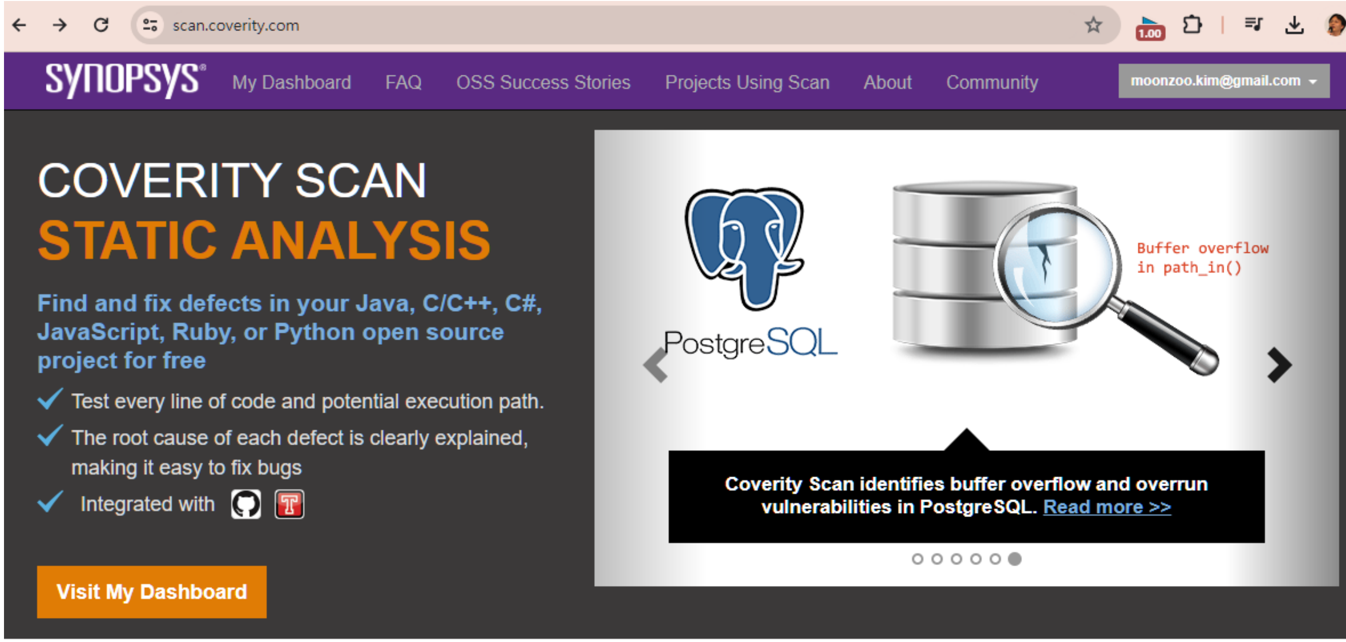
# Introduction to Static Analyzer

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




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 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

Jenkins

741 projects
2.5M LOC

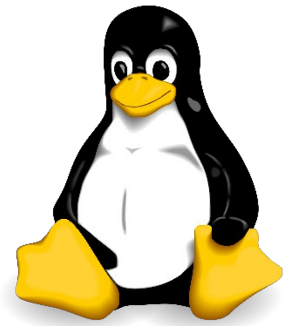
Coverity Scan

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 |

http://softwareintegrity.coverity.com/rs/coverity/images/2013-Coverity-Scan-Report.pdf
http://events.linuxfoundation.org/sites/events/files/slides/2013_10_16_sent.pdf

# 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]

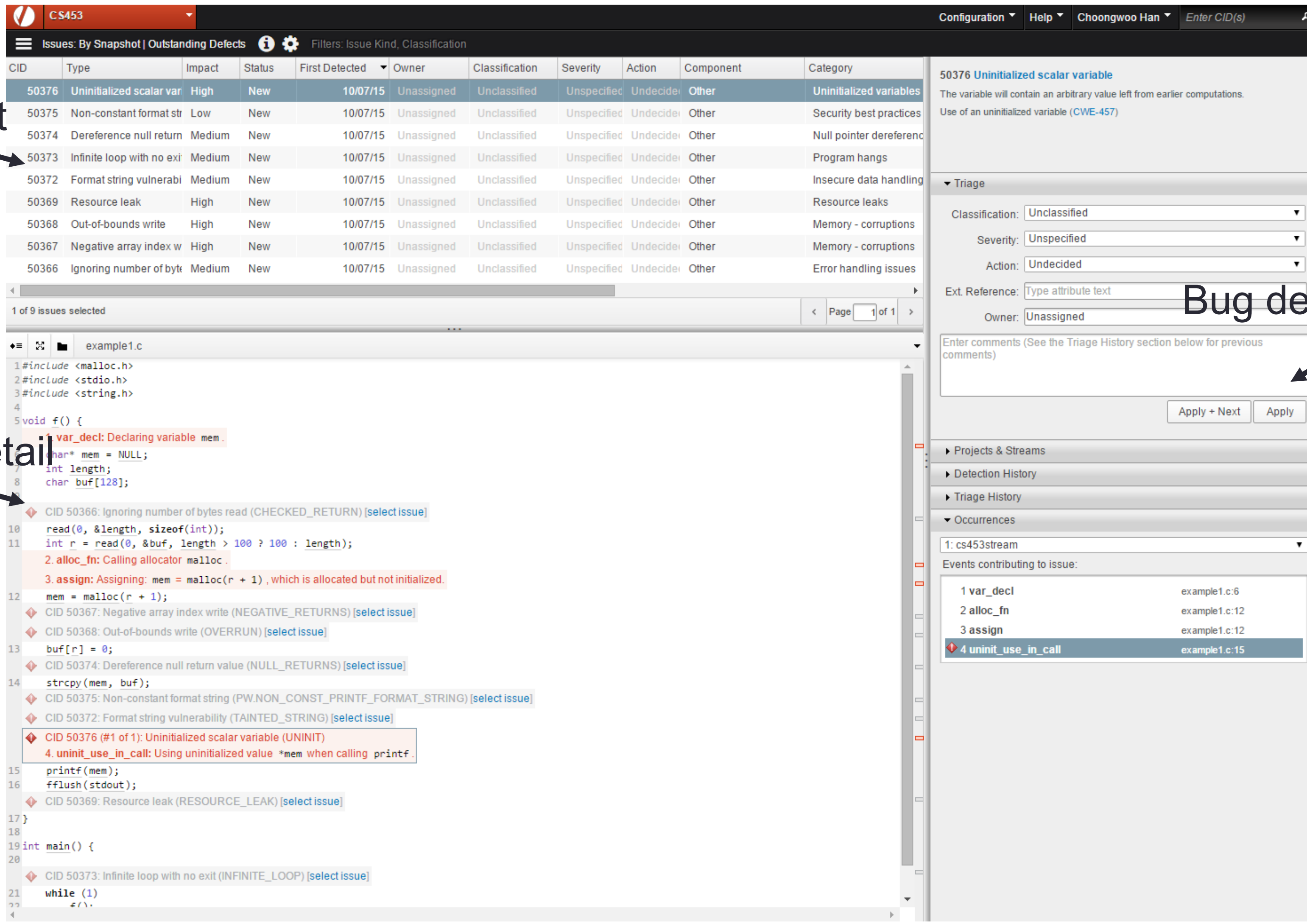
```
$ cov-configure --config gcc.config --gcc
$ cov-build --dir output --config gcc.config gcc example1.c
$ cov-analyze --dir output --all aggressiveness-level high
$ cov-commit-defects --dir output --host localhost--stream cs453stream
--user cs453 --password 1234
```

# Manage Analyzed Results in Web Interface

Bug list

Bug detail

Bug description



# Example1 - Target C source code

- Bug in this code
  - Copy & paste error

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

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

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

            if (n1 >= 0 && n1 <= 100)
                num1 = n1;
            else
                num1 = 5;

            if (n2 >= 0 && n2 <= 100)
                num2 = n1;
            else
                num2 = 5;
        }
        printf("%d %d", num1, num2);
}
```

# 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)
   original: num1 = n1 looks like the original copy.
12                 num1 = n1;
13             else
14                 num1 = 5;
15
16             if (n2 >= 0 && n2 <= 100)
   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?
17                 num2 = n1;
18             else
19                 num2 = 5;
20
21         }
22
```

# 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

```c
//example1.c
#include <malloc.h>
#include <stdio.h>
#include <string.h>

void f() {
    char* mem = NULL;
    int length;
    char buf[100];
    // file descriptor 0 is connected to keyboard
    read(0, &length, sizeof(int));
    int r = read(0, &buf, length > 100 ? 100 : length);
    mem = malloc(r + 1);
    buf[r] = 0;
    strcpy(mem, buf);
    printf(mem);
    fflush(stdout);
}

int main() {
    while (1)
        f();
}
```

# 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)

```
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);
       1. returned_null: malloc returns null.
       2. var_assigned: Assigning: mem = null return value from malloc .
12     mem = malloc(r + 1);
13     buf[r] = 0;
       CID 50374 (#1 of 1): Dereference null return value (NULL_RETURNS)
       3. dereference: Dereferencing a pointer that might be null mem when calling strcpy .
14     strcpy(mem, buf);
15     printf(mem);
16     fflush(stdout);
17 }
```

Execution sequence that triggers the bug

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));
       1. tainted_string_argument: read taints variable buf .
11     int r = read(0, &buf, length > 100 ? 100 : length);
12     mem = malloc(r + 1);
13     buf[r] = 0;
       2. tainted_data_transitive: Call to function strcpy with tainted argument buf transitively taints mem .
14     strcpy(mem, buf);
       CID 50372 (#1 of 1): Format string vulnerability (TAINTED_STRING)
       3. tainted_string: Passing tainted string mem to a parameter that cannot accept a tainted format string.
15     printf(mem);
16     fflush(stdout);
17 }
```

# 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);
       1. alloc_fn: Storage is returned from allocation function malloc .
       2. var_assign: Assigning: mem = storage returned from malloc(r + 1) .
12     mem = malloc(r + 1);
13     buf[r] = 0;
       3. noescape: Resource mem is not freed or pointed-to in strcpy .
14     strcpy(mem, buf);
       4. noescape: Resource mem is not freed or pointed-to in printf .
15     printf(mem);
16     fflush(stdout);
       CID 50369 (#1 of 1): Resource leak (RESOURCE_LEAK)
       5. leaked_storage: Variable mem going out of scope leaks the storage it points to.
17 }
```

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));
       1. negative_return_fn: Function read(0, &buf, ((length > 100) ? 100 : length)) returns a negative number.
       2. var_assign: Assigning: signed variable r = read .
11     int r = read(0, &buf, length > 100 ? 100 : length);
12     mem = malloc(r + 1);
       CID 50367 (#1 of 1): Negative array index write (NEGATIVE_RETURNS)
       3. negative_returns: Using variable r as an index to array buf .
13     buf[r] = 0;
14     strcpy(mem, buf);
15     printf(mem);
16     fflush(stdout);
17 }
```

# 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. }
```

[from coverity example]

# 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) {
     A1. example_lock: Example 1: Locking Example3.guardingLock.
     A2. example_access: Example 1 (cont.): Example3.data is accessed with lock Example3.guardingLock held.
9          synchronized(guardingLock) { data.add(o); }
10     }
11     public void removeData(Object o) {
     B1. example_lock: Example 2: Locking Example3.guardingLock.
     B2. example_access: Example 2 (cont.): Example3.data is accessed with lock Example3.guardingLock held.
12         synchronized(guardingLock) { data.remove(o); }
13     }
14     public Object guardedByViolation(int i) {
     CID 50402 (#1 of 1): Unguarded read (GUARDED_BY_VIOLATION)
     1. missing_lock: Accessing data without holding lock Example3.guardingLock. Elsewhere, "Example3.data" is accessed with
     Example3.guardingLock held 2 out of 3 times.
15         return data.get(i);
16     }
17 
18 }
```