SE465-001 Project

Sameer Ahmad (20458331) Patrick White (20464876)
Andrew Yiu (20466096)

April 6, 2015

Part I - B

Bug 1 (16 line locations):

```
bug: apr_array_make in ap_init_virtual_host, pair: (apr_array_make, apr_array_push),
support: 40, confidence: 86.96%
```

Code for ap_init_virtual_host:

```
AP_CORE_DECLARE(const char *) ap_init_virtual_host(apr_pool_t *p,
                                                   const char *hostname,
                                                   server_rec *main_server,
                                                   server_rec **ps)
{
        server_rec *s = (server_rec *) apr_pcalloc(p, sizeof(server_rec));

        /* TODO: this crap belongs in http_core */
        s->process = main_server->process;
        s->server_admin = NULL;
        s->server_hostname = NULL;
        s->server_scheme = NULL;
        s->error_fname = NULL;
        s->timeout = 0;
        s->keep_alive_timeout = 0;
        s->keep_alive = -1;
        s->keep_alive_max = -1;
        s->error_log = main_server->error_log;
        s->loglevel = main_server->loglevel;
        /* useful default, otherwise we get a port of 0 on redirects */
        s->port = main_server->port;
        s->next = NULL;

        s->is_virtual = 1;
        s->names = apr_array_make(p, 4, sizeof(char **));
        s->wild_names = apr_array_make(p, 4, sizeof(char **));

        s->module_config = create_empty_config(p);
        s->lookup_defaults = ap_create_per_dir_config(p);

        s->limit_req_line = main_server->limit_req_line;
        s->limit_req_fieldsize = main_server->limit_req_fieldsize;
        s->limit_req_fields = main_server->limit_req_fields;

        *ps = s;

        return ap_parse_vhost_addrs(p, hostname, s);
}
```

This bug indicates a false positive. The bug reported by the static analysis tool states that calling apr_array_make without apr_array_push is considered a bug, given that in 86.96% of uses of apr_array_make, the function will use apr_array_push as well. The reason that in this case the function does not call apr_array_push is because the array that is made is constructed with arguments that do not need pushed values. It is constructed on some values passed in and therefore needs no pushing to.

Bug 2 (2 line locations):

```
bug: qsort in ap_core_reorder_directories, pair: (qsort, strcmp), support: 4,
```

confidence: 66.67%

Code for ap_core_reorder_directories

```
void ap_core_reorder_directories(apr_pool_t *p, server_rec *s)
        {
        core_server_config *sconf;
        apr_array_header_t *sec_dir;
        struct reorder_sort_rec *sortbin;
        int nelts;
        ap_conf_vector_t **elts;
        int i;
        apr_pool_t *tmp;

        sconf = ap_get_module_config(s->module_config, &core_module);
        sec_dir = sconf->sec_dir;
        nelts = sec_dir->nelts;
        elts = (ap_conf_vector_t **)sec_dir->elts;

        if (!nelts) {
        /* simple case of already being sorted... */
        /* We're not checking this condition to be fast... we're checking
        * it to avoid trying to palloc zero bytes, which can trigger some
        * memory debuggers to barf
        */
        return;
        }

        /* we have to allocate tmp space to do a stable sort */
        apr_pool_create(&tmp, p);
        sortbin = apr_palloc(tmp, sec_dir->nelts * sizeof(*sortbin));
        for (i = 0; i < nelts; ++i) {
        sortbin[i].orig_index = i;
        sortbin[i].elt = elts[i];
        }

        qsort(sortbin, nelts, sizeof(*sortbin), reorder_sorter);

        /* and now copy back to the original array */
        for (i = 0; i < nelts; ++i) {
        elts[i] = sortbin[i].elt;
        }

        apr_pool_destroy(tmp);
        }
```

This bug line does represents a false positive. The function qsort is used to sort a set of object, while strcmp is used to compare strings - this means that if the objects that are to be sorted do not rely on strings for ordering, then there would be no reason to use the strcmp function, which is the case here - this function is sorting elts which are not or do not rely on string comparrisons.

Part I - C

Usage

To use the inter-procedural analysis feature, add the expansion depth as a fourth command line parameter to the program: `./pipair 3 65` will not expand nodes. `./pipair 3 65 1` will expand nodes one level deep. `./pipair 3 65 0` will expand to an infinite depth.

Solution Description

When using inter-procedural analysis, the program will expand nodes up to the depth provided, so that each node keeps track of functions that are actually called by the node, and also a set of functions that are called by nodes up to the depth provided. This does not affect the support values of the functions. Consider the following example with $depth = 1$:

```
void A(){}
void B(){}
void C(){
B();
}
void D(){
A();
C();
}
```

In this example, $support(\{A, B\})$ is still 0 because A and B are never called together in the same function, even though D becomes expanded so that its expanded call set is $\{A, B, C\}$. If they did affect each other's *support* levels, the appearance of B in C would be considered to be a bug, since the support of A and B would be 1, but B appears without A in C.
The use of this extended call set is that when a lone occurrence of a normally-paired function is a bug, it checks the extended call set for the presence of the other function instead of only the other functions called by the same node. Consider the example:

```
void A(){}
void B(){}
void C1(){
A();
B();
}
void C2(){
A();
B();
}
//... repeat a number of times
//... enough for the program to determine A->B is an invariant
void D(){
A();
C1();
}
```

In this example, A and B appear together enough for the program to determine that the presence of one without the other is a bug. With no inter-procedural analysis, the appearance of A in D is considered to be a bug because D does not directly call B, even though $C1$ always calls B. With inter-procedural analysis to any depth greater than zero, the $C1$ node is expanded within D so the expanded call set of D is $\{A, B, C1\}$, and this is not reported as a bug.

Experiment

The analysis was run on the `httpd` source code with varying levels of depth with $T\_CONDFIDENCE = 65\%$ and $T\_SUPPORT = 3$. At each level, the number of bugs reported was recorded.

Inter-procedural analysis depth	Reported Bugs
No inter-procedural analysis	205
1	165
2	159
3	156
∞	156

After three levels, no more inter-procedural analysis could reduce the number of bugs reported by the tool. While the inter-procedural analysis still leaves 156 reported bugs, many of which are undoubtedly false positives, it does reduce the overall number of bugs reported by almost 20%. Since the difference between these is considering when functions at different call depths, it is unlikely that this creates any false negatives, since it is only eliminating cases where a bug is reported from the absence of a function call, but it is found at a deeper level. It's worth nothing that most of the bugs were eliminated with a single level of expansion. This means that the programmer was consciously calling a function that was known to be replaced by a call to a function that calls the expected function.

Validity of Solution

As described in the previous section, this solution is unlikely to create false negatives. Also, it cannot create additional bug reports (false positives) since it does not modify the support values of the original analysis. An example of this working can be found by running the analysis with no inter-procedural analysis and one level of inter-procedural analysis on the `httpd` source code. As discussed in (b), it is not incorrect for `apr_array_make` to be called without `apr_array_push`, or vice versa. The version with inter-procedural analysis eliminates several instances of this false positive pair.

Part II - A

Coverity ID 10022

False Positive - All other instances that lead to this warning make use of a call to super for an object to pass to a decorator. In this case, it seems the developer only wanted to return the `KeySetView` instance instead of a decorated set, which is a different scenario from the similar examples resulting in a non-issue.

Coverity ID 10023

Intentional - In the function, the map that is being referenced is from a parent class (AbstractMapDecorator). The other classes which make the call to super also does the same thing in the parent class. A suggested fix should be use a call to super like all the other similar calls.

Coverity ID 10024

Intentional - In the function, the map that is being referenced is from a parent class (AbstractMapDecorator). The other classes which make the call to super also does the same thing in the parent class. A suggested fix should be use a call to super like all the other similar calls.

Coverity ID 10025

False Positive - At the beginning of the flagged block, we have `currentIterator` equal null. After a few method calls, we see a point where `currentIterator` is dereferenced (line 186). Once we enter the first method, we have the `iterator` variable in `findNextByIterator` equal to not null. Then in the first if statement, `currentIterator` is set to `iterator`, which is a non-null value. Then when `currentIterator` is dereferenced, it is no longer null and we do not run into a `nullPointerException`.

Coverity ID 10026

Bug - Since FastArrayList is a thread-safe data structure, and the variable `last` is not volatile, last is not expected to be modified from multiple threads. Without holding a lock before the modification, the value of `last` recorded before the increment could become stale. As well, multiple reads and writes to `last` can come from multiple threads, causing undefined behaviour. The line in question is 852 and a suggested fix is to put it inside a `synchronized` block.

Coverity ID 10027

False Positive - If `deletedNode` is null, then the bug would happen earlier in the function block, which is not related to this function. Since we only get to the flagged statement if `deletedNode.getRight(` `!= null`, then inside the `nextGreater` function call, we take the branch for `deletedNode.getRight(in` `!= null` and end up returning a non-null node. Then the value passed into swapPosition is valid and causes no exceptions.

Coverity ID 10028

Bug - Since FastArrayList is a thread-safe data structure, and the variable `last` is not volatile, last is not expected to be modified from multiple threads. Without holding a lock before the modification, the value of `last` recorded before the increment could become stale. As well, multiple reads and writes to `last` can come from multiple threads, causing undefined behaviour. The line in question is 1241 and a suggested fix is to put it inside a `synchronized` block.

Coverity ID 10029

Bug - This is a bug with multi-threading because the first thread to reach the lock will modify the `lastReturned` value to null. When the next thread to reach the lock is able to enter the critical section, `lastReturned` will be null and line 664 will dereference `lastReturned`, causing a `nullPpinterException`. A proposed fix is to include the null check found on lines 656 to 658 inside the critical section.

Coverity ID 10030

False Positive - Because the lock on `lock` is guaranteed before entering the `isEmpty` function. This means that the calls within the isEmpty() function are guaranteed to have the lock. The isEmpty function attempts to `synchronize(lock)` which it is guaranteed to have. The false positive comes from the order in which the locks occur.

Coverity ID 10031

Intentional - The developer understands that they can assume there is a right subtree to rotate upon, by the choice of making rotate right a private function to be called only through the function that balances the tree. If the tree must be balanced, then there must exist a subtree for the right branch.

Coverity ID 10032

Bug - This is because if the threads both entered the while loop , then one of the threads locked the bucket, performed its `hasnext` function, then unlocks the bucket. The other thread returns to inside the while loop, while it is possible that `bucket++` is null. Then it will dereference null, causing an error.

Coverity ID 10033

False Positive - This function is intentionally creating a reverse mapping from the original mapping, it is not a bug.

Coverity ID 10034

Bug - The report for CID 10034 is the same as the bug report for CID 10032, and all the suggestions for fixing the bug are the same.

Coverity ID 10035

Bug - Since FastArrayList is a thread-safe data structure, and the variable `last` is not volatile, last is not expected to be modified from multiple threads. Without holding a lock before the modification, the value of `last` recorded before the increment could become stale. As well, multiple reads and writes to `last` can come from multiple threads, causing undefined behaviour. The line in question is 1221 and a suggested fix is to put it inside a `synchronized` block.

Coverity ID 10036

Bug - This code block is entered under the assumption that `lastReturned` is not null. When two threads check for null at the same time and pass, one thread will lock before the other, set the value of `lastReturned` to null, unlock and let the other thread in. Then the other thread will hit line 767 and dereference null. A suggested fix is to put the null check inside the lock to ensure we do not dereference null.

Coverity ID 10037

Intentional - The function can be modified to perform a null check on the left subtree before dereferencing it. The developer assumes that the `rotateRight` function is called when a left subtree exists. This leaves the user to perform these checks before calling, or to handle `nullPointerExceptions`.

Coverity ID 10038

Bug - Most other examples will just call `synchronize(lock)`. If this instance gets the lock on `lock` after `synchronize(list)`, then there is a large chance to deadlock when others hold the lock on `lock`. The suggested fix is to be applied around line 1135 is to lock on `lock` first.

Coverity ID 10039

False Positive - If `deletedNode` is null, then the bug would happen earlier. Since we only get to the flagged statement if `deletedNode.getRight(index) != null`, then inside the `nextGreater` call, the returned value will not be null.

Coverity ID 10040

False Positive - By inspecting other similar code snippets, the order of the locks occur with `synchronized(map)` followed by `synchronized(lock)`. Then no locking order is violated and no deadlock will occur.

Coverity ID 10041

Intentional - This data structure is not a synchronized (thread-safe) type. Since this class is not intended to be synchronized, `modCount` does not need ot be handled as a volatile type, but the developer knowingly set it as a volatile type. A suggested change is to change the type of `modCount` to not be volatile.

Coverity ID 10042

Intentional - This data structure is not a synchronized (thread-safe) type. Since this class is not intended to be synchronized, `modCount` does not need ot be handled as a volatile type, but the developer knowingly set it as a volatile type. A suggested change is to change the type of `modCount` to not be volatile.

Part II - B

Coverity ID 10363 - Uninitialized Pointer Read

This coverity found defect suggests that a pointer has not been initialized by the time it is used for comparison while parsing input (the call graph). This will arise when the case statement in the main while loop goes to `case CALL` first. With well-formed input, this should never happen. It is expectected to enter the `NODE` case first, initializing the `current_function` pointer.

Coverity ID 10362 - Uncaught Exception

When using the boost library's string formatter, it could throw an exception for too many or too few arguments. This is not a real problem because we can deduce the exact number of arguments required for the formatted string. Then the exceptions will not be thrown and explicit exception handling is not required.