Working with ROSE

David Svoboda
Overview

The Problem:

• How to recognize Insecure Code?

Techniques:

• Automated Security Checking
• Static Analysis
• Abstract Syntax Tree (AST)
• ROSE

So how do we actually use ROSE?
Agenda

We will build a rule checker, showing how ROSE helps us.

- ROSE Setup
- ROSE Documentation
- Background
- Design
- Examining Source Code using ROSE
- Code
- Run & Test
- Useful ROSE Functions
What is ROSE?

Developed at Lawrence Livermore National Labs (LLNL)

- Analyzes program source code
- Produces Abstract Syntax Tree (AST)
- Can then be used for static analysis

We will use ROSE to enforce secure coding rules

http://rosecompiler.org/
Rosebud

Rosebud is a Virtual Machine that is useful for working with Rose.

- Rose and the checkers are already built; no need to compile
- Cross-platform, runs as a VM
- Includes popular developer tools (Eclipse, emacs, etc)
Rosebud 2

Download the ‘rosebud’ VM from

rosecheckers.sourceforge.net

You will need VMWare Player, to run Rosebud. VMWare Player is freely available at:

downloads.vmware.com

Extract the Rosebud package and start VM Player.
Rosebud 3

In VMPlayer, select *Open an Existing Virtual Machine*. When it prompts you for a virtual machine (VM) to open, go to the *rosebud directory*, and Select *rosebud.vmx*. This 'boots up' the Rosebud virtual machine. After a few seconds, a login prompt will appear. Enter username: *rose* password: *roserose*
The system will then re-prompt you for the password, re-enter it.
The system will then give you a command-line prompt (a single `%`)
Type *startx* `<RETURN>`. This will bring up the GUI.
Rosebud 4

After desktop turns blue, right-click on the desktop. This brings up the program menu.

You should now be able to build and test the rules...you can do this with these commands in a terminal:

```bash
  cd ~/src/rosecheckers
  make tests
```
ROSE Setup on Andrew

Your environment should contain the following:

```
setenv ROSE /afs/andrew/usr/svoboda/public/rose
setenv LD_LIBRARY_PATH $ROSE/lib:$LD_LIBRARY_PATH
setenv PATH $ROSE/bin:$PATH
```

Check out the Rosecheckers project from SourceForge.

```
svn checkout
  https://anonymous@rosecheckers.svn.sourceforge.net/svnroot/rosecheckers/trunk/rosecheckers
```

You should now be able to build and test the rules.
ROSE is a project to define a new type of compiler technology which allows compilation techniques to address the optimization of user-defined abstractions. Due to the nature of the solution we provide, it is also an open compiler infrastructure that can be used for a wide number of other purposes. The software developed to support ROSE research work provides an open general purpose and robust compiler infrastructure to support numerous tools and external collaborations in C, C++, and F90.

User-defined abstractions are built from within an existing base language and carry specific semantic information which can't be communicated to the base language's compiler. In many situations, the semantic information could be useful within program...
ROSE Documentation

User Manual
  Full documentation for the Rose features and techniques

Tutorial
  Guide to installing ROSE and some of its utilities

Programmer’s Reference
  Web-based documentation for each class and method in ROSE.

Generated by Doxygen
Programmer’s Reference 1

SglfStmt Class Reference

#include <Cxx_Gramar.h>

Inheritance diagram for SglfStmt:

```
SgNode
  ↓
SgLocatedNode
  ↓
SgStatement
  ↓
SgScopeStatement
  ↓
SglfStmt
```

Collaboration diagram for SglfStmt:

[Diagram]
Programmer’s Reference 2
Building a Rule Checker

We’ll study rule \textit{STR31-C}
Test Cases

Before coding, we need at least one positive test case and one negative test case. These will prove to us that the code works.

The test directory contains compliant and noncompliant code examples...all compliant examples pass all the secure coding rules. The noncompliant code examples each fail a single secure coding rule.

Our first two test files will be test/c.ncce.wiki.STR.c and test/c.cce.wiki.STR.c
Non-Compliant Code Example

```c
#include <string.h>
#include <stdlib.h>

int main() {
    /* ... */
    char buff[256];
    strcpy(buff, getenv("EDITOR"));
    /* ... */
    return 0;
}
```

From `test/c.ncce.wiki.STR.c`
Compliant Code Example

```c
#include <string.h>
#include <stdlib.h>

int main() {
    /* ... */
    char* editor;
    char* buff;
    editor = getenv("EDITOR");
    if (editor) {
        buff = (char*) malloc(strlen(editor)+1);
        if (!buff) {
            /* handle malloc() error */
        }
        strcpy(buff, editor);
    }
    /* ... */
    return 0;
}
```

From `test/c.cce.wiki.STR.c`
Design Idea

```c
#include <string.h>
#include <stdlib.h>

int main() {
    /* ... */
    char buff[256];
    strcpy(buff, getenv("EDITOR"));
    /* ... */
    return 0;
}
```

An attacker can compromise the system by setting the EDITOR environment variable to a string larger than 256 chars!

getenv() makes no promise about the size of the string it returns!

2\textsuperscript{nd} arg to `strcpy()` is a char*

1\textsuperscript{st} arg to `strcpy()` is a local char[

We could flag any instance of `strcpy()` where the 1\textsuperscript{st} arg is a local fixed array and the 2\textsuperscript{nd} arg is a pointer.
Other Test Cases

STR31-C has many other positive and negative examples, which we could include when testing our rule.

- Can we test them all?
- Will our idea of checking `strcpy()`’s arguments work on them?
- If not, how can we check them
Non-Compliant Code Example: (off-by-1)

```c
char dest[ARRAY_SIZE];
char src[ARRAY_SIZE];
size_t i;
/* ... */
for (i=0; src[i] &&
     (i < sizeof(dest)); i++) {
    dest[i] = src[i];
}
dest[i] = '\0'; /* ... */
```
Non-Compliant Code Example: (strcpy())

```c
int main(int argc, char *argv[]) {
    /* ... */
    char prog_name[128];
    strcpy(prog_name, argv[0]);
    /* ... */
}
```
Compliant Code Example: (strcpy_s())

```c
int main(int argc, char *argv[]) {
    /* ... */
    char * prog_name;
    size_t prog_size;
    prog_size = strlen(argv[0])+1;
    prog_name = (char *)malloc(prog_size);
    if (prog_name != NULL) {
        if (strcpy_s(prog_name, prog_size, argv[0])) {
            /* Handle strcpy_s() error */
        }
    } else {
        /* Couldn't get the memory - recover */
    }
    /* ... */
}
```
Testing Conclusions

• We can’t handle the off-by-1 example with our design at all.
• Our current design will work on the strcpy() example without any modifications.
  • We should add the strcpy() example to our test suite, in `test/c.ncce.wiki.STR.c`
• Our design won’t work on the strcpy_s() example, but we could always extend it to recognize the arguments to strcpy_s() as well as strcpy().
  • We should note this as a task to be done later.
Checker Design for STR31-C

1. Traverse AST.

2. For each `strcpy()` function call
   1. Get both arguments to `strcpy()`. If
   2. 1st argument is a variable AND
   3. the variable’s type is a fixed-length array AND
   4. 2nd argument’s type is NOT a fixed-length array
      • Report a violation of STR31-C!
Design Limitations

- Will report all cases of `strcpy(char[], char*)`, including false positives.
- Will not report any other cases of `strcpy()`, including false negatives.
- Will not catch other string-copy functions like `strncpy()`, `strcpy_s()`, or `memcpy()`.
- Will not catch string-copying done ‘by hand’ (for instance, our off-by-1 example)
Design Conclusions

- Designing checkers helps to ‘flesh out’ secure coding rules.
- Be aware of
  - false positives
  - false negatives
- A checker does not need to be complete to be useful.
- It’s OK to write more than one checker for a rule.
- Don’t worry about pathological cases, focus primarily on violations likely to occur ‘in the wild’.
ROSE in Action

When we run our ROSE program, called \texttt{diagnose}, on our insecure source code, we get an error message:

\begin{verbatim}
% ./rosecheckers test/c.ncce.wiki.STR.c
c.ncce.wiki.STR.c:7: error: STR31-C: String copy destination must contain sufficient storage
%
\end{verbatim}

If we run \texttt{rosecheckers} on a secure program, we get no output:

\begin{verbatim}
% ./rosecheckers test/c.cce.wiki.STR.c
%
\end{verbatim}

So our \texttt{rosecheckers} program acts like a compiler, or \texttt{lint}.
ROSE integrated with Emacs
```
#include <string.h>
#include <stdlib.h>

int main() {
    /* ... */
    char buff[256];
    strcpy(buff, getenv("EDITOR"));
    /* ... */
    return 0;
}
```
Source Code Syntax Tree

The command

```bash
cpp2ps -t foo.c foo.c.ps
dot2ps foo.c.dot
```

produce a PostScript file `foo.c.dot.ps` that contains the Abstract Syntax Tree (AST) of the source code.

On rosebud, the `gv` program can be used to view PostScript files.

```bash
gv foo.c.dot.ps
```
Abstract Syntax Tree 1

```c
#include <string.h>
#include <stdlib.h>

int main() {
    /* ... */
    char buff[256];
    strcpy(buff, getenv("EDITOR"));
    /* ... */
    return 0;
}
```
Abstract Syntax Tree 2
Abstract Syntax Tree 3

[Diagram of an abstract syntax tree]
AST Attributes

The command

    cpp2pdf foo.c

produces a PDF `foo.c.pdf` that contains the source code AST, and also shows each class's attributes

On rosebud, the `xpdf` program can be used to view PDFs.

    xpdf foo.c.pdf
AST Attributes cont.
Whole Syntax Tree

The AST does not contain semantic information, such as:

- Type Definitions
- Symbol Tables
- Variable Definitions

The Whole AST adds these bits of information to the AST.
char* strcpy(char*, char*);
char* getenv(char*);

int main() {
    char buff[256];
    strcpy(buff, getenv("EDITOR"));
    return 0;
}
Whole Syntax Tree 3
Whole Syntax Tree 4
How rosecheckers Uses ROSE

```c
#include "rose.h"
#include "utilities.h"

int main( int argc, char* argv[]) {
    SgProject* project = frontend(argc,argv);
    ROSE_ASSERT( project );
    visitorTraversal exampleTraversal;
    exampleTraversal.traverseInputFiles(
        project, preorder);
    return 0;
}
```

ROSE parses source code

Traverse AST, examine each node.
AST Node Analysis

This is called for each node in the AST:

```c
bool EXP(const SgNode *node) {
    bool violation = false;
    violation |= EXP01_A(node);
    violation |= EXP09_A(node);
    violation |= EXP34_C(node);
    return violation;
}
```

Similar code exists for other sections STR, MEM, etc.
ROSE Checker Skeleton

```c
#include "rose.h"
#include "utilities.h"

bool STR31_C(const SgNode *node) {
    // ensure sufficient storage for strings
    /* ??? */
}

bool STR(const SgNode *node) {
    bool violation = false;
    /* ... */
    violation |= STR31_C(node);
    return violation;
}
```

This routine will be called for every node in the AST. We want it to print an error message and return `true` exactly once when run on our non-compliant example.
Traverse AST.  For each `strcpy()` function call

1. Get both arguments to `strcpy()`. If
2. 1st argument is a variable AND
3. the variable’s type is a fixed-length array AND
4. 2nd argument’s type is NOT a fixed-length array
   - Report a violation of `STR31-C`!
Utility Functions from utilities.h

// Returns non-NULL if node is a call of function with given name
const SgFunctionSymbol *isCallOfFunctionNamed(
    const SgNode *node, const std::string &name);

// Returns reference to ith argument of function reference. Dives through typecasts. // Returns NULL if no such parm
const SgExpression* getFnArg(
    const SgFunctionRefExp* node, int i);

void print_error(
    const SgNode* node, const char* rule, const char* desc, bool warning = false);
Traverse AST.
For each `strcpy()` function call

1. Get both arguments to `strcpy()`. If
2. 1<sup>st</sup> argument is a variable AND
3. the variable’s type is a fixed-length array AND
4. 2<sup>nd</sup> argument’s type is NOT a fixed-length array
   - Report a violation of STR31-C!

At this point, `node` will always point to a `strcpy()` function call
The isSg Family

A set of useful functions that are useful for typecasting a `SgNode*` into an appropriate node type. They return `NULL` if the node is the wrong type.

```c
const SgNode* node;
const SgFunctionRefExp* sig_fn
    = isSgFunctionRefExp( node);
if (sig_fn == NULL) {
    cerr << "Node is not a "
         << "SgFunctionRefExp!" << endl;
}
```
bool STR31_C(const SgNode *node) {
    if (!isCallOfFunctionNamed(node, "strcpy"))
        return false;
    const SgVarRefExp* ref = isSgVarRefExp(getFnArg(isSgFunctionRefExp(node), 0));
    if (ref == NULL)
        return false;
/* ??? */
}

Traverse AST.
For each `strcpy()` function call
1. Get both arguments to `strcpy()`. If
2. 1st argument is a variable AND
3. the variable’s type is a fixed-length array AND
4. 2nd argument’s type is NOT a fixed-length array
   • Report a violation of STR31-C!

At this point, ref refers to the 1st arg of `strcpy()` and it is a variable.
bool STR31_C(const SgNode *node) {
    if (!isCallOfFunctionNamed(node, "strcpy"))
        return false;

    const SgVarRefExp* ref =
        isSgVarRefExp( getFnArg( isSgFunctionRefExp(node), 0));
    if (ref == NULL)
        return false;
    if (!Type( getRefDecl( ref->get_type()).isArray()))
        return false;

    /* ??? */
}

Traverse AST.
For each `strcpy()` function call

1. Get both arguments to `strcpy()`. If
2. 1st argument is a variable AND
3. the variable’s type is a fixed-length array AND
4. 2nd argument’s type is NOT a fixed-length array
   • Report a violation of STR31-C!
const SgVarRefExp* ref =
    isSgVarRefExp(getFnArg(isSgFunctionRefExp(node), 0));
    if (ref == NULL)
        return false;
    if (!Type(getRefDecl(ref)->getType()).isArray())
        return false;
    if (Type(getFnArg(isSgFunctionRefExp(node), 1)->getType()).isArray())
        return false;

Traverse AST.
For each `strcpy()` function call
1. Get both arguments to `strcpy()`. If
2. 1st argument is a variable AND
3. the variable’s type is a fixed-length array AND
4. 2nd argument’s type is NOT a fixed-length array
   • Report a violation of STR31-C!
DONE
ROSE Checker for STR31-C

```c
#include "rose.h"
#include "utilities.h"

bool STR31_C(const SgNode *node) {
    // ensure sufficient storage for strings
    if (!isCallOfFunctionNamed(node, "strcpy"))
        return false;

    const SgVarRefExp* ref =
        isSgVarRefExp( getFnArg( isSgFunctionRefExp(node), 0));
    if (ref == NULL)
        return false; // strcpy() not copying into simple var
    if (!Type( getRefDecl( ref)->get_type()).isArray())
        return false;
    if (Type( getFnArg( isSgFunctionRefExp(node), 1)->get_type()).isArray())
        return false;

    print_error( node, "STR31-C", "String copy destination must contain sufficient storage");
    return true;
}
```

Called for every node in the AST.

We have an instance of `strcpy()`

1st arg is a local fixed array

2nd arg is a pointer (eg NOT an array)
Build and Test

To rebuild `rosecheckers` with a new rule, type

```
make pgms
```

To test `rosecheckers` on all rules, type

```
make tests
```
Testing New Rule

When run on the bad example, rosecheckers produces an error message:

```
% ./rosecheckers test/c.ncce.wiki.EXP.c
EXP.c:5: error: EXP09-A: malloc called using something other than sizeof()
%
```

When run on the good example, rosecheckers produces nothing.

```
% ./rosecheckers test/c.cce.wiki.EXP.c
%
```
Useful ROSE Functions

- `isSg*** (node)`
- `unparseToString()`
- `querySubTree(SgNode* node, type)`
unparseToString()

Returns a string representation of the source code associated with a node. Useful for debugging:

```cpp
const SgNode* node;
cout << "Node: "
    << node->unparseToString() << endl;
```
querySubTree(node, type)

Traverses the AST that descends from node. Returns a list (a std::vector, actually) of all subnodes of appropriate type.

```cpp
const SgNode* node;
Rose_STL_Container<SgNode *> nodes = NodeQuery::querySubTree(
    const_cast<SgNode*>( def), V_SgVarRefExp);
Rose_STL_Container<SgNode*>::iterator i;
for (i = nodes.begin(); i != nodes.end(); ++i) {
    cout << "A SgVarRefExp: " << (*i)->unparseToString() << endl;
}
```

Note that `querySubTree` requires a non-const SgNode* as 1st argument.