合、Coverity Connect のビュー ペインでは、CID のプロジェクト全体の分類は [混在 (Various)] として表示される。 すべてのストリームが単一の選別ストアを共有する場合は、この選別状態のCID が発生することはない。

6-改定来歴

改定年月日 改定理由