

# Introduction to Static Analyzer

---

Moonzoo Kim

SWTV group

School of Computing, KAIST

# Content

- Coverity Static Analysis
- Use cases of Coverity
- Examples
  - C program 1
  - C program 2
  - Java program

The screenshot displays the Coverity Scan website interface. The header includes the Synopsys logo and navigation links: My Dashboard, FAQ, OSS Success Stories, Projects Using Scan, About, and Community. A user profile for moonzoo.kim@gmail.com is visible in the top right. The main content area features the 'COVERITY SCAN STATIC ANALYSIS' title and a description: 'Find and fix defects in your Java, C/C++, C#, JavaScript, Ruby, or Python open source project for free'. Below this, three bullet points highlight the tool's capabilities: testing every line of code, explaining the root cause of defects, and integration with GitHub and Travis CI. A 'Visit My Dashboard' button is located at the bottom left. On the right, a featured banner for PostgreSQL shows a database icon with a magnifying glass over a buffer overflow vulnerability in the path\_in() function. The banner text states: 'Coverity Scan identifies buffer overflow and overrun vulnerabilities in PostgreSQL. Read more >>'. The banner also includes a series of five dots at the bottom, indicating it is part of a carousel.

More than 8800 open source projects and 48000 developers use Coverity Scan

# Coverity Static Analysis

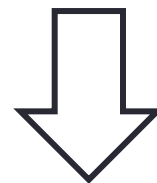
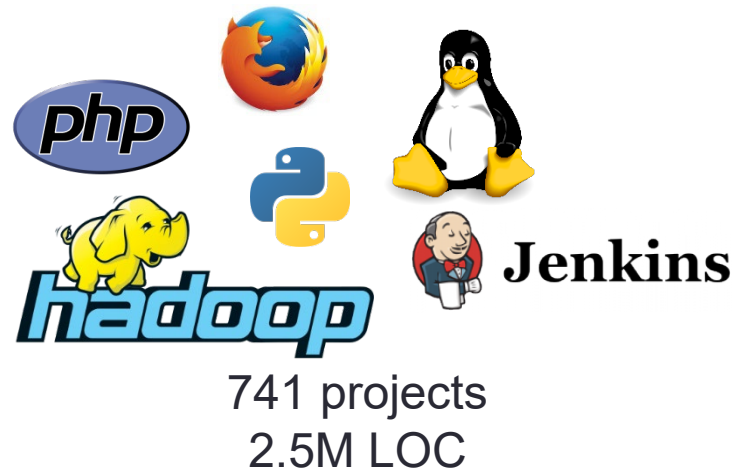
- Coverity Static Analysis is a static code analysis tool for C, C++, C#, Java, and JavaScript
- Coverity Static Analysis is derived from the Stanford Checker, a research tool for finding bugs through static analysis [from Wikipedia]
- Coverity Static Analysis detects dozens of defect patterns in the following categories
  - Memory corruptions
  - Concurrency
  - Security
  - Performance inefficiencies
  - Unexpected behavior

# Power of Coverity

- Coverity can find critical issues such as:
  - API usage errors
  - Buffer overflows
  - Concurrent data access violations
  - Cross-site scripting (XSS)
  - Cross-site request forgery (CSRF)
  - Deadlocks
  - Error handling issues
  - Integer overflows
  - Integer handling issues
  - Memory corruptions
  - Memory illegal accesses
  - Path manipulation
  - Performance inefficiencies
  - Program hangs
  - Security misconfigurations
  - SQL Injection
  - Uninitialized members
  - Control flow issues
  - Hard-coded credentials

# Coverity and Open Source Projects

- Coverity is providing a free service for open source projects



**Coverity Scan**

**44,641 defects are fixed**

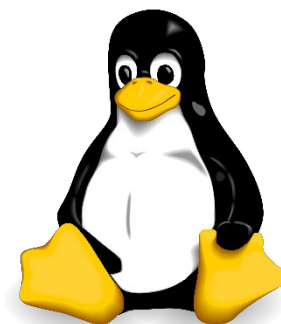
(Only 10.2% of identified defects are false positives in 2013)

# Coverity and Linux

- 18,103 defects are identified in Linux for 8 years (- 2013)
  - 11,695 defects are fixed

Linux defects fixed in 2013

Category	Fixed
Memory illegal access, corruption	1,135
Integer handling issues	816
Null pointer dereferences	291
Uninitialized variables	207
Resource leaks	128
Concurrent data access violations	3
Others	766
Total	3,346



# How To Analyze a program with Coverity

1. Configure coverity
  - `cov-configure --config [configure file] --[gcc | msvc | java]`
2. Build with coverity
  - `cov-build --dir [output directory] --config [configure file] [compile command]`
3. Analyze
  - `cov-analyze --dir [output directory] --all --aggressiveness-level high`
4. Commit analyzed results to server
  - `cov-commit-defects --dir [output directory] --host [server host] --stream [stream name] --user [id] --password [password]`

```
$ cov-configure --config gcc.config --gcc
$ cov-build --dir output --config gcc.config gcc example1.c
$ cov-analyze --dir output --all aggressiveness-level high
$ cov-commit-defects --dir output --host localhost --stream cs453stream
--user cs453 --password 1234
```

# Manage Analyzed Results in Web Interface

**Bug list**

CID	Type	Impact	Status	First Detected	Owner	Classification	Severity	Action	Component	Category
50376	Uninitialized scalar var	High	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Uninitialized variables
50375	Non-constant format str	Low	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Security best practices
50374	Dereference null return	Medium	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Null pointer dereference
50373	Infinite loop with no exit	Medium	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Program hangs
50372	Format string vulnerability	Medium	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Insecure data handling
50369	Resource leak	High	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Resource leaks
50368	Out-of-bounds write	High	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Memory - corruptions
50367	Negative array index write	High	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Memory - corruptions
50366	Ignoring number of bytes read	Medium	New	10/07/15	Unassigned	Unclassified	Unspecified	Undecided	Other	Error handling issues

**Bug description**

**50376 Uninitialized scalar variable**

The variable will contain an arbitrary value left from earlier computations.  
Use of an uninitialized variable (CWE-457)

▼ Triage

Classification:

Severity:

Action:

Ext. Reference:

Owner:

Enter comments (See the Triage History section below for previous comments)

► Projects & Streams

► Detection History

► Triage History

▼ Occurrences

1: cs453stream

Events contributing to issue:

1 var_decl	example1.c:6
2 alloc_fn	example1.c:12
3 assign	example1.c:12
4 uninit_use_in_call	example1.c:15

**Bug detail**

```
1#include <malloc.h>
2#include <stdio.h>
3#include <string.h>
4
5void f() {
6    var_decl: Declaring variable mem.
7    char* mem = NULL;
8    int length;
9    char buf[128];
10   read(0, &length, sizeof(int));
11   int r = read(0, &buf, length > 100 ? 100 : length);
12   2. alloc_fn: Calling allocator malloc.
13   3. assign: Assigning: mem = malloc(r + 1), which is allocated but not initialized.
14   mem = malloc(r + 1);
15   CID 50367: Negative array index write (NEGATIVE_RETURNS) [select issue]
16   CID 50368: Out-of-bounds write (OVERRUN) [select issue]
17   buf[r] = 0;
18   CID 50374: Dereference null return value (NULL_RETURNS) [select issue]
19   strcpy(mem, buf);
20   CID 50375: Non-constant format string (PW_NON_CONST_PRINTF_FORMAT_STRING) [select issue]
21   CID 50372: Format string vulnerability (TAINTED_STRING) [select issue]
22   CID 50376 (#1 of 1): Uninitialized scalar variable (UNINIT)
23   4. uninit_use_in_call: Using uninitialized value *mem when calling printf
24   printf(mem);
25   fflush(stdout);
26   CID 50369: Resource leak (RESOURCE_LEAK) [select issue]
27 }
28
29int main() {
30   CID 50373: Infinite loop with no exit (INFINITE_LOOP) [select issue]
31   while (1)
32       f();
33 }
```



# Example1 - Target C source code

- Bug in this code
  - Copy & paste error

```
1. //example2.c
2. #include <stdio.h>

3. int main(int argc, char** argv) {
4.     int num1=0, num2=0;

5.     if (argc >= 2) {
6.         int n1 = atoi(argv[1]);
7.         int n2 = atoi(argv[1]);

8.         if (n1 >= 0 && n1 <= 100)
9.             num1 = n1;
10.        else
11.            num1 = 5;

12.        if (n2 >= 0 && n2 <= 100)
13.            num2 = n1;
14.        else
15.            num2 = 5;
16.    }
17.    printf("%d %d", num1, num2);
18.}
```

# Example1 - Target C source code

- Copy-paste mistakes also can be detected
  - n1 (line 17) may be relevant to be n2

```
3 int main(int argc, char** argv) {  
4  
5     int num1=0, num2=0;  
6  
7     if (argc >= 2) {  
8         int n1 = atoi(argv[1]);  
9         int n2 = atoi(argv[1]);  
10  
11         if (n1 >= 0 && n1 <= 100)  
12             num1 = n1;  
13         else  
14             num1 = 5;  
15  
16         if (n2 >= 0 && n2 <= 100)  
17             num2 = n1;  
18         else  
19             num2 = 5;  
20  
21     }  
22 }
```

**original:** num1 = n1 looks like the original copy.

❖ CID 50398 (#1 of 1): Copy-paste error (COPY\_PASTE\_ERROR)  
**copy\_paste\_error:** n1 in num2 = n1 looks like a copy-paste error.

💡 Should it say n2 instead?

# Example2 - Target C source code

- Bugs in this code

1. Infinite loop
2. Null pointer dereference
3. Format String Bug
4. Resource Leak
5. Negative Array Index

```
1. //example1.c
2. #include <malloc.h>
3. #include <stdio.h>
4. #include <string.h>

5. void f() {
6.     char* mem = NULL;
7.     int length;
8.     char buf[100];
9.     // file descriptor 0 is connected to keyboard
10.    read(0, &length, sizeof(int));
11.    int r = read(0, &buf, length > 100 ? 100 : length);
12.    mem = malloc(r + 1);
13.    buf[r] = 0;
14.    strcpy(mem, buf);
15.    printf(mem);
16.    fflush(stdout);
17. }

18. int main() {
19.     while (1)
20.         f();
21. }
```

# Example2 – Null pointer dereference

- malloc() may return null if it fails to allocate a memory (line 12)
  - e.g.) malloc(0)
  - e.g.) malloc(BIG\_NUMBER)

Execution sequence that triggers the bug

```
5 void f() {  
6     char* mem = NULL;  
7     int length;  
8     char buf[128];  
9  
10    read(0, &length, sizeof(int));  
11    int r = read(0, &buf, length > 100 ? 100 : length);  
12    mem = malloc(r + 1);  
13    buf[r] = 0;  
14    strcpy(mem, buf);  
15    printf(mem);  
16    fflush(stdout);  
17 }
```

1. returned\_null: malloc returns null.

2. var\_assigned: Assigning: mem = null return value from malloc.

CID 50374 (#1 of 1): Dereference null return value (NULL\_RETURNS)

3. dereference: Dereferencing a pointer that might be null mem when calling strcpy.

Attempt to write a data to mem (NULL)

# Example2 – Format String Bug

- User input is directly used for the first argument of `printf()` (line 15)
  - User can inputs arbitrary format strings such as `printf("%s")` and `printf("%n")` without second argument
    - The program considers a garbage memory value is a second argument
    - This bug causes information leakage or remote code execution vulnerability

```
5 void f() {  
6     char* mem = NULL;  
7     int length;  
8     char buf[128];  
9  
10    read(0, &length, sizeof(int));  
11    1. tainted_string_argument: read taints variable buf.  
12    int r = read(0, &buf, length > 100 ? 100 : length);  
13    mem = malloc(r + 1);  
14    buf[r] = 0;  
15    2. tainted_data_transitive: Call to function strcpy with tainted argument buf transitively taints mem.  
16    strcpy(mem, buf)  
17    3. tainted_string: Passing tainted string mem to a parameter that cannot accept a tainted format string.  
18    printf(mem);  
19    fflush(stdout);  
20 }
```

◆ CID 50372 (#1 of 1): Format string vulnerability (TAINTED\_STRING)

# Example2 – Resource Leak

- mem is not freed although the mem goes out of scope (line 17)

```
5 void f() {  
6     char* mem = NULL;  
7     int length;  
8     char buf[128];  
9  
10    read(0, &length, sizeof(int));  
11    int r = read(0, &buf, length > 100 ? 100 : length);  
12    mem = malloc(r + 1);  
13    buf[r] = 0;  
14    strcpy(mem, buf);  
15    printf(mem);  
16    fflush(stdout);  
17 }
```

1. alloc\_fn: Storage is returned from allocation function malloc.

2. var\_assign: Assigning: mem = storage returned from malloc(r + 1).

3. noescape: Resource mem is not freed or pointed-to in strcpy.

4. noescape: Resource mem is not freed or pointed-to in printf.

5. leaked\_storage: Variable mem going out of scope leaks the storage it points to.

CID 50369 (#1 of 1): Resource leak (RESOURCE\_LEAK)

Not freed

Out of scope of mem



# Example2 – Negative Array Index

- `read()` (line 11) can return negative number if it fails to read
  - The return value is used for array indexing (out of index)

```
5 void f() {
6     char* mem = NULL;
7     int length;
8     char buf[128];
9
10    read(0, &length, sizeof(int));
11    1. negative_return_fn: Function read(0, &buf, ((length > 100) ? 100 : length)) returns a negative
12    number.
13    2. var assign: Assigning: signed variable r = read.
14    int r = read(0, &buf, length > 100 ? 100 : length);
15    mem = malloc(r + 1);
16    3. negative_returns: Using variable r as an index to array buf.
17    buf[r] = 0;
18    strcpy(mem, buf);
19    printf(mem);
20    fflush(stdout);
21 }
```

◆ CID 50367 (#1 of 1): Negative array index write (NEGATIVE\_RETURNS)

# A Missing Bug Case in Example 2

- If a user inputs -1 for length variable (line 9)
  - `(length > 100)` is false (line 10)
  - `read()` receives -1 as a third argument (line 10)
  - The type of the third argument of `read()` is unsigned integer type
    - -1 is converted to `0xffffffff`
  - read an input to `buf` more than 100 bytes (line 10)
    - Stack overflow

```
1. //example1.c
2. #include <malloc.h>
3. #include <stdio.h>
4. #include <string.h>

5. void f() {
6.     char* mem = NULL;
7.     int length;
8.     char buf[100];

9.     read(0, &length, sizeof(int));
10.    int r = read(0, &buf, length > 100 ? 100 : length);
11.    mem = malloc(r + 1);
12.    buf[r] = 0;
13.    strcpy(mem, buf);
14.    printf(mem);
15.    fflush(stdout);
16.}

17.int main() {
18.    while (1)
19.        f();
20.}
```



# Example3 – Target Java Source Code

- There exists a bug in this Java source code
  - Race Condition
- 3 methods
  - Synchronized add and remove methods (line 6, 9)
  - A getter method (line 12)

```
1. // Example3.java
2. import java.util.*;
3. public class Example3 {
4.     private final Object guardingLock = new Object();
5.     private List<Object> data = new ArrayList<Object>();
6.     public void addData(Object o) {
7.         synchronized(guardingLock) { data.add(o); }
8.     }
9.     public void removeData(Object o) {
10.        synchronized(guardingLock) { data.remove(o); }
11.    }
12.    public Object guardedByViolation(int i) {
13.        return data.get(i);
14.    }
15. }
```

# Example3 – Race Condition

- Context switching can happens while executing get() method (line 15)

```
1 import java.util.*;
2
3 public class Example3 {
4
5     private final Object guardingLock = new Object();
6     private List<Object> data = new ArrayList<Object>();
7
8     public void addData(Object o) {
9         A1. example_lock: Example 1: Locking Example3.guardingLock.
10        A2. example_access: Example 1 (cont.): Example3.data is accessed with lock Example3.guardingLock held.
11        synchronized(guardingLock) { data.add(o); }
12    }
13    public void removeData(Object o) {
14        B1. example_lock: Example 2: Locking Example3.guardingLock.
15        B2. example_access: Example 2 (cont.): Example3.data is accessed with lock Example3.guardingLock held.
16        synchronized(guardingLock) { data.remove(o); }
17    }
18    public Object guardedByViolation(int i) {
19        CID 50402 (#1 of 1): Unguarded read (GUARDED_BY_VIOLATION)
20        1. missing_lock: Accessing data without holding lock Example3.guardingLock. Elsewhere, "Example3.data" is accessed with
21        Example3.guardingLock held 2 out of 3 times.
22
23        return data.get(i);
24    }
25 }
```