Carnegie Mellon University
18-732: Secure Software Systems
Spring 2014
Homework 3

John Filleau
Norman Wu
Deanna Zhu

1. Coverity Static Analysis

Cyrus IMAP

False Positive 1:
Category: MISSING_BREAK.errors.xml

```
<event>
<tag>fallthrough</tag>
<description>{CovLStrv2{{t{The above case falls through to this one.}}}}</description>
<line>1359</line>
</event>
```

Code:

```
case '\\':
   if (*++yyp != '\\')
      goto do_not_strip_quotes;
   /* Fall through. */
default:
   …
```

While fall-thrus are generally considered to be an error by Coverity, this instance is obviously intentional as indicated by the programmer's "Fall through." comment.

True Positive 1:
Category: NEGATIVE_RETURN.errors.xml

```
<event>
<main>true</main>
<tag>negative_returns</tag>
<description>{CovLStrv2{{t{{0} is passed to a parameter that cannot be
negative.}{&amp;quot;mailbox-&amp;gt;cache_fd&amp;quot;}}}}</description>
<line>533</line>
```

Code:

```
/* make sure we have a file */
if (mailbox->cache_fd == -1) {
   fname = mailbox_meta_fname(mailbox, META_CACHE);
   mailbox->cache_fd = open(fname, 0_REDWR|0_TRUNC|0_CREAT, 0666);
}

/* update the generation number */
*((bit32 *)(buf)) = htonl(mailbox->i.generation_no);
retry_write(mailbox->cache_fd, buf, 4);
```

This bug could be realized if the open() method fails for some reason, and the -1 value could be passed to the retry_write command, which requires an unsigned_integer. -1 would be interpreted as the maximum positive integer value.

True Positive 2:
Category: STRING_NULL.errors.xml

<event>
<tag>string_null</tag>
<description>{CovLStrv2{{t{Passing unterminated string {0} to {1}.}{"response"}{"syslog(int, char const *, ...)"}}}}</description>
<line>467</line>
</event>

Code:

```
syslog(LOG_ERR,
   “ptload(): bad response from ptloader server: %s”, response);
```

Coverity finds that during certain executions, the string “response” can be lacking a null termination. This can be bad news for syslog, where vital data can be written to log that should not have been.

Ratbox IRC

False Positive 2:
Category: UNINIT.errors.xml

```
        <event>
        <tag>var_decl</tag>
        <description>{CovLStrv2{{t{Declaring variable {0} without
        initializer.}{&amp;quot;buffer&amp;quot;}}}}</description>
        <line>1291</line>
        </event>
```

Code:

```
1291:   char buffer[1024];
        FILE *out;
        const char *filename; /* filename to use for kline */

        filename = get_conf_name(type);

        if((out = fopen(filename, "a")) == NULL)
        {
                sendto_realops_flags(UMODE_ALL, L_ALL, "*** Problem opening %s ",
filename);
                return;
        }

        if(oper_reason == NULL)
                oper_reason = "";

        if(type == KLINE_TYPE)
        {
                ircsnprintf(buffer, sizeof(buffer),
                        "\"%s\",\"%s\",\"%s\",\"%s\",\"%s\",\"%s\",%ld\n",
                        user, host, reason, oper_reason, current_date,
                        get_oper_name(source_p), CurrentTime);
        }
        else if(type == DLINE_TYPE)
        {
                ircsnprintf(buffer, sizeof(buffer),
                        "\"%s\",\"%s\",\"%s\",\"%s\",\"%s\",%ld\n", host,
                        reason, oper_reason, current_date, get_oper_name(source_p),
CurrentTime);
        }
        else if(type == RESV_TYPE)
```

```
{
        ircsnprintf(buffer, sizeof(buffer), "\"%s\",\"%s\",\"%s\",%ld\n",
                host, reason, get_oper_name(source_p), CurrentTime);
}
```

This is not a bug because the variable do not need to be initialized when declared. The false positive appears because the static checker have no idea how will variables used later, so it requires the variable to be initialized when it is declared. However, the coder usually knows how the variables will be used later.

False Positive 3:
Category: UNUSED_VALUE.error

```
<event>
<main>true</main>
<tag>returned_pointer</tag>
<description>{CovLStrv2{{t{Pointer {0} returned by {1} is never
used.}{&amp;quot;cf&amp;quot;}{&amp;quot;find_conf_item(tc, name)&amp;quot;}}}}</description>
<line>1692</line>
</event>
```

Code:

```
1692:  if((cf = find_conf_item(tc, name)) != NULL)
               return -1;

       cf = MyMalloc(sizeof(struct ConfEntry));

       DupString(cf->cf_name, name);
       cf->cf_type = type;
       cf->cf_func = func;
       cf->cf_arg = NULL;
```

This is not a bug because whether cf will be used have no effect to rightness of the program. The false positive appears because the static checker believe all variables need to be used after given an value.

False Positive 4:
Category: OVERRUN.error

```
<event>
<tag>alias</tag>
<description>{CovLStrv2{{t{Assigning: {0} =
{1}.}{&amp;quot;user&amp;quot;}{&amp;quot;splat&amp;quot;}}{t{ {0} now points to byte {1} of {2} (which
consists of {3} bytes).}{&amp;quot;user&amp;quot;}{0}{&amp;quot;splat&amp;quot;}{2}}}}</description>
```

<line>551</line>
</event>

Code:

551: nick = user = host = splat;

The false positive appears because the static checker did not understand this uncommon variables initialize way.

True Positive 3:

Category: STRING_OVERFLOW.error

<event>
<main>true</main>
<tag>fixed_size_dest</tag>
<description>{CovLStrv2{{t{You might overrun the {0} byte fixed-size string {1} by copying {2} without checking the length.}{128}{"comment1"}{"source_p->name"}}}}</description>
<line>1479</line>
</event>

Code:

1479: strcat(comment1, source_p->name);

True Positive 4:

Category: REVERSE_INULL.error

<event>
<tag>deref_ptr</tag>
<description>{CovLStrv2{{t{Directly dereferencing pointer {0}.}{"source_p"}}}}</description>
<line>288</line>
</event>

Code:

```
register_local_user(struct Client *client_p, struct Client *source_p, const char *username)
{
        struct ConfItem *aconf;
288:    struct User *user = source_p->user;
```

This is risk because source_p could be NULL. Then dereference NULL pointer will cause error.

OptiPNG

False Positive 5:
Category: SIZEOF-MISMATCH.errors

<event>
<main>true</main>
<tag>suspicious_sizeof</tag>
<description>{CovLStrv2{{t{Passing argument "{0}" to function {1} and then casting the return value to "{2}" is suspicious.}{comp->max_output_ptr * 4U /* sizeof (png_charpp) */}{"png_malloc(png_structp, png_uint_32)"}{png_charpp}}{t{ Did you intend to use "sizeof({0})" instead of "{1}" ?}{char *}{sizeof (png_charpp)}}{t{ In this particular case sizeof({0}) happens to be equal to sizeof({1}), but this is not a portable assumption.}{png_charpp}{char *}}}}</description>
<line>241</line>
</event>

Code:

```
241:    comp->output_ptr = (png_charpp)png_malloc(png_ptr,
(png_uint_32)(comp->max_output_ptr * png_sizeof (png_charp)));
```

This is a false positive because png_charpp is defined to be the size of a char* pointer.

False Positive 6:
Category: UNINIT.errors.xml

<event>
<main>true</main>
<tag>uninit_use_in_call</tag>
<description>{CovLStrv2{{t{Using uninitialized element of array {0}: field {1}.{2} is uninitialized when calling {3}.}{"palette"}{"palette"}{"blue"}{"opng_insert_palette_entry(png_colorp, int *, png_bytep, int *, int, unsigned int, unsigned int, unsigned int, unsigned int, int *)"}}}}</description>
<line>903</line>

Code:

```
if (opng_insert_palette_entry(palette, &num_palette,
            trans, &num_trans, 256,
            red, green, blue, alpha, &index) < 0)
```

This is a false positive because palette is meant to be uninitialized and filled in with values based on the argument values.

True Positive 5:
Category: DEADCODE.errors.xml

<event>
<main>true</main>
<tag>dead_error_line</tag>
<description>{CovLStrv2{{t{Execution cannot reach this expression {0} inside statement {1}.}{"s->outbuf == NULL"}{"if (err != 0 || s->outbuf =..."}}}}</description>
<line>161</line>
</event>

Code:

```
161:    if (err != Z_OK || s->outbuf == Z_NULL) {
        return destroy(s), (gzFile)Z_NULL;
        }
```

This is checking for errors in creating input/output buffers for reading/writing a file, and destroying the stream if an error occured. This is a potential vulnerability because the checker has determined it to be unreachable during execution so the stream is never destroyed. This could cause issues with other resources that want to use the file.

AAlib

True Positive 6:
Category: USE_AFTER_FREE.errors.xml

```
<event>
<main>true</main>
<tag>deref_arg</tag>
<description>{CovLStrv2{{t{Calling {0} dereferences freed pointer
{1}.}{&amp;quot;aa_currentfont(aa_context *)&amp;quot;}{&amp;quot;c&amp;quot;}}}}</description>
<line>21</line>
```

Code:

```
if (f == NULL) {
    aa_close(c);
    printf("Can not open output file\n");
}
fprintf(f, "#include \"aalib.h\"\n"
        "static unsigned char %sdata[] =\n"
        "{", argv[2]);
font = aa_currentfont(c);
```

If the file cannot be opened, then the variable c is closed, freeing the memory. Later, aa_currentfont is called which required a dereferenced pointer, even though the memory has been freed.

Category:
Code:

2. Simple Checker
The way our simple checker work is by finding those insecure functions which including gets(), strcpy(), strcat(), sprintf(), rand(). It explains why these functions are insecure and warns programmers to replace them with more secure functions which are fgets(), strncpy(), strncat(), snprintf(), srand().

Our checker does not check all the insecure functions as we do not have a full list of insecure function in Coverity. But other insecure functions could be easily checked in the same way.

3. Stateful malloc Return Value Checker

The checker matches function calls to malloc and saves the pointer variable in the int store with a value of 1 to indicate malloc has been called for it. The checker also matches if statements that check if a variable is null (ie. “if(x)”). If that variable is already in the int store (meaning it was assigned to a malloc value), it will assign that variable a value of 0 in the int store. The checker also checks every pointer reference. If that pointer hasn’t been checked yet, it will be assigned a value of 2 and an error will be committed with the variable name and line number in the file. It only reports the error once so if the pointer is dereferenced again before being reassigned, it won’t report the error twice.

With this checker, if a pointer were set to null (x = null), there would be a false positive because it wouldn’t really matter if the return value of malloc were null. A possible improvement would be to see how the return value is being used to reduce false positives.

This checker could be improved by expanding the “Ifpat” expression pattern. Currently, only statements that check for the variable itself will be matched but if other conditions were added, the statement wouldn’t be matched (for example “if (x && 1)”). This would improve the number of true positives.

4. Stateful Double Free Checker

The stateful double-free checker looks for three types of trees: a Pointer object being given a value by malloc, a Pointer object being passed to the function “free”, and a Pointer object being set to the value of some other pointer object.

Freeing the Pointer will first check if a mapping between that Pointer and a “freed” enum exists. If so, this is an error. Next, that Pointer will be mapped to a “freed” value, so that any future freeings can be errored. Calling malloc on that Pointer will clear the mapping, allowing us to free once again in the future without invoking the error. Using a Pointer = Pointer assignment will copy the mapping (if any exists) from the previous Pointer to the new Pointer. So if a freed Pointer is assigned to a new Pointer, and that new Pointer is freed, that will be considered an error.

5. Checker for server2

For server 2 line 281, the vulnerability is in char * strncat (char * destination, const char * source, size_t num). The num variable cannot be equal to the size of the destination array. It should be smaller than it because strncat adds a null-terminator at the end. So the checker we build for server2 and other similar vulnerabilities mainly checks the size of the destination array and compares it with third argument num. If the size of the destination array is smaller than or equal to num, then the checker will warn the developer that the code is vulnerable and why it's a risk.

The checker can be improved in these ways:

1. Rather than getting the size of destination, getting the actually **remaining size** of destination buffer and then compare the remaining size with num. This would improve the checker because strncat appends the source string to the destination buffer which may not be empty before appending.
This change could increase the number of true positives and decrease false positives.

2. Could compare the actual size of the source string with num then use the **smaller** value to compare with the **remaining size** of destination buffer. This could improve the the checker as the size of source string may be smaller than num.
This change could decrease the number of false positives.

6. Extra Credit: API Tutorial Documentation
A simple checker can be built by following the three steps below:

1. Define the goal
The first step is to define a sequence of source code actions along a code path that could trigger an error. For example: the argument of malloc() should be a positive number.

pointer * p = (pointer *) malloc (v); //error if v is not a valid size, i.e. -2

2. Specify pattern matches
The second step is to specify a Coverity expression pattern to match the line and effect in the source code. At the simplest level, matches can be text-based, such as a function name. In the malloc problem, the pattern would be the function name malloc and the argument in malloc function should be a positive number.

```
 if(MATCH(CallSite("malloc"))) {...}
```

3. Write code and test
The third step is to write code for the checker and test it. By reading the Hello checker sample in SDK ref you will get an idea of the structure of a checker. You can find the API functions you want from the SDK reference document, use those functions with your patterns, and then write some test cases to verify the checker works. For the problem above, the main code can be like below: (note the code below ignores more complicated cases like malloc(sizeof(struct) * x)) as it needs to use more advanced pattern expression which will make the simple checker much more complex).

```
ANALYZE_TREE()
{
   CallSite malloc("malloc");
   Const_int x;

   if(MATCH(malloc(x)))
   {
      //get the first argument  of malloc and compare the value of first argument with 0
      if(x.llval() <= 0)
      {
         COMMIT_ERROR(x, "malloc", "The argument in malloc should be a positive number"<<
                  " [VERY RISKY]" << CURRENT_TREE);
      }
   }
}
```

The test case could be:

pointer * p = (pointer *) malloc (-2); //bug

Ref: http://www.verifysoft.de/en_static_analysis_extend.pdf

API Function
Coverity: ANALYZE_END_OF_PATH()
At the end of every path that is being checked by the checker, this function is called.
Example:

```
ANALYZE_END_OF_PATH(){
   const ASTNode *t;
   int v;
   FOREACH_IN_STORE( t, v ) {
      if ( v > 2) {
        cout << t << " has value: " << v << endl;
      } else {
        SET_STATE(t, 2);
      }
   }
}
```