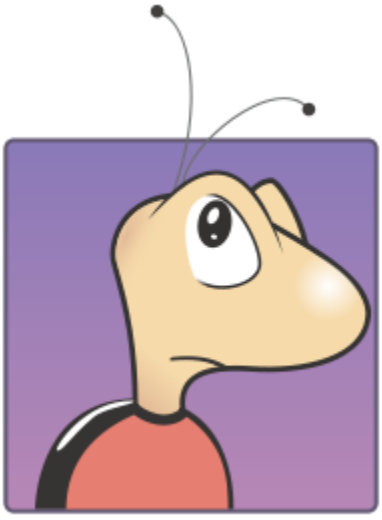
# Vulnerability Disclosure



SQL Injection

In

Bugzilla

*Netanel Rubin*

*26.11.2015*

## Vulnerable Versions

*? – Fully Patched (5.0.1)*

## Summary

Bugzilla is a very popular open source bug tracking software written in Perl. It allows an organization to easily organize the bugs in its products and developers to conveniently document any bug they've found or fixed.

This vulnerability allows an unauthenticated attacker to perform an SQL Injection attack, effectively allowing the complete compromise of the server in some cases. The only requirement for the successful exploitation of the vulnerability is that Perl's "Taint Mode" is set to "Off" in the particular vulnerable script (or in general).

Although the exploitation of the vulnerability is restricted by Perl's "Taint Mode", some installations are deliberately disabling/removing parts of it due to conflicts with some 3rd party modules, and Mozilla even considers disabling it (link1, link2) in their own Bugzilla installation (from our test it looks like it is actually disabled) and possibly setting it as an optional configuration value for all installations.

## Technical Description

Bugzilla, like a lot of other Perl projects, makes heavy use of the “ref” keyword. The “ref” keyword is responsible for returning the variable type, and Bugzilla uses it a lot, especially in its object initialization functions.

Such function is *‘_load_from_db()’*, which can be seen here:

```
sub _load_from_db {
    my $class = shift; # The object instance
    my ($param) = @_; # The input parameter
    my $dbh = Bugzilla->dbh; # The DB handler

    # Get the ID from the parameter
    my $id = $param;
    if (ref $param eq 'HASH') {
        $id = $param->{id};
    }

    ## Regular, scalar parameter check ##
    if (defined $id) {
        # Force the ID to be numeric
        detaint_natural($id);

        $object_data = $dbh->selectrow_hashref(qq{
            SELECT $columns FROM $table
             WHERE $id_field = ?}, undef, $id);
    ## Special, hash parameter check ##
    } else {
       # Get the special values from the hash
       $condition = $param->{'condition'};
       push(@values, @{$param->{'values'}});

       # Query using them ($condition is used as it)
       $object_data = $dbh->selectrow_hashref(
            "SELECT $columns FROM $table WHERE $condition", undef, @values);

    }
    return $object_data;
}
```

It is easy to see that the function acts very differently based on *‘$param’* data type.
In case *‘$param’* is a scalar, it is converted into an integer by removing all non-numeric characters from it. On the other hand, if it’s a hash, specific keys are extracted from it and used inside the SQL query.

This means that if we control *‘$param’* value, and, of course, its data type, we will be able to exploit an SQL Injection attack on the system.
Even though controlling *‘$param’* value is easy, controlling its data type will prove to be rather

difficult. Thinking about it, it sounds like an impossible thing to do – How can one even control input parameters data type?

Well, the answer lies in how the input is being parsed. Bugzilla uses the *'CGI.pm'* module, which allows to set lists as the input freely (something an attacker can abuse), but restricts the use of any other data types. In fact, neither hashes nor arrays can be created using this module.
Luckily for us, *'CGI.pm'* is **not** the only module parsing the user input. Another module assigned to that task is, of course, the XMLRPC module.

Bugzilla implemented several ways of communicating with the system API remotely, using REST API, JSONRPC, and our XMLRPC.
When a user sends an XMLRPC request, the server parses it using the XMLRPC standard, which allows the use of arrays and dictionaries as input types. Because of that, Bugzilla also allows the use of these non-standard data types as parameters in its RPC functions.

Unfortunately, the input data-type check is missing on several occasions in this RPC. One of those occasions can be seen in the *'get()'* function inside the *'bug'* web service module:

```
sub get {
    # Get the bug IDs from the user input
    my $ids = $params->{ids};

    # Loop through the bug IDs
    foreach my $bug_id (@$ids) {
        # Check the given bug (basically extract it from the DB and
        # check for permissions)
        $bug = Bugzilla::Bug->check($bug_id);

        # Add the bug to the bugs array
        push(@hashes, $self->_bug_to_hash($bug, $params));
    }

    # Return the bugs array
    return { bugs => \@hashes, faults => \@faults };
}
```

As we control *'$params'*, we control *'$ids'*. Because the input came from the RPC module, we are not restricted to scalar values only as the input, we can also use arrays and hashes.

So we can insert a hash into *'Bug->check()'*, and cause an SQLI if we insert a hash into *'_load_from_db()'*, but we still need to correlate between the two.

This correlation is, in fact, very obvious. Since *'check()'* is responsible for extracting a specific bug out of the database, it uses the *'_load_from_db()'* function to do so. Because *'check()'* assumes hash arguments are entirely system controlled, and are not dangerous, it passes them as-is into *'_load_from_db()'*.
This allows us to exploit our SQLI freely, without any special permissions.

Unfortunately, Perl's "Taint Mode" restricts our injection.
Taint Mode is basically a "safe mode" for Perl applications that makes sure input must be validated prior to its usage in dangerous functions such as '*open()'*, *'eval()'*, or even DB functions such as *'selectrow()'*.
As with any language-specific, built-in-security-mechanism, "Taint Mode" cannot be counted on, mainly because it can be disabled, removed, or changed by 3rd party entities. Therefore, relying solely on it is extremely risky, just like relying on a WAF or other 3rd party software rather than securing your code.

That said, removing it is also a bad idea. As Mozilla already removed parts of it from some of Bugzilla's code and is considering setting it as an optional configuration value (see summary for more details) for the rest of the system, this vulnerability report is critical.

Without "Taint Mode", the severity of this vulnerability would be massive, effectively exposing the entire database to any attacker on any Bugzilla installation worldwide.

## POC

```
POST /jsonrpc.cgi HTTP/1.1
Host: localhost
Content-Type: application/json
Content-Length: XX

{"method":"Bug.update_attachment","params":[{
"Bugzilla_login": "USERNAME",
"Bugzilla_password": "PASSWORD",
"ids": [{"condition":"[SQLI]","values":["test"]}]
}]}
```

## Suggested Fix

The easiest fix is to restrict the use of non-scalar values in RPCs. A more sophisticated fix would require a function-by-function supervision to determine which variables require scalar values and which don't.