ALICE experience using Coverity

Federico Carminati, Peter Hristov, Axel Naumann and Olga Datskova (CERN)
AliRoot is the ALICE Off-line framework used for simulation, reconstruction, analysis and visualisation of experiment data. [1]

AliRoot is developed with the following dependencies in mind [2]:

- **ROOT** is the foundation framework upon which AliRoot is built.
- **AliEn** provides GRID support to users and sites.
- **GEANT3** and **GEANT4** are used for simulation and Monte-Carlo productions.

Main tasks for static C/C++ code analysis in AliRoot:

- Maintaining good quality of code with a large developer base.
- Ensuring rapid fixes in order to keep up with weekly release cycles.
- Debugging irreproducible problems.
“Coverity analysis solutions enable users to test their code against business policies and thresholds while in development. Finding and addressing defects early in the lifecycle saves developer’s time by minimizing rework and keeping releases on schedule.” [3]

Coverity analysis suite consists of the following set of tools [3]:

- **Coverity Static Analysis** is a command line program for identifying program defects through static source code inspection. The tool supports C/C++, Java and C# programming languages.

- **Coverity Dynamic Analysis** performs dynamic evaluation of running code. The tool supports Java programming language.

- **Coverity Integrity Manager** is a web interface for managing defects, which were discovered by Coverity Static Analysis and Coverity Dynamic Analysis tools.
Coverity features

Coverity Static Analysis and Integrity Manager have the following features:

- The installation procedure for both distributions is very straightforward and is done through the provided installation scripts.

- Coverity Static Analysis tool provides the flexibility of a command line, allowing for greater control over the build and analysis processes.

- Coverity Integrity Manager has the following aspects:
  - Database for managing defects and users assigned to them. The database has the functionality to be queried securely for remote database administration.
  - Web server allows for a centralised web service, where users can log in and examine their code. The system supports LDAP authentication.
AliRoot has been set up to undergo consistent daily static analysis. The following steps describe the process:

1) AliRoot sources are updated with the latest development code and fixes.

2) The sources are then built through cov-build tool, producing intermediate code in $COVERITY_OUT.

3) Analysis of the built sources can then commence. Here we specify the desired checks that the tool must perform.

4) After successful completion of analysis, the resulting reports are committed to the Coverity Integrity Manager.

5) The developer then checks through the web page for defects assigned to him/her and starts to work on the fixes as necessary.
To ensure persistent quality of code, the following policies were introduced:

- **All** defects must be fixed promptly irrespective of their impact.
- “Top 10” users receive email notifications every day. The message also contains compilation warnings which should be taken care of quickly.
- Recent addition (comes into effect on the 11th of July): users have 7 days to fix all their defects, otherwise they are banned from subversion until they do.
Coverity results for AliRoot

- Coverity was put into active use starting from January 2011.
- Approximately 6000 defects have been identified initially by Coverity.
- At present time, we have 12 defects with more fixes being committed and new code developed every day.
Coverity provides its own classification of discovered defects into appropriate impact categories as seen in the project side menu:

<table>
<thead>
<tr>
<th>Defect Type:</th>
<th>Memory - corruptions</th>
<th>Memory - illegal accesses</th>
<th>Resource leaks</th>
<th>Uninitialized variables</th>
<th>API usage errors</th>
<th>Class hierarchy inconsistencies</th>
<th>Control flow issues</th>
<th>Error handling issues</th>
<th>Incorrect expression</th>
<th>Insecure data handling</th>
<th>Integer handling issues</th>
<th>Null pointer dereferences</th>
<th>Program hangs</th>
<th>Build system issues</th>
<th>Code maintainability issues</th>
<th>Parse warnings</th>
<th>Performance inefficiencies</th>
<th>Security best practices violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Impact:</td>
<td>Set</td>
<td>Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Impact:</td>
<td>Set</td>
<td>Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Coverity Integrity Manager menu (26/06/2011)
High impact defect: Out of bounds write

Defect #16203: when i = 20, an attempt to write to 21\textsuperscript{st} value of fHPionInvMasses will be made, which has been defined to have only 20 elements.

```cpp
fOutput->Add(fHPionMggDgg);
const Int_t nbins = 20;
Double_t xbins[nbins] = {0.5,1.1.5.2.2.5.3.3.5,4.4.5.5,6.7.8.9,10.12.5,15,20,25,50};
fPtRanges = new TAxis(nbins-1,xbins);
```

At conditional(0): "i <= 20" taking the true branch
At conditional(1): "i <= 20" taking the true branch
At conditional(11): "i <= 20" taking the true branch

● for (Int_t i = 0; i<nbins; ++i) {

```
CID 16203: Out-of-bounds write (OVERRUN_STATIC)
Overrunning static array "this->fHPionInvMasses", with 20 elements, at position 20 with index variable "i".

fHPionInvMasses[i] = new TH1F(Form("fPionInvMass%d",i,"","1000,0,2");
fHPionInvMasses[i]->SetXTitle("M_{#gamma#gamma} [GeV/c^{2}]");
```

At conditional(9): "i == 0" taking the true branch.
At conditional(9): "i == 0" taking the false branch.

● if (i==0)
```
fHPionInvMasses[i]->SetTitle(Form("0 < p_{T}\{#gamma#gamma\} <%.1f",xbins[0]));
```

At conditional(10): "i == 20" taking the false branch.

● else if (i==nbins)
```
fHPionInvMasses[i]->SetTitle(Form("p_{T}\{#gamma#gamma\} > 50");
```

● else
```
fHPionInvMasses[i]->SetTitle(Form("%.1f < p_{T}\{#gamma#gamma\} <%.1f",xbins[i-1],xbins[i]);
```

● fOutput->Add(fHPionInvMasses[i]);

}

Solution from the developer:
increment array size

<table>
<thead>
<tr>
<th>#</th>
<th>Line 102</th>
<th>Line 104</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>TH2F *fHPionMggPt: //histo for pion mass vs pt</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>TH2F *fHPionMggAsym: //histo for pion mass vs asym</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>TH2F *fHPionMggDgg: //histo for pion mass vs opening angle</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>TH1F *fHPionInvMasses[20]: //histos for invariant mass plots</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TH2F *fHPionMggPt: //histo for pion mass vs pt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TH2F *fHPionMggAsym: //histo for pion mass vs asym</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TH2F *fHPionMggDgg: //histo for pion mass vs opening angle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TH1F *fHPionInvMasses[21]: //histos for invariant mass plots</td>
<td></td>
</tr>
</tbody>
</table>
Another example: Out of bounds write

Defect #16971: mismatch in the number of labels within the 'for' loop index and GetLabel() result.

```c
void StartEvent();

enum {kMaxLab=24}; // maximum number of MC labels associated to the cluster
Int_t ProcessHit(Int_t layer, UInt_t col, UInt_t row, UShort_t charge, Int_t label[kMaxLab]);
```

```c
for (Int_t i=0; i<kMaxLab; i++){
    label[i] = pix->GetLabel(i);
}
```

```c
/ITS/UPGRADE/AlIITSUPixelModule.h
34   UInt_t GetCol() const { return fCol; }
35   UInt_t GetRow() const { return fRow; }
36   UInt_t GetCharge() const { return fCharge; }
37   Int_t GetLabel(Int_t i) const { return fLabels[i]; }
38   void PrintInfo();
39
40 protected:
743 } SetLabels(label);
744 }
746 }
```

```c
enum {kMaxLab=12}; // maximum number of MC labels associated to the cluster
Int_t GetLabel(Int_t i) const { return fLabels[i];}
void PrintInfo();
```

protected:
  UInt_t fCharge;
  UShort_t fModule;
  UInt_t fCol;
  UInt_t fRow;
  Int_t fLabels[kMaxLab];
```
High impact defect: Use after free

Defect #16195: `arrayValues` and `arrayWeights` are released in memory, then subsequently used in `TMath::Mean()` function.

```cpp
2334     arrayWeights[i-i_entrySOR] = (Double_t)(timestamp2 - (Int_t)v->GetTimeStamp());
2335     arrayValues[i-i_entrySOR] = (Float_t)v->GetUInt();
2336 }

CID 16194 (2): Use after free (USE_AFTER_FREE) [select defect]
| 2337     delete [] arrayValues;
CID 16195: Use after free (USE_AFTER_FREE)
“operator delete[]” frees “arrayWeights”.
| 2338     delete [] arrayWeights;

Passing freed pointer “arrayWeights” as an argument to function “double TMath::Mean<float>(long long, float const *, double const *)”.
| 2339     aDCSArrayMean = TMath::Mean(iCountsRun, arrayValues, arrayWeights);
2340 }
```

Solution from the developer: get the Mean before deleting the arrays
High impact defect: Resource leak

Defect #14153: memory leak occurs when one allocates memory with a \texttt{new} operator and does not release the resources at the end of variable scope.

```
1229  Double_t * adx  = new Double_t[ncalibs];
1230  Double_t * ady  = new Double_t[ncalibs];
1231  Double_t * adz  = new Double_t[ncalibs];
1232  Double_t * adr  = new Double_t[ncalibs];

CID 14153: Resource leak (RESOURCE_LEAK) [select: detect]
Calling allocation function "operator new[](unsigned long)".
Assigning "adrph" = storage returned from "new Double_t[ncalibs]"
```

```
1233  Double_t * adroph = new Double_t[ncalibs];
```

```
1320  printf("x0=%f finished",x[0]);
1321  }
```

Solution: \texttt{always} use \texttt{delete} with the \texttt{new} operator
Medium and Low impact defects

Defect #16952: nothing out of the ordinary?

```c
for (Int_t i = 0; i < dim + 1; i++) {
    Int_t entries = 0;
    if (i < dim) entries = fTree->Draw(((TObjString*)formulaTokens->At(i))->GetName(), cutStr.Data(), "goff", stop-start,start);
    else entries = fTree->Draw(drawStr.Data(), cutStr.Data(), "goff", stop-start,start);
    if (entries != entries) {
        CID 16952: Infinite loop (INFINITE_LOOP)
        Top of the loop.
        Bottom of the loop.
        "j < dim + 1" must remain true for the loop to continue.
        for (Int_t j = 0; j < dim + 1; i++) {
            if(values[j]) delete values[j];
        }
        delete[] values;
    }
}
```

Defect #15833: unsafe copy. [7]

```c
if(data.Sec<36)
    ddlNumber=data.Sec*2+data.SubSec;
else
    ddlNumber=72+(data.Sec-36)*4+data.SubSec;
```

CID 14891: Calling risky function (SECURE_CODING) [select defect]
CID 15833: Copy into fixed size buffer (STRING_OVERFLOW)
You might overrun the 100 byte fixed-size string "filename" by copying the return value of "AliDAQ::DdFileNam(char const *, int)" without checking the length.

```c
strcpy(filename,AliDAQ::DdFileNam("TPC",ddlNumber));
Int_t patchIndex = data.SubSec;
if(data.Sec>=36) patchIndex += 2;
```
False positives

Defect #14425: only using a string in `sscanf` and `fscanf` may be dangerous.

```c
if (sopt.Contains("MEAN") ) {
    Int_t j(0);
}
```

CID 14425: Calling risky function (SECURE_CODING) (VERY RISKY). Using "sscanf" can cause a buffer overflow when done incorrectly. `sscanf()` assumes an arbitrarily large string, so callers must use correct precision specifiers or never use `sscanf()`. Use correct precision specifiers or do your own parsing.

```c
sscanf(sopt.Data(), "MEAN", &j);
```

Defect #11174: `kSPECIES` being a constant value can not be less than or equal to 0 and `w` is initialised before use.

```c
Double_t probability[5] = {0.0, 0.0, 0.0, 0.0, 0.0};
Double_t w[5] = {0.0, 0.0, 0.0, 0.0, 0.0};
```

CID 11174: Improper use of negative value (NEGATIVE_RETURNS) Function "TMath::LocMax<double>(5LL, w)" returns a negative number. [hide details]

Assigning: signed variable "partType" = "long long TMath::LocMax<double>(long long, double const *)".

```c
if(!PIDtype.Contains("Bayesian")) {
    if (PIDtype.Contains("Bayesian")) {
        Double_t partType = TMath::LocMax(AliPID::kSPECIES, w);
    }
}
```

```
/coverage/root/trunk/include/TMath.h
```

```c
Long64_t TMath::LocMax(Long64_t n, const T *a) {
    // Return index of array with the maximum element.
    // If more than one element is maximum returns first found.
    // Implement here since it is faster (see comment in LocMin function)
    // At conditional (1): "n <= 0LL" taking the true branch.
    Explicitly returning negative value "-1LL".
    if (n <= 0 || !a) return -1;
    T xmax = a[0];
    Long64_t loc = 0;
```

```c
class AliPID : public TObject {
public:
    enum {
        kSPECIES = 5, // Number of particle species recognized by the PID
        kSPECIESN = 10, // Number of charged-neutral particle species recognized by the PHOS/EMCAL PID
        kSPECIESLN = 4 // Number of light nuclei: deuteron, triton, helium-3 and alphea
    };
```
False positives and modeling

Defect #16969: defect description suggests that the dynamic_cast may fail and return a NULL. However as seen below the inputHandler will immediately go into AliFatal and exit the program.

In order to avoid recurring reports of false positive cases, Coverity static analysis allows for custom function implementations:

- Function producing a false positive case is implemented in a separate .cxx file and adjusted as necessary.
- The source file is then built into a model using cov-make-library.
- Finally, when starting the analysis the –user-model-file is specified along with the library file to use.
Coverity use overview for AliRoot

- The quality of AliRoot code has improved.

- Developers have become more aware of their coding habits. With a centralised system providing complete visibility of all the defects along with some encouragement to fix the problems, developers have become more diligent in their development practice. Defects are now fixed promptly.

- Complementing dynamic analysis, Coverity has greatly helped in debugging hard to diagnose problems.

- Coverity as a static analysis tool used in conjunction with dynamic analysis is an invaluable solution in any development process.
Additional static analysis tools

- **Rule Checker** [8]: performs static code check according to predefined rules, ensuring compliance with both C/C++ coding standards and experiment specific coding conventions. The analysis process is as follows:
  - In the source directory: `$ svn update && make check-all`
  - After analysis has completed, the following reports are produced and sent to the developer to be fixed:

  **NamingRule**: "RN13": Local variables names start with a lower case letter. The variable: AcoHit [file: AliACORDEQADataMaker.cxx line:198] does not start with a lower case letter

  **CodingRule**: "RC11": Make const all member functions that are not supposed to change member data. The method: ASideHasHit in file [file:AliFMDOfflineTrigger.cxx line: 60 ] can be declared const

The Rule Checker has been developed by the **Bruno Kessler Foundation** [9].

Bibliography


