Bachelor's Thesis : Finding Bugs in Open Source Software using Coccinelle

Sune Rievers - sunerievers@stud.ku.dk
Supervisor: Julia Lawall

January 13, 2010

Contents

1 Abstract

Coccinelle is a framework developed for semantic patching of C code, to allow automated transformations of code. Originally this was done with regards to driver implementations in Linux kernel code.

Coccinelle could possibly be used for other uses as well, e.g. finding bugs in software. I have been examining this, by applying Coccinelle to the Open Source security project OpenSSL.[1] I have compared the results of Coccinelle to results from other software, to determine Coccinelle's applicability towards finding bugs.

As others have experienced,[2] Coccinelle is perfectly applicable for finding bugs, and I have found bugs using Coccinelle which other static analyzers have not.

Compared to other related software, Coccinelle has been able to find 37 defects in OpenSSL, where the commercial product Coverity Prevent has found 36. This indicates that Coccinelle is indeed very suitable for bug finding.

All bugs have been reported to the OpenSSL community, after thorough manual inspection. I have received some feedback, but none of my patches has been applied to OpenSSL at the time of concluding this report.

[1] More information on OpenSSL can be found in Section 7 and on http://www.openssl.org

[2] The Coccinelle web site has a list of bugs found and patches applied to various software, amongst other the Linux kernel.

2 Preface

This report concludes the 15 ECTS Bachelor's thesis course at DIKU.[3]

The author is Sune Rievers, student of Computer Science at DIKU. The supervisor is Julia Lawall, an associate professor working in the APL group at DIKU.

3 Acknowledgments

I would like to thank my supervisor, Julia Lawall, who has been available for many helpful comments and guidance, even at weekends and late at night during the course of this project. Also I would like to thank the Coccinelle team, especially Nicolas Palix for advice regarding Python and the scripting of semantic patches. The process of being involved in a project like this has been very educational, and I hope that this report may help Coccinelle in any way possible.

[3]DIKU stands for "Datalogisk Institut Københavns Universitet" or Institute of Computer Science at University Copenhagen.

4 Introduction

It is a well known fact that software is full of bugs, and even though the source code of Open Source software is potentially read by more people than closed source, a lot of bugs remain undetected. Furthermore, when a bug is found, it is hard to locate similar bugs unless their syntax matches exactly.

4.1 Background and motivation

Using Coccinelle, a tool developed at DIKU and the Ecole des Mines de Nantes for semantic patching of C code, it is possible to use semantic patches [4] to search for code defects. Each case (defect type) has to be specifically tailored by the user into semantic patches.

4.2 Problem definition

Coccinelle has been designed for doing matching and transformations in Linux code, and therefore it is interesting to know whether it can be applied to other types of software as well. If the project is a success, it might open Coccinelle for bug finding in all sorts of software.

The main question we need to answer in this project, is: Is it possible to apply Coccinelle to other uses than Linux Kernel code, and what is the reward of doing this?

4.3 Scope

I have put my emphasis on analyzing bugs and writing Coccinelle scripts, as I have seen this as the main purpose of this project. Analyzing Coccinelle and its ability to find bugs using semantic searching is also very important.

Fixing bugs has been a minor part of the project, and has not originally received as much attention as locating them. I have however spent quite some time learning the syntax of Coccinelle, and also in writing and modifying semantic patches for finding bugs.

4.4 Intended audience

Readers of this report should be familiar with basic C programming, types of code defects, and some compiler theory, like memory allocation and use of pointers. Students of Computer Science would be an ideal audience, at

[4] The scripting language of Coccinelle uses files called semantic patches. See Section 5.3 for more information on semantic patches.

least ones that have completed basic courses in compiler theory and machine architecture.

5 Analysis of Coccinelle

5.1 General description of Coccinelle

Coccinelle deals with the problem of collateral code evolutions [8]. When an interface changes (e.g. for a driver or an API function), there are multiple places where code needs to be adapted to the new interface or coding style. These adaptations are collateral evolutions. Using Coccinelle it is possible to create scripts called semantic patches (SP), that perform these transformations.

It is also possible to use semantic patches to find code defects in software, as this is basically a pattern matching activity. The Coccinelle web site [8] has many examples of bugs found using Coccinelle. Users of Coccinelle have for example found and corrected many defects in Linux kernel code, and these patches have been applied to the official kernel tree.

5.2 Static analysis

When finding code defects in software, one can try several approaches: Testing live code (debugging), manual inspection of source code and static analysis. Coccinelle deals with the latter.

Static analysis is a type of analysis that does not need the code to be compiled and executed to be analyzed. It can be used on partial code, code that needs Internet access to work, and code that is too dangerous to actually compile and run (e.g. virus or malware software). While testing only reveals defects in the actual executed code paths, static analysis can potentially find defects in all code paths, even infeasible ones. Static analysis can be used to detect defective code constructs, as well as improper use of API functions, invalid naming of variables according to a coding standard and much more.

Static analysis can be automated, for instance, running in an interactive build script or during a nightly build operation. The results can then be emailed to the person responsible for manual inspection. Testing and manual code inspection needs an actual person to perform the work and is very time consuming. Furthermore, testing can only be applied rather late in the process (when the code actually compiles and runs). Of course, static analysis can also be used as a stand alone tool, like testing.

5.3 Cocci files (scripting)

The language in which Coccinelle semantic patches are written, is called the Semantic Patch Language (SmPL). SmPL has some similarities to the syntax

of standard patches,[5] with added temporal logic and support for metavariables. Metavariables are a kind of variables in a semantic patch, which act as placeholders for normal variables, functions and expressions. This means, for instance, that one can match any given function by using a metavariable, and then refer to that same function later on in the semantic patch. See Figure 1 as an example of a semantic patch using a metavariable. In this figure, E is a placeholder for an arbitrary expression, that will be removed along with the call to free and inserted with a new call to OPENSSL_free. [6]

```
@@
expression E;
@@

- free(E);
+ OPENSSL_free(E);
```

Figure 1: Part of malloc_style.cocci

Metavariables make it possible to abstract over irrelevant subterms, making it possible, e.g. to find all functions that have a particular structure or usage pattern. Similarly for variables, you could for instance find all occurrences of a variable of a certain type or one that is never assigned a value, or is never used like in the unused SP. In other words, one is not restricted to the basic adding and removal of text lines like in standard patches.

SmPL has been extended with Python support by Henrik Stuart as part of his Master's Thesis [14]. This extension means that not only is it possible to modify code using + and − for adding and removing code patterns,[7] scripting is also possible, for instance to allow the results to be filtered or inserted in a database for further analysis.

SmPL is quite an extensive language, and is described in full on the web site, including both a grammar and a tutorial [8], which is why I have only given a brief summary in this report.

As many code constructs are semantically similar, for instance $i = i + 1$ is the same as $i + +$, Coccinelle has a isomorphism method that is a form

[5] A standard patch is a file used for expressing the difference between two sets of text, e.g. two different versions of a source code file. Standard patches are in general just called patches, but I will call them standard patches in this report to avoid confusion with semantic patches.

[6] See Appendix B.3 for the full semantic patch.

[7] Like in a standard patch, where + and − signifies the adding and removal of code lines.

of macro expansion, so that all possible code is checked, when searching for one out of several semantically similar code expressions.

6 Bug classification

In order to deal with code defects in a systematic and uniform way, a taxonomy is often used. A taxonomy is a classification scheme, which can be used to organize or order objects of its domain. There are many examples of these, but I have chosen the CWE standard [10], as it seems to have widespread support in industry. Also I think it is important to be able to have a standard method of classification.

In Section 7.2, I used the CWE as basis for some statistical analysis, in order to determine which defect types are most dominant in OpenSSL.

The CWE has also given me some inspiration for creating patches. For instance, on the CWE web site there is a list of common weaknesses in C programming [11]. This has been partially used for examples of use after free in my semantic patch use_after_free.

Many of the defect types from the CWE C specific list are already covered by the generic patches on the Coccinelle web site, but there are also a lot that aren't covered by the generic patches, for instance vulnerabilities about temporary files and Windows-specific code defects.

7 Description of OpenSSL

I have chosen OpenSSL [8] as the subject of my evaluation, as it is both Open Source and widely used.

OpenSSL is a library and a toolkit, and is used in a lot of Open Source software, including email servers, web servers and VPN software.

The OpenSSL project was started in 1999, and is based on SSLeay by Eric Young. Eric Young still hold copyright to parts of the code, which is noted several places in the OpenSSL source code. If someone is to use OpenSSL in a software project, Eric Young's original license still applies, and must be included with the new software.

The OpenSSL development team consists of eleven developers, of which four are on the core team and manage the OpenSSL project. The core team also coordinates the mailing lists, as well as OpenSSL announcements.

As a version control system, OpenSSL uses CVS, which has read-only access for anonymous users. I have used the CVS repository as basis for my bug finding, as well as their official releases.

Patches are generally submitted to the OpenSSL development mailing list, and in some cases also to the public Request Tracker.[9]

7.1 Why this software project is interesting

I learned that OpenSSL contained a lot of bugs, and since I knew that it was used in a wide range of software, I thought it would be useful and interesting to help out, and to learn about the various bug types in OpenSSL. I am also a big fan and supporter of Open Source, and would like to improve this by finding and fixing code defects in Open Source software.

7.2 Particular bugs in project

OpenSSL has a public list of fixed vulnerabilities on their web page, based on CVE reports [12]. I have looked at this list, and classified them according to the CWE [10]. The list is in Figure 9 in Appendix B.1.

The most predominant defect type is "CWE 20: Improper Input Validation", which represents 14 of the 31 CWE classifications on my list. In second and third place are NULL pointer dereference and buffer overflow. The rest are in very different categories, ranging from highly domain-specific bugs such as "CWE-347: Improper Verification of Cryptographic Signature" to more common ones like "CWE-415: Double Free".

[8] More information on OpenSSL can be found on http://www.openssl.org

[9] OpenSSL has a public list of bugs on http://rt.openssl.org/NoAuth/Buglist.html

8 Bug hunting

8.1 Using pre-made semantic patches

The people behind Coccinelle has created some generic bug-finding semantic patches, which I have used as the starting point of my project. Some of these are Linux or kernel specific, but I have only dealt with the truly generic ones. A brief overview of these is as follows.

The badzero SP deals with noncompliance of best practice, where pointers are compared to literal 0 instead of NULL. This semantic patch returned five cases on the current release version, and no false positives.[10] These were submitted to the OpenSSL community. Later, I ran it on the CVS version, which gave a sixth result. This was also submitted. An example of a bug found with this SP is in Figure 2.

```
-   if ((m == 0) || (r == 0) || (f == 0))
+   if ((m == NULL) || (r == NULL) || (f == NULL))
  return 0;
```

Figure 2: Example of a badzero bug

Most rewarding has been the notnull SP. The notnull SP deals with a NULL test on an already tested (known to be not-NULL) value, which is redundant. It has found 11 defects in a release version of OpenSSL, of which none were false positives.

The find_unsigned SP has found the same error in multiple versions of OpenSSL, as shown in Figure 6. This is a case where an unsigned integer is tested for a value below zero, which of course is pointless. It seems that the reason for using an unsigned type is because the variable is assigned a value from a function that returns an unsigned value, so the variable can never have a value below zero. The check below zero i probably left over from a point where the variable were assigned differently, or perhaps the function had a different return type in the API. On the other hand, if an unsigned variable was used, and the called function were to return a negative value in case of error, this would indicate a serious problem. This bug has also been submitted to the OpenSSL community.

The andand SP is a very simple case that detects expressions where && and || are mixed up, for instance in these constructs:

[10] A false positive is when valid code is incorrectly detected as defective. See Section 8.5 for more information about dealing with false positives.

```
a) if (!E && E->fld) ...
b) if (E || E->fld) ...
```

In these cases, the author probably meant to replace && with || in a, and vice versa in b. Otherwise, E->fld would be derefenced only in cases where E is NULL.

The isnull SP is a bit like andand, where an expression is checked to be NULL and subsequently dereferenced. This could for instance be an if-statement that checks if a variable is NULL, and in that case dereferenced. This could lead to some serious errors, and as Figure 3 shows, I found an example of this defect in the CVS version of OpenSSL.

Null_ref is a semantic patch that detects a dereference followed by a NULL check, when this should have been reversed. The mini_null_ref mini_null_ref2 are simpler versions of this test, dealing with special cases of null_ref. These semantic patches search for a more specific, localized pattern, so they have lower rate of false positives. As they are simpler, they are also faster to run, and easier to analyze. I have found no matches in any of my tests, so I assume that they are too Linux specific, or deal with code constructs unfamiliar to OpenSSL.

I have used the generic semantic patches from the Coccinelle website, and applied these to the release version of OpenSSL, as well as the CVS one. Results are displayed in Figure 3 and 4.

Cocci script	Bugs found	False positives
andand	0	0
badzero	5	0
find_unsigned	0	0
isnull	0	0
mini_null_ref	0	0
mini_null2_ref	0	0
notand	0	0
notnull	11	0
null_ref	0	0

Figure 3: Results from running the generic semantic patches on the 0.9.8j version of OpenSSL

The null_ref SP reported a false positive, as shown in Figure 5. If tree was NULL, the function would have returned immediately after the call to OPENSSL_malloc. However, the SP sees the assignments of tree's properties

Cocci script	Bugs found	False positives
andand	0	0
badzero	6	0
continue	2	0
find_unsigned	1	0
isnull	1	0
mini_null_ref	0	0
mini_null2_ref	0	0
notand	0	0
notnull	10	0
null_ref	5	1

Figure 4: Results from running the generic semantic patches on the CVS (rev. 17904) version of OpenSSL

as dereferences, and the following NULL check as an improper order, and detects this as a defect. This is in fact a not_null bug, and has also been detected by the not_null SP.

Regarding false negatives,[11] there are probably some, since I would not assume that my semantic patches have found all types of every bug tested for. There could very well be some defects that were not found, but these might be found using other SP's or even other tools. Regarding the pre-made SP's from Coccinelle web site, I assume they have a rather low rate of false negatives. For instance, the badzero and find_unsigned SP's are so simple and concise, that I would not expect them to have any false negatives at all.

In each case, I have submitted my findings to the OpenSSL mailing list, after manual verification. The first report did not receive much attention, presumably since I used the release version of OpenSSL as target of my scanning, so many of the bugs I found had already been fixed in the CVS (development) version of OpenSSL.

Therefore, I downloaded the latest CVS version, and reran my tests on this version. Some bugs were however still present, so I resubmitted those to OpenSSL, and stated that I had used the current CVS version, to get more attention this time.

[11] A false negative is when defective code is incorrectly labeled as valid code, or the opposite of a false positive.

```
tree = OPENSSL_malloc(sizeof(X509_POLICY_TREE));

if (!tree)
   return 0;

tree->flags = 0;
tree->levels = OPENSSL_malloc(sizeof(X509_POLICY_LEVEL) * n);
tree->nlevel = 0;
tree->extra_data = NULL;
tree->auth_policies = NULL;
tree->user_policies = NULL;

if (!tree)
   {
      OPENSSL_free(tree);
      return 0;
   }
```

Figure 5: Example of null_ref false positive

8.2 Bug type analysis

I have looked through the CVS commit comments, the website and the mailing list to determine which kind of bugs are common in OpenSSL. I have also done some statistical analysis, in order to know which bugs are dominant in OpenSSL, this was described in Section 7.2.

I discovered from reading the OpenSSL FAQ and mailing list archives that Valgrind, Purify and Coverity have already been applied to OpenSSL [12], so a lot of the obvious errors have probably already been fixed, which I assume is the reason that my initial bug count was relatively low.

However, OpenSSL is still a living product, and new bugs are introduced constantly. For instance, the badzero SP found one more bug in the CVS and the beta release than in the 0.9.8j stable release.

There are a lot of application-specific issues on the mailing list, e.g. when someone is adapting OpenSSL or using it in a specific project. These seem to be very specific to the task at hand, and not general defects as such.

I also noticed that there are a lot of stale bug reports on the OpenSSL issue tracker system, some of which have not been updated for 5 years. I assume that this is because they are either not relevant anymore, or because

[12] See Section 10.1 on page 25 for more information about the effort with Coverity

```
unsigned int ret;
if (ret < 0)
{
  IBMCAerr(IBMCA_F_IBMCA_RAND_BYTES,
    IBMCA_R_REQUEST_FAILED);
  goto err;
}
```

Figure 6: Example of a find_unsigned bug

they are invalid (i.e. not a bug or not reproducible). On the mailing list, there is a lot of focus on getting the critical bugs fixed, and mails about memory leaks or security issues are dealt with promptly and seriously.

As mentioned in Section 7.2, most of the critical bugs deal with input validation and NULL pointer dereferencing. Input validation is very specific and needs special care in handling each case, whereas NULL pointer dereferencing could be detected using the null_ref or isnull SP's from the Coccinelle web site.

8.3 Creating new scripts

After running the pre-made semantic patches, I wanted to find more types of defects, so I created the following new Coccinelle scripts (The scripts are shown in full in Appendix B.3):

- malloc_style
- use_after_free
- openssl_malloc_free
- openssl_malloc

As the figures show, there are not many false positives, if any. The malloc_style SP in Figure 7 has an unknown rate of false positives, as I have not been able to find documentation of where OPENSSL_malloc should be used, and where the general malloc function should be used.

I went through the OpenSSL source code to see if any special OpenSSL API functions were used. I noticed that OpenSSL uses some wrapper methods for memory allocation (malloc) and deallocation (free) called OPENSSL_malloc and OPENSSL_free, so I took two malloc-related semantic patches from the

Cocci script	Bugs found	False positives
andand	0	0
badzero	6	0
continue	2	0
find_unsigned	1	0
isnull	0	0
mini_null_ref	0	0
mini_null2_ref	0	0
notand	0	0
notnull	7	0
null_ref	0	0
malloc_style*	19	?
openssl_malloc*	1	0
use_after_free*	2	1

Figure 7: Results from running the generic and custom semantic patches on the beta2 version of OpenSSL. Semantic patches generated by me are marked with a *.

Coccinelle web site and modified these to fit the OpenSSL wrapper methods and error handling:

Malloc is a semantic patch that detects a case where a pointer is not freed upon returning of the function, in this case I exchanged calls to malloc and free with the OpenSSL equivalents[13]. In OpenSSL, it is common to have an error label in large functions, that when an error occurs, is jumped to via a goto statement. In many cases this can lead to leaks, as allocated memory is not always freed upon a return. Therefore I added a case where a goto was used. Before each return in the function I added the missing call to OPENSSL_free, so that the variable would always be freed. This semantic patch has been named openssl_malloc.[14]

Malloc_free is a semantic patch that detects a case where an allocated variable is not freed upon an error return. In this case I also replaced the malloc and free function calls like in the malloc SP. As openssl_malloc already deals with the error label path, this SP only deals with real error conditions. This semantic patch has been named openssl_malloc_free.[15]

[13]malloc is replaced with OPENSSL_malloc and free is replaced with OPENSSL_free to fit the OpenSSL standard way of allocating memory

[14]See Appendix B.3 for this semantic patch.

[15]See Appendix B.3 for this semantic patch.

This detected some cases where memory was not freed when returning on error conditions and otherwise.

I then modified the OPENSSL_malloc and OPENSSL_free patches, so memory would be always freed before returning from the function.

Regarding use after free, I had some matches (6 when run on beta1 of OpenSSL). On closer inspection I noticed that there were a lot of cases where my patch had found this structure:

```
free(ptr);
ptr = NULL;
```

This is clearly not a use after free bug, but merely following good coding style. If the pointer is not set to NULL, subsequent calls to free can lead to an error condition, as the memory may be freed or occupied by other data. If the pointer like in this case is explicitly set to NULL, the call to free will have no effect. This is also recommended as best practice in the CERT C Secure Coding Standard [13]. There is also the point that dereferencing a NULL pointer only leads to a system or program crash, while dereferencing a freed pointer (possibly pointing to other data), could be a serious security issue. This is a very important issue as OpenSSL is a piece of security software. Of course, the potential double free should be detected and removed, as this would be redundant.

Therefore I added this structure to my use_after_free semantic patch after each new call to free, which removed all of the false positives. Unfortunately there were no real positives, so this SP detected zero defects on this version of OpenSSL.

```
+ E = NULL;
```

8.4 Reception from the community

The OpenSSL community has not been overly positive in reception of my patches, some have been ignored, and others have been deemed redundant. For instance, I submitted a patch with extra null tests (from using the notnull SP), and this was deemed an unnecessary change, as they like to keep to standard ways of doing things, and removal of redundant checks has little value over standards compliance.

After generating the custom scripts shown in Section 8.3, I submitted them to the OpenSSL mailing list.

Both submissions have yet to to implemented in the OpenSSL source tree, and I have not received any more replies to my submission, either on the mailing list or in the request tracker.

8.5 Dealing with false positives

False positives, or when valid code is incorrectly marked as defective, is a huge problem dealing with automated code defect analyzers. If you want to eliminate false positives, you must be willing to spend more time writing precise semantic patches or let the analyzer weed out the false positives. As mentioned in [4], "Unfortunately precision usually depends on analysis time. The more precise the analysis is, the more resource consuming it is, and the longer it takes. Hence, precision must be traded for time of analysis". False positives have to be dealt with by hand, which is tedious and time consuming, or by using a heuristic approach like Coverity does. The comparative study in [4] states that it is likely that Coverity Prevent uses a probabilistic technique involving Bayesian learning. As Coverity Prevent is closed source, it is not possible to get total insight in this filtering technique. When false positives always have a distinct structure, for instance in the case of my use after free SP (where the pointer was set to NULL after the call to free, instead of being assigned anew), it is possible to filter these results by adding this case to the SP. This does not cause false negatives, as long as the filtering is done very carefully.

If the static analyzer or bug finding system deals with the false positive in some way, i.e. by filtering the false positives, the matches should not be labeled as such, as the user is not presented with the detected false positive. This approach can however still result in a false negative, if the filtering is done wrong, and effectively hiding defective code. When using Coverity's approach, which uses aggressive filtering, there are probably false negatives, as the user does not in general create the scripts used for defect finding, and therefore does not know which false positives are filtered.

9 Evaluation of Coccinelle

9.1 Usage

Regarding the trivial bugs, such as dereferencing a NULL pointer, or comparing a pointer to literal zero instead of NULL, Coccinelle is outstanding. The resulting standard patches from my test contained no false positives. With regards to more complicated bugs, using Coccinelle is harder, because writing these semantic patches is more complicated, but judging from running the premade-scripts, Coccinelle is highly usable for this purpose as well.

When properly trained in using Coccinelle and the SmPL syntax, it is possible to create semantic patches that detect very complex code defects, spanning many lines of code, and finding defects that are virtually impossible to find by means of manual code inspection. Simple patches like the notnull semantic patch, has for instance detected this defect in OpenSSL, where a redundant NULL check could be removed. The first and second NULL check were almost 500 lines apart. This would have been almost impossible to find by hand, or at least very time consuming. The defect is shown in Figure 8.

```
1693: tmptm=ASN1_UTCTIME_new();
1694: if (tmptm == NULL)
1695: {
1696: BIO_printf(bio_err,"malloc error\n");
1697: return(0);
1698: }
...
2192: if (tmptm != NULL)
2193:  ASN1_UTCTIME_free(tmptm);
2194:  if (ok <= 0)
```

Figure 8: Example of a notnull bug: apps/ca.c do_body function

9.2 Usability

There exists a grammar of the Coccinelle semantic language, but for people without extensive compiler knowledge (e.g. students of computer science), this is probably too technical. There is also a tutorial on using SmPL on the web site [8], to allow beginners to learn how to use Coccinelle. There has also recently been established a wiki,[9] so that users can help each other generating helpful content and tips on the SmPL language.

The results from running Coccinelle are either standard patch files in unified diff, or text files with notes of problematic code lines (generated by the Python extension). The generated standard patch files are directly applicable, but contain little information about the actual bug found. The patches could be augmented with notes about the possible code defect, but this might not be read, or introduce too much static. The Python generated files can contain any information (depending on the actual Python code), but in general only line numbers are written.

While running Coccinelle, the user is not presented with an estimate of how long the process would take, or a file by file progress. This would help the user in knowing whether to let the scan run overnight or whether it could be used almost interactively (for instance in a makefile or similar).

During the installation of Coccinelle on my Debian server I ran into some problems caused by an incompatible version of Python, which resulted in many wasted hours and a lot of mail correspondence with the Coccinelle authors. This could probably have been avoided if more work was spent on testing Coccinelle on various platforms, as well as different hardware. I have been told by Julia Lawall from the Coccinelle team that the installation procedure has been improved in the latest release, but I have not yet been able to confirm this myself.

9.3 Technical results

Even though OpenSSL has been tested with loads of static analyzers and programs like Valgrind, Coccinelle has found code defects and programming style errors, which the others did not (or the OpenSSL team disregarded as such). This information is from the OpenSSL mailinglist as well as CVS commit comments, both if which note several of the previously mentioned analyzers and checkers.

Regarding Coverity's scanning of OpenSSL,[16] Coccinelle has found as many defects as Coverity Prevent, and the severity appears to be similar. This indicates that Coccinelle is comparable to a commercial product that has been developed on for more than ten years, which I think is pretty impressive, since Coccinelle only has a handful of developers and has been active for just a few years.

It might be limiting for Coccinelle that it only deals with C code, but I would assume that it would be possible to create support for more C-like languages in the future, when more resources are available for this purpose. Coverity Prevent also started with only C code, and now it has support for

[16] See Section 10.1 on page 25 for more about Coverity's Open Source initiative

C++, C# and Java as well.

10 Related work

There are quite a few projects that deal with static code analysis, but the following seem to be the most interesting (only tools dealing with the C programming language are listed). According to [4], Coverity, PolySpace and K7 are the three market leaders. The study in [4] is from January 2008, so they are probably still in wide use, if not still in the top.

As most of these tools are commercial and closed-source, I have not been able to evaluate them first hand. Because of this, I have used the available documentation and the comparative study [4] as basis for my comparison.

- **Coverity Prevent**
  A commercial tool.
- **PolySpace Verifier**
  A commercial tool, used for testing embedded systems.
- **K7 Klocwork**
  A commercial tool.
- **CP-Miner/PR-Miner**
  Two academic tools, used to find examples of copy/paste code and incorrect usage of API functions.

10.1 Coverity Prevent

Coverity Prevent from Coverity is a commercial tool. I will refer to Coverity Prevent as CP in the remainder of this section.

CP started as an academic research project by Dawson Engler and some of his students at Stanford University in 2002 [5], and has since grown to a company with more than 150 employees and 600 clients, according to the Coverity web page [2].

As CP is proprietary and closed source, it is not easy to get information about the inner workings of the software. I have therefore leaned heavily on the comparative study [4], the initial Coverity paper [5], and for a lesser part, the Coverity web site and information material [1]. I have compared the information on the Coverity web site [2] and the Coverity Prevent product information[1] with the study [4], and there are so many similarities in their descriptions of the software, that I feel it is close enough for comparison to Coccinelle and the other analyzers.

When CP was purely an academic tool, it used a meta language called Metal, which could be used analogously to SmPL in Coccinelle. As a commercial product, Coverity Prevent has a similar language called Coverity

Extend, which was probably rewritten as CP turned commercial. At least the description of Metal and Extend differ on in syntax, as mentioned by [4].

The analysis from CP is not sound nor complete [17] according to the comparative study [4], which states that Coverity Prevent does not report all defects and may have false positives, even though it is stated that Coverity Prevent has 100% code path coverage in their product information [1].

CP does a lot of work of eliminating infeasible code paths and false positives, in order to give the user less code to review. This also means that false negatives are possible, as results are filtered.

CP does not give code patches to eliminate the bugs found, it merely gives the user information that may be helpful in rewriting the code.

Coverity (as it was initially called) originally started as a tool for analyzing C, but the language coverage has grown since then, as CP now supports C, C++, Java and C# according to their latest data sheet [1].

In addition to code defect checkers, CP has also a few security related checkers, like checking for insecure coding techniques, overflows or incorrect use of temporary files.

In a joint effort with amongst others the U.S. Department of Homeland Security, Coverity Prevent has been applied to a broad range of Open Source software, including OpenSSL. With regards to OpenSSL, this initiative led to the finding and fixing of 24 bugs in about 222K LoC, according to the Coverity Scan Web site [3]. According to the CVS commit comments, 36 defects have been fixed based on CP reports, but it is not clear which are from the joint effort. These numbers are however very similar to the 16 to 37 code defects I have found using Coccinelle.

10.2 PolySpace Verifier

PolySpace Verifier (PV) is a sound, flow-sensitive, inter-procedural analyzer [4].

PV is a commercial tool, and can analyze C, C++ and Ada. PV has a MISRA [18] standard compliance checker, and is highly specialized for testing software in embedded systems.

Because PV analyzes all code paths, it is infeasible to use it on very large programs. According to [4] the upper limit for analysis is approx. 50K LoC.

PolySpace has support for "non-trivial relationships between variables"

[17] A complete analyzer has no false negatives, i.e. does not filter results, whereas a sound analyzer has no false positives.

[18] Motor Industry Software Reliability Association - http://www.misra.org.uk/

[4], and is able to reason about variable aliasing [19] and concurrency. While PV has great support for arithmetic analysis, it lacks support for memory handling defects, such as overflows and leaks. Since MISRA disallows dynamic memory allocation, this is not really a problem. However, if applied to software projects such as OpenSSL, the code defect detection rate will probably be quite low compared to e.g. Coverity Prevent because of this limitation in PV.

The results from PV are in the form of warnings of different colors/degrees, ranging from green to red according to the possibility of code defects. Dead code is marked with gray.

There are no security checkers in PV.

10.3 Klocwork K7

Klocwork K7 supports C, C++ and Java, has support for some aliasing and has an unsound inter-procedural analysis.

Like Coverity Prevent, K7 has some dedicated security checkers. For instance, K7 has checkers that find code that is syntactically valid, but that due to possible misspellings or mistypings could be insecure. This could for example be an assignment in a condition, which could lead to some hard to find bugs, or using insecure function calls.

There are also checkers that find use of poor, unsafe encryption algorithms. As this seems to be too complex for a static analyzer, I would assume that this is based on known API functions, string-based search (e.g. functions with "md5" "rot13" in their names) or some sort of heuristic approach.

K7 has also been applied to some Open Source projects, and it is claimed that K7 has found some defects that Coverity Prevent did not [4].

Like PV, K7 returns a report with possible code defects, ranked according to severity.

10.4 CP-Miner/PR-Miner

CP-Miner and PR-Miner are two academic tools, created by Zhenmin Li and Yuanyuan Zhou et al. at the University of Illinois in 2004 and 2005.

CP-Miner is a tool for detecting copy-pasted code, and bugs related to copy-paste operations. These bugs could for instance be introduced because a programmer copies a function from another library, but forgets to modify an identifier, or when porting code to another platform, neglects to resolve

[19] Aliasing is when two or more pointers are pointing to the same memory, which can be very hard to detect properly

issues regarding the new platform in the copied code. CP-Miner uses data mining along with a frequent sub-sequence algorithm to detect copy-pasted code, according to the CP-Miner report [6]. CP-Miner can not only detect 100% identical text, but also small variations, thanks to the frequent sub-sequence algorithm.

CP-Miner is limited to very simple types of bugs, as it is impossible to deduct what the programmer has left out or forgotten to implement, when code was copied. Nevertheless, it has shown great results, as it has detected 28 copy-paste related bugs in Linux and 23 in FreeBSD according to the CP-Miner report [6]. Compared to plagiarism tools, which typically has a complexity of $O(n^3)$ when applied to n statements, CP-Miner has a complexity of only $O(n^2)$. This is because of the involved algorithm, and the fact that CP-Miner operates on tokens instead of the entire text, character by character.

PR-Miner is a similar tool, that also facilitates data mining, in order to extract programming rules, and subsequently check for compliance of these rules. As the PR-Miner report [7] writes: "Programs usually follows many implicit programming rules, most of which are too tedious to be documented by programmers. When these rules are violated by programmers who are unaware of or forgot about them, defects can be easily introduced.". As PR-Miner automatically can extract these rules, most of the tedious work is done for the programmer. Furthermore, as rules change over time, PR-Miner can be executed again on the source code, to extract an updated version of the rules. If these rules were maintained by hand, there is a good chance that these changes were forgotten.

PR-Miner uses a data mining technique called frequent itemset mining, which is comparable to the frequent sub-sequence algorithm used in CP-Miner, except frequent itemset mining focuses on the frequency of precise data (e.g. often used code constructs), and frequent sub-sequence focuses on similarity of data.

PR-Miner has like CP-Miner been applied to the latest version of Linux, where 16 has been detected and confirmed. This might not seem as much, but taken into account that PR-Miner is almost fully automatic and fast, it is highly applicable in finding defects that violates proper coding style.

11 Conclusion

Compared to the related software, Coccinelle has found as many defects as Coverity Prevent, and has many of the same features as the others. For instance, there are some semantic patches for Coccinelle that deal with memory allocation and pointer dereferencing.

The severity of the detected defects from Coccinelle are about the same level as the ones found from Coverity Prevent, but I suspect that the OpenSSL community has a higher degree of trust in Coverity Prevent than in Coccinelle. This is probably also why the submitted patches have not yet been integrated into OpenSSL.

There are some areas where Coccinelle could use a hand, for instance regarding the installation, as mentioned in Section 9.3.

Usability-wise, Coccinelle could definitely use some work, as setting it up could be a non-trivial task, as broken dependencies are not always detected.

A Bibliography

References

[1] Coverity, Coverity Prevent Product Information (pdf), Worldwide Web Document (2009). Available at `http://www.coverity.com/library/pdf/coverity_prevent.pdf` ([Online; accessed 5-May-2009].)

[2] Coverity, Coverity Web Site, Worldwide Web Document (2009). Available at `http://www.coverity.com/` ([Online; accessed 2-Jun-2009].)

[3] Coverity, Coverity's Open Source Scan Ladder, Worldwide Web Document (2009). Available at `http://www.scan.coverity.com/rungAll.html` ([Online; accessed 5-May-2009].)

[4] P. Emanuelsson and U. Nilsson, A Comparative Study of Industrial Static Analysis Tools, Technical report, Linköping University (2008).

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem, Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI)* (2000).

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code., *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04)* (2004), 289–302.

[7] Z. Li and Y. Zhou, PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code, *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)* (2005).

[8] Misc, Coccinelle: Semantic Patches for Collateral Evolutions, Worldwide Web Document (2009). Available at `http://www.emn.fr/x-info/coccinelle/` ([Online; accessed 19-May-2009].)

[9] Misc, Coccinelle Wiki, Worldwide Web Document (2009). Available at `http://cocci.ekstranet.diku.dk/wiki/doku.php` ([Online; accessed 08-Jun-2009].)

[10] Misc, Common Weakness Enumeration List, Worldwide Web Document (2009). Available at `http://cwe.mitre.org/data/index.html` ([Online; accessed 4-April-2009].)

[11] Misc, CWE-658: Weaknesses in Software Written in C, Worldwide Web Document (2009). Available at `http://cwe.mitre.org/data/definitions/658.html` ([Online; accessed 4-April-2009].)

[12] Misc, OpenSSL vulnerabilities, Worldwide Web Document (2009). Available at `http://openssl.org/news/vulnerabilities.html` ([Online; accessed 4-April-2009].)

[13] R. C. Seacord, *The CERT C Secure Coding Standard*, Addison-Wesley (2009).

[14] H. Stuart, Hunting bugs with Coccinelle, Masters Thesis, Technical report, DIKU, University of Copenhagen (2008).

B Appendix

The OpenSSL specific patches are in Appendix B.3 on page 44.

B.1 OpenSSL CVE Reports

CVE ID	CWE ID
CVE-2009-1386	CWE-476: NULL Pointer Dereference
CVE-2009-0789	CWE-20: Improper Input Validation
CVE-2009-0591	CWE-20: Improper Input Validation
CVE-2009-0590	CWE-20: Improper Input Validation
CVE-2008-5077	CWE-347: Improper Verification of Cryptographic Signature
CVE-2008-1672	CWE-415: Double Free CWE-476: NULL Pointer Dereference
CVE-2008-0891	CWE-415: Double Free CWE-476: NULL Pointer Dereference
CVE-2006-4343	CWE-20: Improper Input Validation CWE-131: Incorrect Calculation of Buffer Size
CVE-2006-4339	CWE-20: Improper Input Validation
CVE-2006-3738	CWE-131: Incorrect Calculation of Buffer Size
CVE-2006-2940	CWE-20: Improper Input Validation
CVE-2006-2937	CWE-20: Improper Input Validation
CVE-2005-2969	CWE-20: Improper Input Validation
CVE-2004-0975	CWE-377: Insecure Temporary File
CVE-2004-0112	CWE-476: NULL Pointer Dereference
CVE-2004-0081	CWE-241: Improper Handling of Unexpected Data Type
CVE-2004-0079	CWE-476: NULL Pointer Dereference
CVE-2003-0851	CWE-20: Improper Input Validation CWE-241: Improper Handling of Unexpected Data Type
CVE-2003-0545	CWE-20: Improper Input Validation
CVE-2003-0544	CWE-20: Improper Input Validation
CVE-2003-0543	CWE-190: Integer Overflow or Wraparound
CVE-2003-0147	CWE-380: Technology-Specific Time and State Issues
CVE-2003-0131	CWE-20: Improper Input Validation
CVE-2003-0078	CWE-20: Improper Input Validation CWE-201: Information Leak Through Sent Data
CVE-2002-0659	CWE-20: Improper Input Validation
CVE-2002-0657	CWE-131: Incorrect Calculation of Buffer Size
CVE-2002-0656	CWE-131: Incorrect Calculation of Buffer Size
CVE-2002-0655	CWE-681: Incorrect Conversion between Numeric Types

Figure 9: List of CVE reported vulnerabilities based on the CWE taxonomy

The CWE classification in Figure 9 are based on reports on vupen.com, and to a certain degree on my own judgement.

CWE ID	Count
CWE-20: Improper Input Validation	14
CWE-476: NULL Pointer Dereference	5
CWE-131: Incorrect Calculation of Buffer Size	4
CWE-415: Double Free	2
CWE-241: Improper Handling of Unexpected Data Type	2
CWE-347: Improper Verification of Cryptographic Signature	1
CWE-377: Insecure Temporary File	1
CWE-380: Technology-Specific Time and State Issues	1
CWE-681: Incorrect Conversion between Numeric Types	1
CWE-190: Integer Overflow or Wraparound	1
CWE-201: Information Leak Through Sent Data	1

Figure 10: List of CWE IDs based on their appearence in OpenSSL CVE reports

B.2 Pre-made Patches

All pre-made semantic patches are also available at the Coccinelle web page at http://www.emn.fr/x-info/coccinelle/rules/

andand.cocci

```
// The right argument of || or && is dereferencing something known to be
    ↪NULL
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/andand.html
// Options:

@ expression@
expression E;
identifier fld;
@@

- !E &&
+ !E ||
  <+...E->fld...+>

@ expression@
expression E;
identifier fld;
@@

- E ||
+ E &&
  <+...E->fld...+>
```

badzero.cocci

```
// A pointer should not be compared to NULL
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/badzero.html
// Options:

@ disable is_zero,isnt_zero @
expression *E;
expression E1,f;
@@

E = f(...)
<...
(
- E == 0
+ !E
|
- E != 0
+ E
|
- 0 == E
+ !E
|
- 0 != E
+ E
)
...>
?E = E1

@ disable is_zero,isnt_zero @
expression *E;
@@

(
  E ==
- 0
+ NULL
|
  E !=
- 0
+ NULL
|
- 0
+ NULL
  == E
|
- 0
+ NULL
  != E
)
```

continue.cocci

```
// Continue at the end of a for loop has no purpose
//
// Confidence: Moderate
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/continue.html
```

```
// Options:

@@
position p;
@@

for (...;...;...) {
   ...
   if (...) {
     ...
-    continue;
   }
}
```

find_unsigned.cocci

```
// A variable that is declared as unsigned should not be tested to be less
    ↪than
// zero.
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/find_unsigned.html
// Options: -all_includes

@u@ type T; unsigned T i; position p; @@

 i@p < 0

@script:python@
p << u.p;
i << u.i;
@@

print "* file: %s signed reference to unsigned %s on line %s" %
    ↪(p[0].file,i,p[0].line)
```

isnull.cocci

```
// Dereference of an expression that has been checked to be NULL
//
// Confidence: Moderate
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/isnull.html
// Options:

@r exists@
expression E, E1;
identifier f;
statement S1,S2,S3;
position p;
@@

if (E == NULL)
{
  ... when != if (E == NULL) S1 else S2
      when != E = E1
  E@p->f
  ... when any
  return ...;
```

```
}
else S3

@script:python@
E << r.E;
p << r.p;
@@

print "* file: %s deref of NULL value %s on line %s" %
    ↪(p[0].file,E,p[0].line)
```

malloc.cocci

```
@r exists@
local idexpression x;
statement S;
expression E;
identifier f,l;
position p1,p2,p3;
expression *ptr != NULL;
@@

(
if ((x@p1 = malloc(...)) == NULL) S
|
x@p1 = malloc(...);
...
if (x == NULL) S
)
<... when != x
     when != if (...) { <+...x...+> }
(
goto@p3 l;
|
x->f = E
)
...>
(
 return \(0\|<+...x...+>\|ptr\);
|
 return@p2 ...;
)

@script:python@
p1 << r.p1;
p2 << r.p2;
p3 << r.p3;
@@

file = p1[0].file
line1 = p1[0].line
colb1 = p1[0].column
cole1 = p1[0].column_end
line2 = p2[0].line
colb2 = p2[0].column
cole2 = p2[0].column_end
line3 = p3[0].line
colb3 = p3[0].column
cole3 = p3[0].column_end
print "* TODO [[view:%s::face=ovl-face1::linb=%s::colb=%s::cole=%s][ALLOC:
    ↪%s::%s]]" % (file,line1,colb1,cole1,file,line1)
```

```
print "[[view:%s::face=ovl-face2::linb=%s::colb=%s::cole=%s][return with no
    ↪free: %s]]" % (file,line2,colb2,cole2,line2)
print "[[view:%s::face=ovl-face3::linb=%s::colb=%s::cole=%s][via goto:
    ↪%s]]" % (file,line3,colb3,cole3,line3)
print "in function %s" % (p1[0].current_element)
cocci.include_match(False)

@script:python@
p1 << r.p1;
p2 << r.p2;
@@

file = p1[0].file
line1 = p1[0].line
colb1 = p1[0].column
cole1 = p1[0].column_end
line2 = p2[0].line
colb2 = p2[0].column
cole2 = p2[0].column_end
print "* TODO [[view:%s::face=ovl-face1::linb=%s::colb=%s::cole=%s][ALLOC:
    ↪%s::%s]]" % (file,line1,colb1,cole1,file,line1)
print "[[view:%s::face=ovl-face2::linb=%s::colb=%s::cole=%s][return with no
    ↪free: %s]]" % (file,line2,colb2,cole2,line2)
print "in function %s" % (p1[0].current_element)
```

mini_null_ref.cocci

```
// find cases where a pointer is dereferenced and then compared to NULL
// this considers a very special case where the dereference is part of a
// declaration
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/mini_null_ref.html
// Options:

@@
type T;
expression E;
identifier i,fld;
@@

- T i = E->fld;
+ T i;
  ... when != E
      when != i
  if (E == NULL) { ... return ...; }
+ i = E->fld;
```

mini_null_ref2.cocci

```
// find cases where a pointer is dereferenced and then compared to NULL
// this considers a very special case where error handling code has been
// constructed incorrectly
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/mini_null_ref2.html
// Options:
```

```
@@
expression E,E1;
identifier f,fld,fld1;
statement S1,S2;
@@

E->fld = f(...);
... when != E = E1
    when != E->fld1 = E1
if (
-   E
+   E->fld
           == NULL) S1 else S2
```

mini_null_ref3.cocci

```
// find cases where a pointer is dereferenced and then compared to NULL
// this considers a very special case that typically finds dereferences in
// debugging code
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/mini_null_ref3.html
// Options:

@disable is_null@
identifier f;
expression E;
identifier fld;
statement S;
@@

+ if (E == NULL) S
  f(...,E->fld,...);
- if (E == NULL) S

@@
identifier f;
expression E;
identifier fld;
statement S;
@@

+ if (!E) S
  f(...,E->fld,...);
- if (!E) S
```

notand.cocci

```
// !x&y combines boolean negation with bitwise and
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/notand.html
// Options:

@@ expression E; constant C; @@
(
  !E & !C
|
```

```
- !E & C
+ !(E & C)
)
```

notnull.cocci

```
// this detects NULL tests that can only be reached when the value is known
// not to be NULL
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/notnull.html
// Options:

@r exists@
local idexpression x;
expression E;
position p1,p2;
@@

if (x@p1 == NULL || ...) { ... when forall
   return ...; }
... when != \(x=E\|x--\|x++\|--x\|++x\|x-=E\|x+=E\|x|=E\|x&=E\)
    when != &x
(
x@p2 == NULL
|
x@p2 != NULL
)

// another path to the test that is not through p1?

@s exists@
local idexpression r.x;
position r.p1,r.p2;
@@

... when != x@p1
(
x@p2 == NULL
|
x@p2 != NULL
)

// another path to the test from p1?

@t exists@
local idexpression x;
position r.p1,r.p2;
@@

if (x@p1 == NULL || ...) { ... x@p2 ... when any
   return ...; }

// another path to the test containing an assignment?

@u exists@
local idexpression x;
expression E;
position r.p1,r.p2;
@@
```

```
if (x@p1 == NULL || ...) { ... when forall
   return ...; }
 ...
 \(x=E\|x--\|x++\|--x\|++x\|x-=E\|x+=E\|x|=E\|x&=E\|&x\)
 ...  when != x@p1
      when any
(
x@p2 == NULL
|
x@p2 != NULL
)

@fix depends on !s && !t && !u@
position r.p2;
expression x,E;
statement S1,S2;
@@

(
- if ((x@p2 != NULL) || ...)
  S1
|
- if ((x@p2 != NULL) || ...)
  S1
- else S2
|
- (x@p2 != NULL) && E
+ E
|
- (x@p2 == NULL) || E
+ E
|
- if ((x@p2 == NULL) && ...) S1
|
- if ((x@p2 == NULL) && ...) S1 else
  S2
|
- BUG_ON(x@p2 == NULL);
)

@script:python depends on !s && !t && !u && !fix@
p1 << r.p1;
p2 << r.p2;
@@

cocci.print_main(p1)
cocci.print_secs("retest",p2)
```

null_ref.cocci

```
// find cases where a pointer is dereferenced and then compared to NULL
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/null_ref.html
// Options:

@match exists@
expression x, E,E1;
identifier fld;
```

```
position p1,p2;
@@

(
x = E;
... when != \(x = E1\|&x\)
x@p2 == NULL
... when any
|
x = E
... when != \(x = E1\|&x\)
x@p2 == NULL
... when any
|
x != NULL && (<+...x->fld...+>)
|
x == NULL || (<+...x->fld...+>)
|
x != NULL ? (<+...x->fld...+>) : E
|
&x->fld
|
x@p1->fld
... when != \(x = E\|&x\)
x@p2 == NULL
... when any
)

@other_match exists@
expression match.x, E1, E2;
position match.p1,match.p2;
@@

(
x = E1
|
&x
)
... when != \(x = E2\|&x\)
    when != x@p1
x@p2

@other_match1 exists@
expression match.x, E2;
position match.p1,match.p2;
@@

... when != \(x = E2\|&x\)
    when != x@p1
x@p2

@ script:python depends on !other_match && !other_match1@
p1 << match.p1;
p2 << match.p2;
@@

cocci.print_main(p1)
cocci.print_sec("NULL test",p2)
```

open.cocci

```
@r exists@
local idexpression x;
statement S;
expression E;
identifier f,l;
position p1,p2,p3;
expression *ptr != NULL;
@@

(
if ((x@p1 = \(open\|fopen\)(...)) == NULL) S
|
x@p1 = \(open\|fopen\)(...);
...
if (x == NULL) S
)
<... when != x
     when != if (...) { <+...x...+> }
(
goto@p3 l;
|
x->f = E
)
...>
(
 return \(0\|<+...x...+>\|ptr\);
|
 return@p2 ...;
)

@script:python@
p1 << r.p1;
p2 << r.p2;
p3 << r.p3;
@@

file = p1[0].file
line1 = p1[0].line
colb1 = p1[0].column
cole1 = p1[0].column_end
line2 = p2[0].line
colb2 = p2[0].column
cole2 = p2[0].column_end
line3 = p3[0].line
colb3 = p3[0].column
cole3 = p3[0].column_end
print "* TODO [[view:%s::face=ovl-face1::linb=%s::colb=%s::cole=%s][ALLOC:
    ↪%s::%s]]" % (file,line1,colb1,cole1,file,line1)
print "[[view:%s::face=ovl-face2::linb=%s::colb=%s::cole=%s][return with no
    ↪free: %s]]" % (file,line2,colb2,cole2,line2)
print "[[view:%s::face=ovl-face3::linb=%s::colb=%s::cole=%s][via goto:
    ↪%s]]" % (file,line3,colb3,cole3,line3)
print "in function %s" % (p1[0].current_element)
cocci.include_match(False)

@script:python@
p1 << r.p1;
p2 << r.p2;
@@

file = p1[0].file
line1 = p1[0].line
```

```
colb1 = p1[0].column
cole1 = p1[0].column_end
line2 = p2[0].line
colb2 = p2[0].column
cole2 = p2[0].column_end
print "* TODO [[view:%s::face=ovl-face1::linb=%s::colb=%s::cole=%s][ALLOC:
    ↪%s::%s]]" % (file,line1,colb1,cole1,file,line1)
print "[[view:%s::face=ovl-face2::linb=%s::colb=%s::cole=%s][return with no
    ↪free: %s]]" % (file,line2,colb2,cole2,line2)
print "in function %s" % (p1[0].current_element)
```

sizeof.cocci

```
// Applying sizeof to the result of sizeof makes no sense
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/sizeof.html
// Options:

@@
expression E;
@@

- sizeof (
  sizeof (E)
- )

@@
type T;
@@

- sizeof (
  sizeof (T)
- )
```

unused.cocci

```
// A variable is only initialized to a constant and is never used otherwise
//
// Confidence: High
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// URL: http://www.emn.fr/x-info/coccinelle/rules/unused.html
// Options:

@e@
identifier i;
position p;
type T;
@@

extern T i@p;

@@
type T;
identifier i;
constant C;
position p != e.p;
@@
```

```
- T i@p;
  <+... when != i
- i = C;
  ...+>
```

B.3 Custom Patches

The following semantic patches are generally inspired by or rewritten from existing patches, to fit the bug types of OpenSSL. Original copyright notes are kept in each case.

malloc_style.cocci

```
//
// Wrong method call for malloc and free
//

@@
expression E;
@@

- free(E);
+ OPENSSL_free(E);

@@
expression E;
@@

- malloc(E);
+ OPENSSL_malloc(E);
```

openssl_malloc.cocci

```
// This tests for cases where a pointer is not freed upon returning of the
    ↪function, and especially
// before jumping to an err label (since this is so widely used in
    ↪OpenSSL). A call to OPENSSL_free() is
// inserted before returning, to prevent leaks.
//
// Modified for OpenSSL by Sune Rievers

@r exists@
local idexpression x;
statement S;
expression E;
identifier f,l;
position p1,p2,p3;
expression *ptr != NULL;
@@

(
if ((x@p1 = OPENSSL_malloc(...)) == NULL) S
|
x@p1 = OPENSSL_malloc(...);
...
if (x == NULL) S
)
<... when != x
     when != if (...) { <+...x...+> }
(
goto@p3 l;
...
```

```
label l;
+ OPENSSL_free(x);
|
x->f = E
)
...>
(
return \(0\|<+...x...+>\|ptr\);
|
+ OPENSSL_free(x);
return@p2 ...;
)
```

openssl_malloc_free.cocci

```
// An ALLOC is not matched by an FREE before an error return.
//
// Confidence: Moderate
// Copyright: (C) Gilles Muller, Julia Lawall, EMN, DIKU.  GPLv2.
// Modified for OpenSSL by Sune Rievers
// URL: http://www.emn.fr/x-info/coccinelle/rules/alloc_free.html
// Options:

@r exists@
local idexpression n;
statement S1,S2;
expression E;
expression *ptr != NULL;
type T;
position p1,p2;
@@

(
if ((n = OPENSSL_malloc@p1(...)) == NULL) S1
|
n = OPENSSL_malloc@p1(...)
)
... when != OPENSSL_free((T)n)
    when != if (...) { <+... OPENSSL_free((T)n) ...+> } else S2
    when != true n == NULL  || ...
    when != n = (T)E
    when != E = (T)n
(
  return \(0\|<+...n...+>\|ptr\);
|
+ OPENSSL_free(n);
return@p2 ...;
)
```

use_after_free.cocci

```
// Use after free
//
// Rearranges calls to free() and OPENSSL_free() so that variables are not
    ↪freed until
// after their last usage.
//
// http://cwe.mitre.org/data/definitions/416.html
//
```

```
@@
expression E;
expression E2 != NULL;
expression f;
@@

- free(E);
... when != free(E)
(
free(E);
|
E2 = E;
+ free(E);
+ E = NULL;
|
f(<+...E...+>);
+ free(E);
+ E = NULL;
)

@@
expression E;
expression E2 != NULL;
expression f;
@@

- OPENSSL_free(E);
... when != OPENSSL_free(E)
(
OPENSSL_free(E);
|
E2 = E;
+ OPENSSL_free(E);
+ E = NULL;
|
f(<+...E...+>);
+ OPENSSL_free(E);
+ E = NULL;
)
```