*Deliverable: Code Review Results Report*

# *Etho Protocol*

# *Security Code Review*

*Security Report*

*May 2021*

## Report Summary

| | | | |
|---|---|---|---|
| **Title** | **Etho Protocol Security Code Review** | | |
| **Project Owner** | Etho Protocol | | |
| | | | |
| **Type** | Public | | |
| **Version** | V1.4.5 | **Version date** | 11/02/2021 |
| **Reviewed by** | Vatsal Raychura | **Revision date** | 04/05/2021 |
| **Approved by** | eNebula Solutions Private Limited | **Approval date** | 05/05/2021 |
| | | **Nº Pages** | **34** |

# Overview

## Background

The Etho Protocol requested that eNebula Solutions perform a security code review audit of the Official Golang implementation of The Etho Protocol.

## Project Dates

The following is the project schedule for this review and report:

- **May 2 - 4**: Security Code Review Completed *(Completed)*
- **May 5**: Delivery of Security Code Review Report *(Completed)*

## Review Team

The following eNebula Solutions team member participated in this review:

- Vatsal Raychura, Security Researcher and Engineer

# Coverage

## Target Specification and Revision

For this audit, we performed research, investigation, and review of the source code of etho protocol followed by issue reporting, along with instructions outlined in this report.

The following documentation repositories were considered in-scope for the review:

- Ether1Project: https://github.com/Ether1Project/Ether1

## Main Areas of Security Concern

Our investigation mainly focused on the following security areas:

- Considerations for Auth
- Considerations for CSRF
- Considerations for Command Injection
- Considerations for Cookies
- Considerations for Cryptography
- Considerations for DoS
- Considerations for File access
- Considerations for HTTP
- Considerations for Input Validation
- Considerations for Insecure Modules Libraries
- Considerations for Insecure Storage
- Considerations for Malicious Code

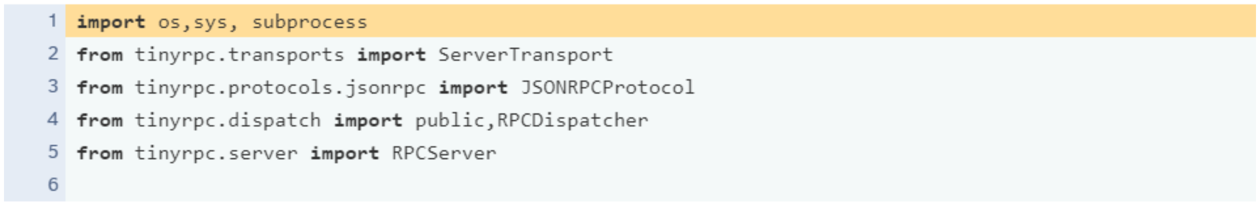
- Considerations for Mass Assignment
- Considerations for Regex
- Considerations for Routes
- Considerations for SQL Injection
- Considerations for SSL
- Considerations for Unexpected Behavior
- Considerations for Visibility
- Considerations for XSS
- Others.

# Findings

## Proof of Concept for the Security Issues from the Code review

### 1. cmd/clef/pythonsigner.py

- *Consider possible security implications associated with subprocess module.*
- *Insecure Modules Libraries - Consider possible security implications associated with some modules.*
- *B404: import_subprocess*



```python
import os,sys, subprocess
from tinyrpc.transports import ServerTransport
from tinyrpc.protocols.jsonrpc import JSONRPCProtocol
from tinyrpc.dispatch import public,RPCDispatcher
from tinyrpc.server import RPCServer

```

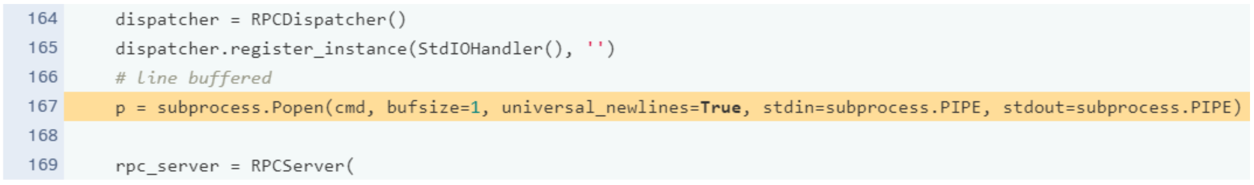
### Why is this an issue?

- Consider possible security implications associated with these modules.

| ID | Name | Imports | Severity |
|---|---|---|---|
| B404 | import_subprocess | subprocess | low |

## 2. cmd/clef/pythonsigner.py

- *Subprocess call - check for execution of untrusted input.*
- *Command Injection - Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system.*
- *B603: Test for use of subprocess without shell equals true*



```
dispatcher = RPCDispatcher()
dispatcher.register_instance(StdIOHandler(), '')
# line buffered
p = subprocess.Popen(cmd, bufsize=1, universal_newlines=True, stdin=subprocess.PIPE, stdout=subprocess.PIPE)

rpc_server = RPCServer(
```

### Why is this an issue?

- Python possesses many mechanisms to invoke an external executable. However, doing so may present a security issue if appropriate care is not taken to sanitize any user provided or variable input.
- This plugin test is part of a family of tests built to check for process spawning and warn appropriately. Specifically, this test looks for the spawning of a subprocess without the use of a command shell. This type of subprocess invocation is not vulnerable to shell injection attacks, but care should still be taken to ensure validity of input.
- Because this is a lesser issue than that described in subprocess_popen_with_shell_equals_true a LOW severity warning is reported.
- See also:
  - ../plugins/linux_commands_wildcard_injection
  - ../plugins/subprocess_popen_with_shell_equals_true
  - ../plugins/start_process_with_no_shell
  - ../plugins/start_process_with_a_shell
  - ../plugins/start_process_with_partial_path

### Config Options:

- This plugin test shares a configuration with others in the same family, namely shell_injection. This configuration is divided up into three sections, subprocess, shell and no_shell. They each list Python calls that spawn subprocesses, invoke commands within a shell, or invoke commands without a shell (by replacing the calling process) respectively.
- This plugin specifically scans for methods listed in subprocess section that have shell=False specified.

```
shell_injection:
    # Start a process using the subprocess module, or one of its
    wrappers.
    subprocess:
        - subprocess.Popen
        - subprocess.call
```

- **Example**

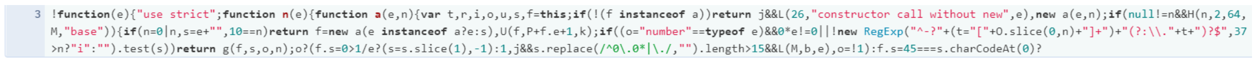
```
>> Issue: subprocess call - check for execution of untrusted input.
   Severity: Low   Confidence: High
   Location: ./examples/subprocess_shell.py:23
22
23    subprocess.check_output(['/bin/ls', '-l'])
24
```

- **See also**

  - https://security.openstack.org
  - https://docs.python.org/3/library/subprocess.html#frequently-used-arguments
  - https://security.openstack.org/guidelines/dg_avoid-shell-true.html
  - https://security.openstack.org/guidelines/dg_use-subprocess-securely.html
  - New in version 0.9.0.

## 3. signer/rules/deps/bignumber.js

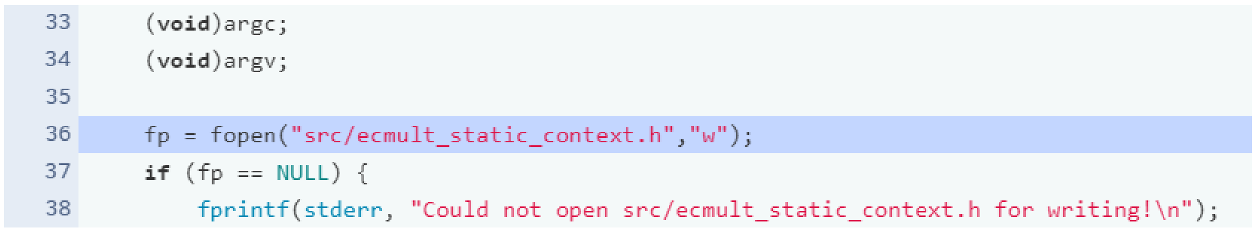
- *Unsafe Regular Expression*
- *Regex - Regex can be used in a Denial-of-Service attack, that exploits the fact that most Regular Expression implementations may reach heavy computation situations that cause them to work very slowly (exponentially related to input size).*



```
!function(e){"use strict";function n(e){function a(e,n){var t,r,i,o,u,s,f=this;if(!(f instanceof a))return j&&L(26,"constructor call without new",e),new a(e,n);if(null!=n&&H(n,2,64,
M,"base")){if(n=0|n,s=e+"",10==n)return f=new a(e instanceof a?e:s),U(f,P+f.e+1,k);if((o="number"==typeof e)&&0*e!=0||!new RegExp("^-?"+(t="["+O.slice(0,n)+"]+")+"(?:\\."+t+")?$",37
>n?"i":"").test(s))return g(f,s,o,n);o?(f.s=0>1/e?(s=s.slice(1),-1):1,j&&s.replace(/^0\.0*|\./,"").length>15&&L(M,b,e),o=!1):f.s=45===s.charCodeAt(0)?
```

## 4. crypto/secp256k1/libsecp256k1/src/gen_context.c

- *Check when opening files - can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents? (CWE-362).*
- *DoS*
- *The Denial of Service (DoS) attack is focused on making a resource (site, application, server) unavailable for the purpose it was designed.*

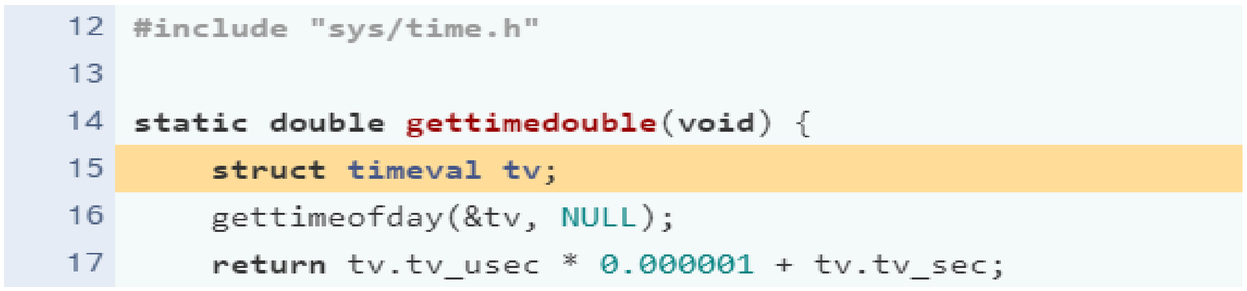


```
    (void)argc;
    (void)argv;

    fp = fopen("src/ecmult_static_context.h","w");
    if (fp == NULL) {
        fprintf(stderr, "Could not open src/ecmult_static_context.h for writing!\n");
```

5. crypto/secp256k1/libsecp256k1/src/bench.h

5.1

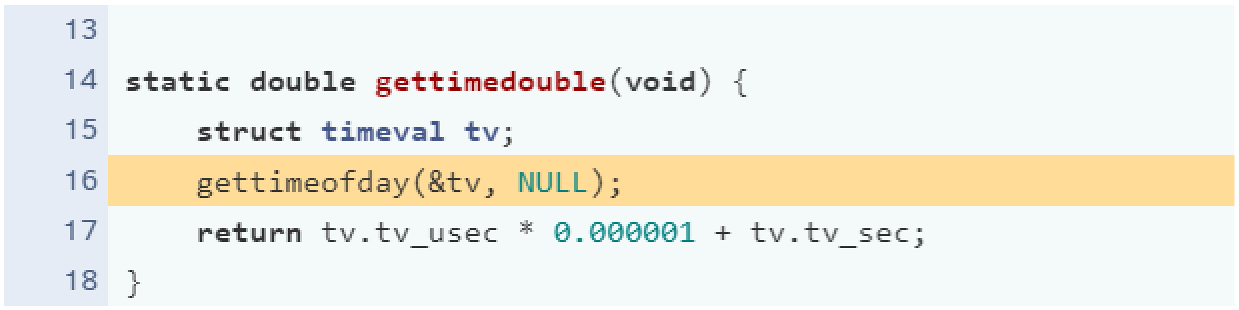
- *timeval is Y2038-unsafe*
- *cppcheck_y2038-unsafe-call*



```
#include "sys/time.h"

static double gettimedouble(void) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_usec * 0.000001 + tv.tv_sec;
```

5.2

- *gettimeofday is Y2038-unsafe*
- *cppcheck_y2038-unsafe-call*

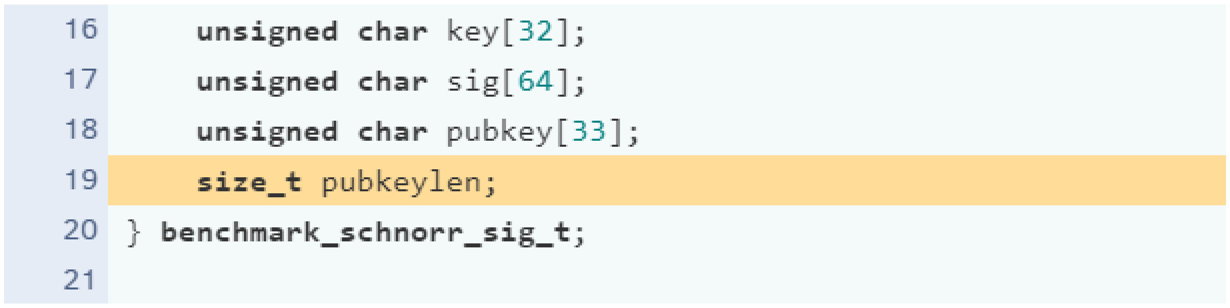


```

static double gettimedouble(void) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_usec * 0.000001 + tv.tv_sec;
}
```

6. crypto/secp256k1/libsecp256k1/src/bench_schnorr_verify.c

### 6.1

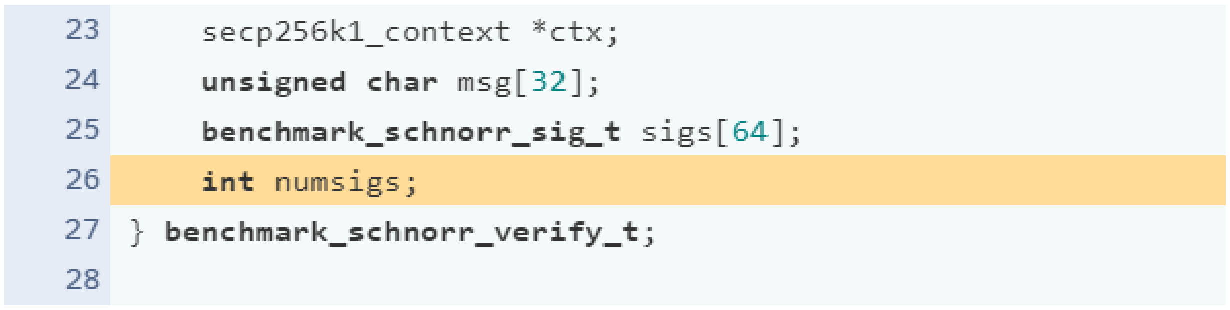
- *Avoid laying out strings in memory directly before sensitive data*
- *cppcheck_cert-API01-C*



```
    unsigned char key[32];
    unsigned char sig[64];
    unsigned char pubkey[33];
    size_t pubkeylen;
} benchmark_schnorr_sig_t;

```

### 6.2

- *Avoid laying out strings in memory directly before sensitive data*
- *cppcheck_cert-API01-C*

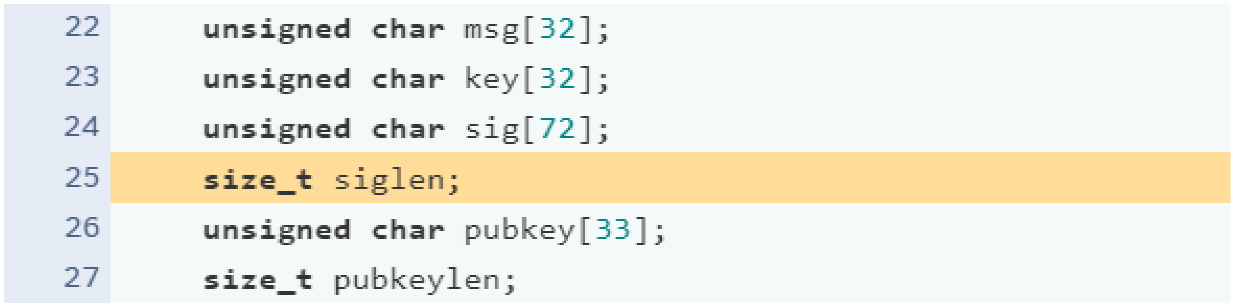


```
    secp256k1_context *ctx;
    unsigned char msg[32];
    benchmark_schnorr_sig_t sigs[64];
    int numsigs;
} benchmark_schnorr_verify_t;

```

## 7. crypto/secp256k1/libsecp256k1/src/bench_verify.c

### 7.1

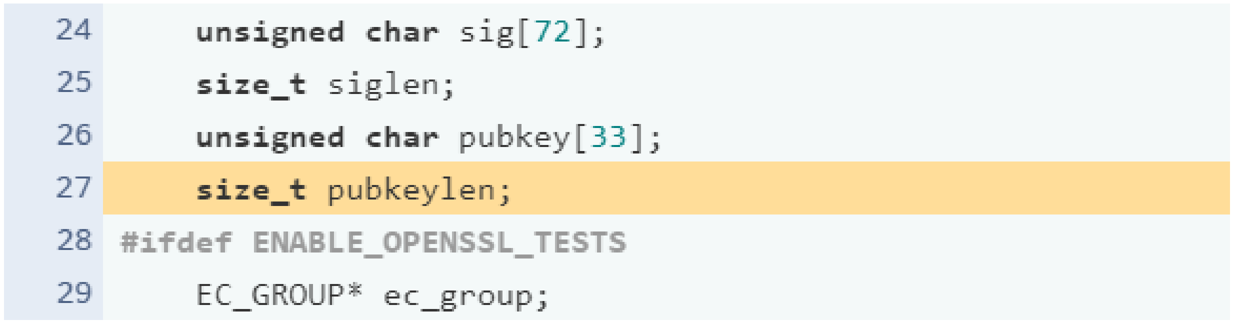
- *Avoid laying out strings in memory directly before sensitive data*
- *cppcheck_cert-API01-C*



```c
    unsigned char msg[32];
    unsigned char key[32];
    unsigned char sig[72];
    size_t siglen;
    unsigned char pubkey[33];
    size_t pubkeylen;
```

### 7.2

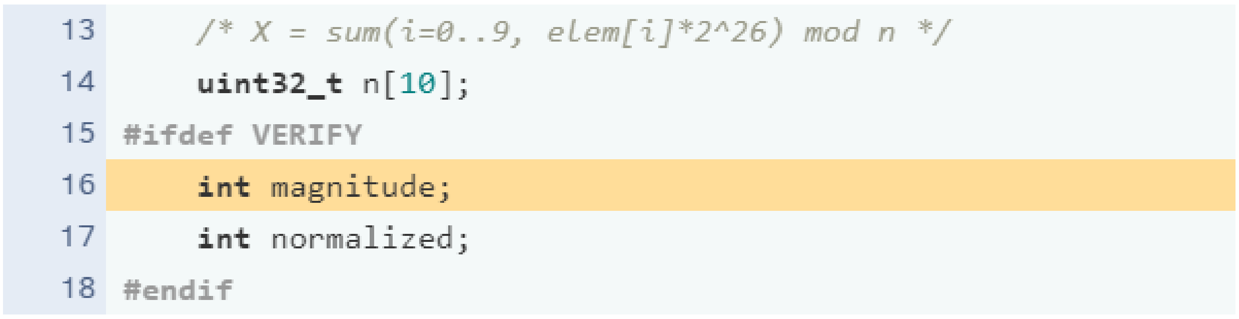
- *Avoid laying out strings in memory directly before sensitive data*
- *cppcheck_cert-API01-C*



```c
    unsigned char sig[72];
    size_t siglen;
    unsigned char pubkey[33];
    size_t pubkeylen;
#ifdef ENABLE_OPENSSL_TESTS
    EC_GROUP* ec_group;
```

8. crypto/secp256k1/libsecp256k1/src/field_10x26.h

- *Avoid laying out strings in memory directly before sensitive data*
- *cppcheck_cert-API01-C*

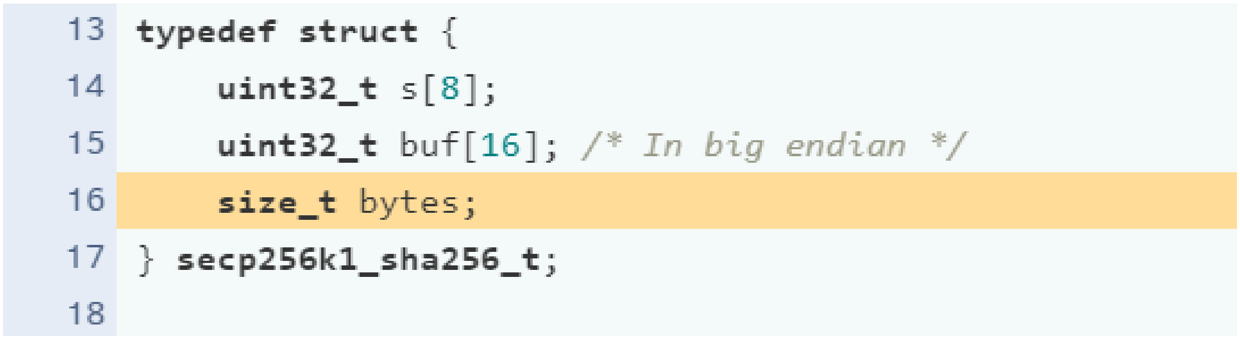


```
    /* X = sum(i=0..9, elem[i]*2^26) mod n */
    uint32_t n[10];
#ifdef VERIFY
    int magnitude;
    int normalized;
#endif
```

9. crypto/secp256k1/libsecp256k1/src/hash.h

9.1

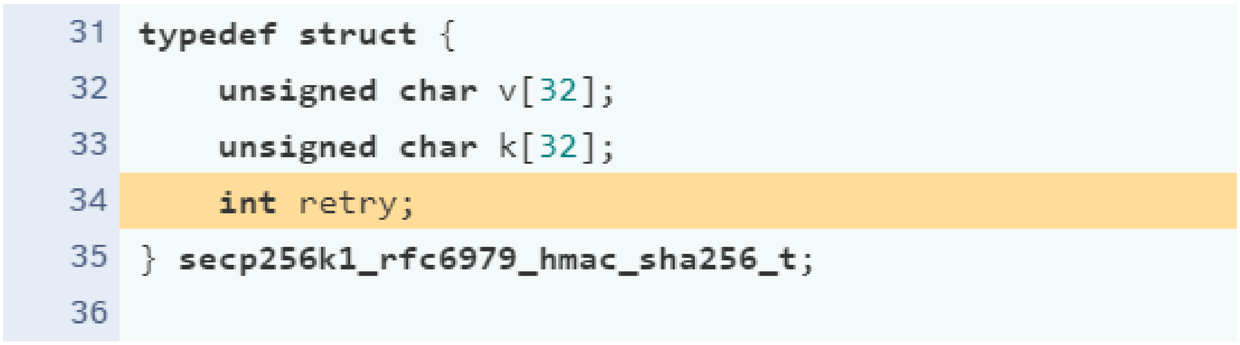
- *Avoid laying out strings in memory directly before sensitive data*
- *cppcheck_cert-API01-C*



```
typedef struct {
    uint32_t s[8];
    uint32_t buf[16]; /* In big endian */
    size_t bytes;
} secp256k1_sha256_t;

```

9.2

- *Avoid laying out strings in memory directly before sensitive data*
- *cppcheck_cert-API01-C*

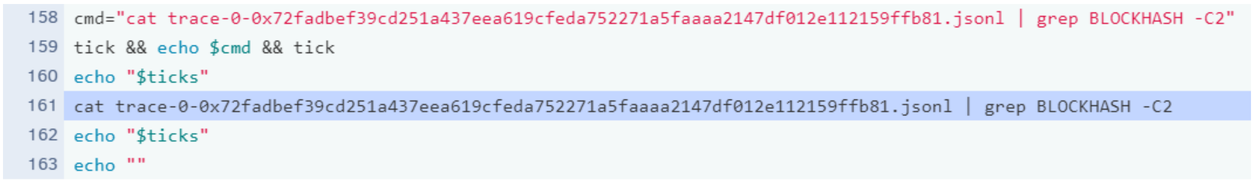


```
typedef struct {
    unsigned char v[32];
    unsigned char k[32];
    int retry;
} secp256k1_rfc6979_hmac_sha256_t;

```

## Proof of Concept for the Performance Issue from the Code review

1. cmd/evm/transition-test.sh

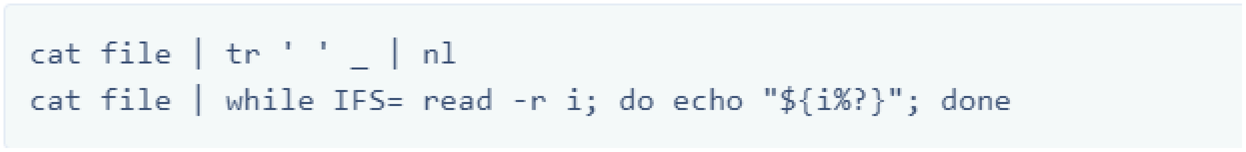
   - *Useless cat. Consider 'cmd < file | ..' or 'cmd file | ..' instead.*



```
cmd="cat trace-0-0x72fadbef39cd251a437eea619cfeda752271a5faaaa2147df012e112159ffb81.jsonl | grep BLOCKHASH -C2"
tick && echo $cmd && tick
echo "$ticks"
cat trace-0-0x72fadbef39cd251a437eea619cfeda752271a5faaaa2147df012e112159ffb81.jsonl | grep BLOCKHASH -C2
echo "$ticks"
echo ""
```

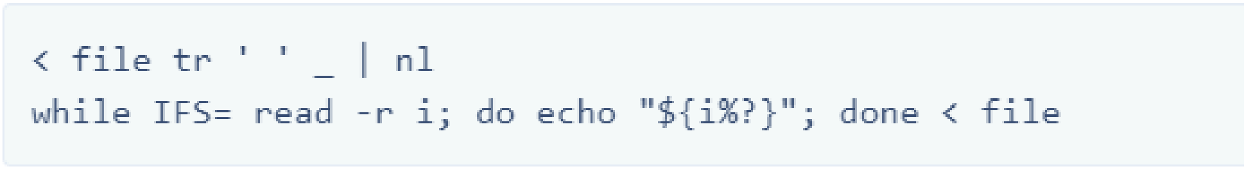
**Why is this an issue?**

- **Problematic code:**



```
cat file | tr ' ' _ | nl
cat file | while IFS= read -r i; do echo "${i%?}"; done
```

- **Correct code:**



```
< file tr ' ' _ | nl
while IFS= read -r i; do echo "${i%?}"; done < file
```

- **Rationale:**

  - ➢ cat is a tool for con"cat"enating files. Reading a single file as input to a program is considered a Useless Use of Cat (UUOC).
  - ➢ It's more efficient and less roundabout to simply use redirection. This is especially true for programs that can benefit from seekable input, like tail or tar.
  - ➢ Many tools also accept optional filenames, e.g., grep -q foo file instead of cat file | grep -q foo.

- **Exceptions:**

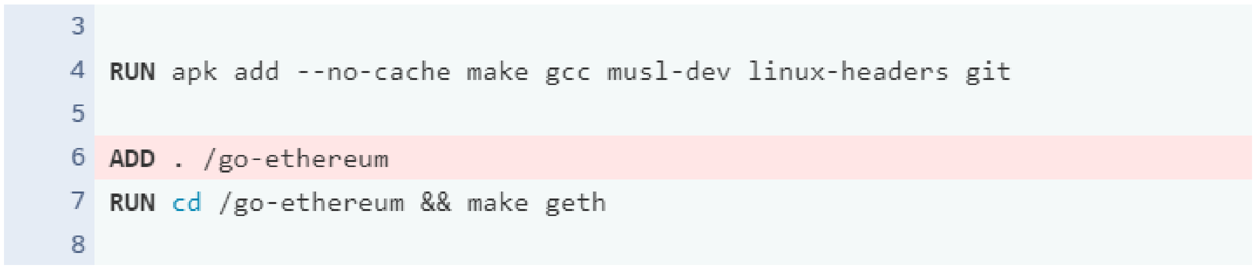
  Pointing out UUOC is a long-standing shell programming tradition, and removing them from a short-lived pipeline in a loop can speed it up by 2x. However, it's not necessarily a good use of time in practice, and rarely affects correctness. [[Ignore]] as you see fit.

# Proof of Concept for the Error Prone Issues from the Code review

1. Dockerfile

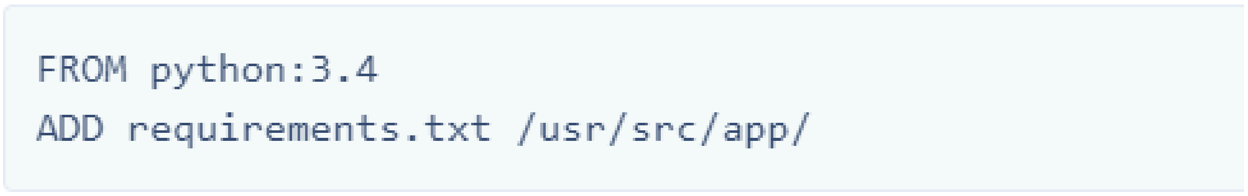
1.1

- *Use COPY instead of ADD for files and folders*



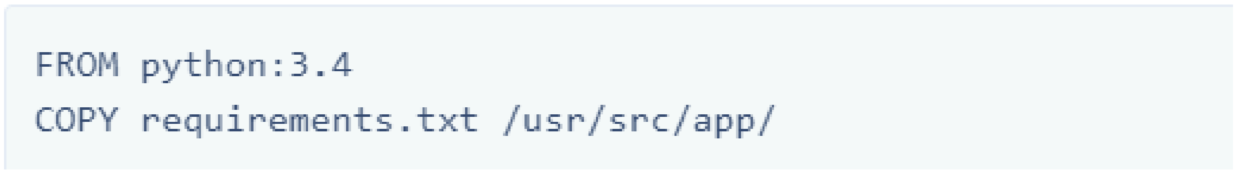
```

RUN apk add --no-cache make gcc musl-dev linux-headers git

ADD . /go-ethereum
RUN cd /go-ethereum && make geth

```

**Problematic code:**



```
FROM python:3.4
ADD requirements.txt /usr/src/app/
```

**Correct code:**



```
FROM python:3.4
COPY requirements.txt /usr/src/app/
```

**Rationale:**

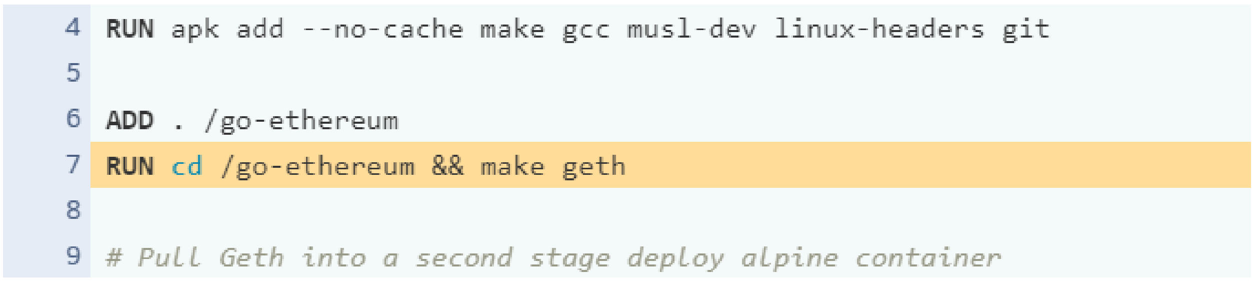
- https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#add-or-copy
- *For other items (files, directories) that do not require ADD's tar auto-extraction capability, you should always use COPY.*
- Rule also implemented in https://github.com/RedCoolBeans/dockerlint/blob/master/src/checks.coffee

**Exceptions:**

*Consequently, the best use for ADD is local tar file auto-extraction into the image.*

1.2

- *Use WORKDIR to switch to a directory*



```
RUN apk add --no-cache make gcc musl-dev linux-headers git

ADD . /go-ethereum
RUN cd /go-ethereum && make geth

# Pull Geth into a second stage deploy alpine container
```

**Problematic code:**

```
FROM busybox
RUN cd /usr/src/app && git clone git@github.com:lukasmartinelli/hadolint.git
```

**Correct code:**

```
FROM busybox
RUN git clone git@github.com:lukasmartinelli/hadolint.git /usr/src/app
```

**Rationale:**

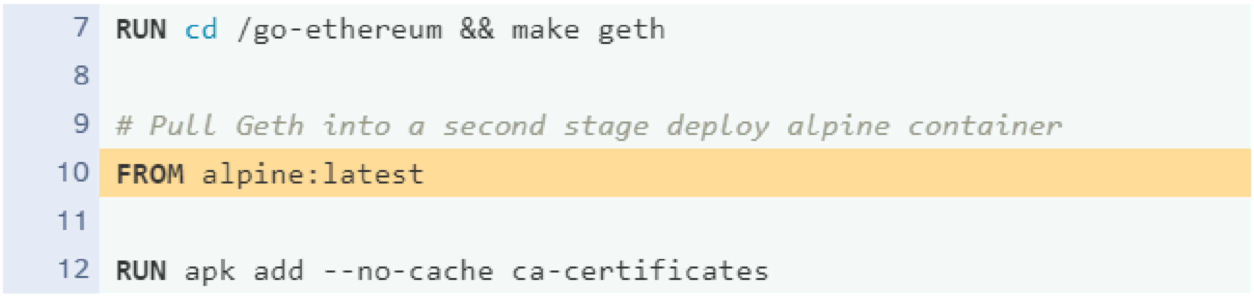
Only use cd in a subshell. Most commands can work with absolute paths and it in most cases not necessary to change directories. Docker provides the WORKDIR instruction if you really need to change the current working directory.

**Exceptions:**

When executed in a Subshell.

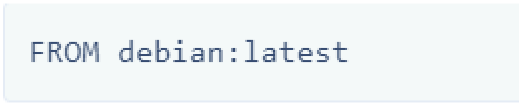
1.3

- *Using latest is prone to errors if the image will ever update. Pin the version explicitly to a release tag.*



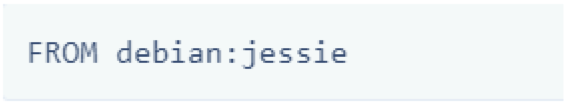
```
RUN cd /go-ethereum && make geth

# Pull Geth into a second stage deploy alpine container
FROM alpine:latest

RUN apk add --no-cache ca-certificates
```

**Problematic code:**



```
FROM debian:latest
```

**Correct code:**



```
FROM debian:jessie
```

**Rationale:**

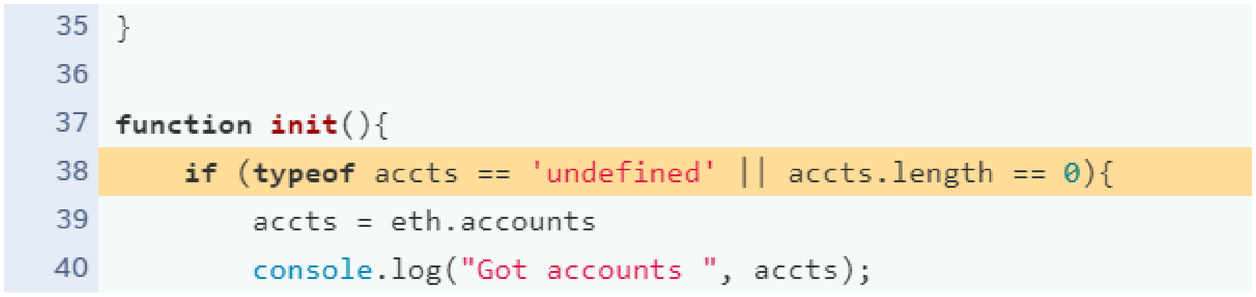
- ➢ You can never rely that the latest tags are a specific version.
- ➢ https://docs.docker.com/engine/userguide/dockerimages/

*Tip: You recommend you always use a specific tagged image, for example ubuntu:12.04. That way you always know exactly what variant of an image is being used.*

## 2. cmd/clef/tests/testsigner.js

### 2.1

- *Use ===/!== to compare with true/false or Numbers*



```
}

function init(){
    if (typeof accts == 'undefined' || accts.length == 0){
        accts = eth.accounts
        console.log("Got accounts ", accts);
```

**Why is this an issue?**

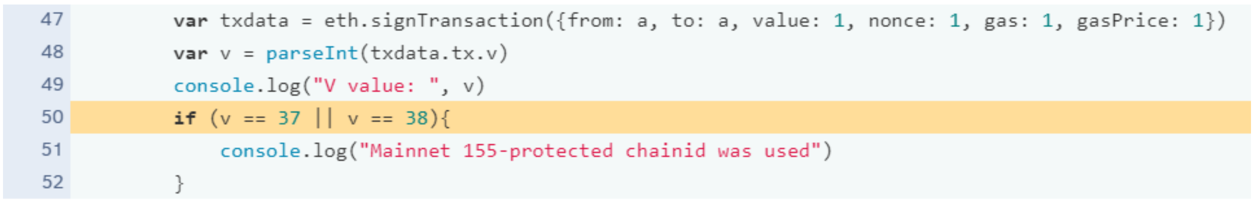
- Since: PMD 5.0
- Using == in condition may lead to unexpected results, as the variables are automatically casted to be of the same type. The === operator avoids the casting.

**Example(s):**

```
// Ok
if (someVar === true) {
  ...
}
// Ok
if (someVar !== 3) {
  ...
}
// Bad
if (someVar == true) {
  ...
}
// Bad
if (someVar != 3) {
  ...
}
```

## 2.2

- *Use ===/!== to compare with true/false or Numbers*



```
        var txdata = eth.signTransaction({from: a, to: a, value: 1, nonce: 1, gas: 1, gasPrice: 1})
        var v = parseInt(txdata.tx.v)
        console.log("V value: ", v)
        if (v == 37 || v == 38){
            console.log("Mainnet 155-protected chainid was used")
        }
```

### Why is this an issue?

- ➢ Since: PMD 5.0
- ➢ Using == in condition may lead to unexpected results, as the variables are automatically casted to be of the same type. The === operator avoids the casting.

### Example(s):

```
// Ok
if (someVar === true) {
  ...
}
// Ok
if (someVar !== 3) {
  ...
}
// Bad
if (someVar == true) {
  ...
}
// Bad
if (someVar != 3) {
  ...
}
```

## 2.3

- *Use ===/!== to compare with true/false or Numbers*

```
        if (v == 37 || v == 38){
            console.log("Mainnet 155-protected chainid was used")
        }
        if (v == 27 || v == 28){
            throw new Error("Mainnet chainid was used, but without replay protection!")
        }
```

### Why is this an issue?

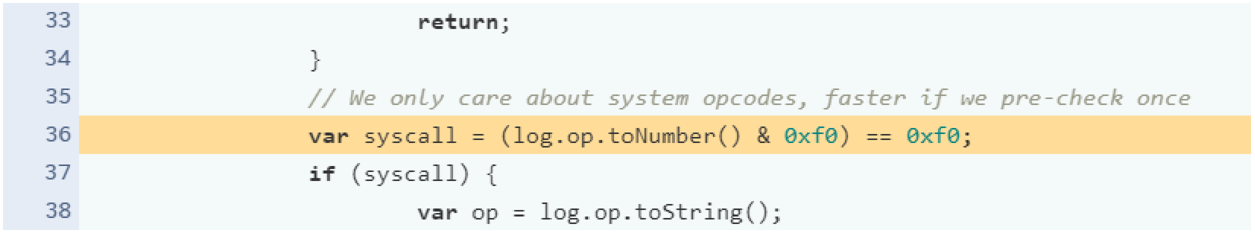
- Since: PMD 5.0
- Using == in condition may lead to unexpected results, as the variables are automatically casted to be of the same type. The === operator avoids the casting.

### Example(s):

```
// Ok
if (someVar === true) {
  ...
}
// Ok
if (someVar !== 3) {
  ...
}
// Bad
if (someVar == true) {
  ...
}
// Bad
if (someVar != 3) {
  ...
}
```

3. eth/tracers/internal/tracers/call_tracer.js

- *Use ===/!== to compare with true/false or Numbers*



```
                return;
            }
            // We only care about system opcodes, faster if we pre-check once
            var syscall = (log.op.toNumber() & 0xf0) == 0xf0;
            if (syscall) {
                var op = log.op.toString();
```

## Why is this an issue?

- Since: PMD 5.0
- Using == in condition may lead to unexpected results, as the variables are automatically casted to be of the same type. The === operator avoids the casting.

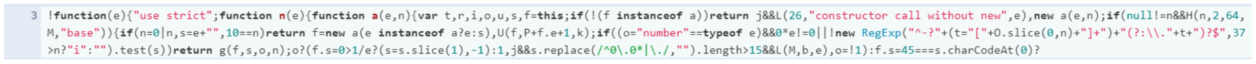
## Example(s):

```
// Ok
if (someVar === true) {
  ...
}
// Ok
if (someVar !== 3) {
  ...
}
// Bad
if (someVar == true) {
  ...
}
// Bad
if (someVar != 3) {
  ...
}
```

## 4. internal/jsre/deps/bignumber.js

### 4.1

- *The numeric literal '9e15' will have at different value at runtime.*
- *Inaccurate Numeric Literal -- The numeric literal ''{0}'' will have at different value at runtime.*

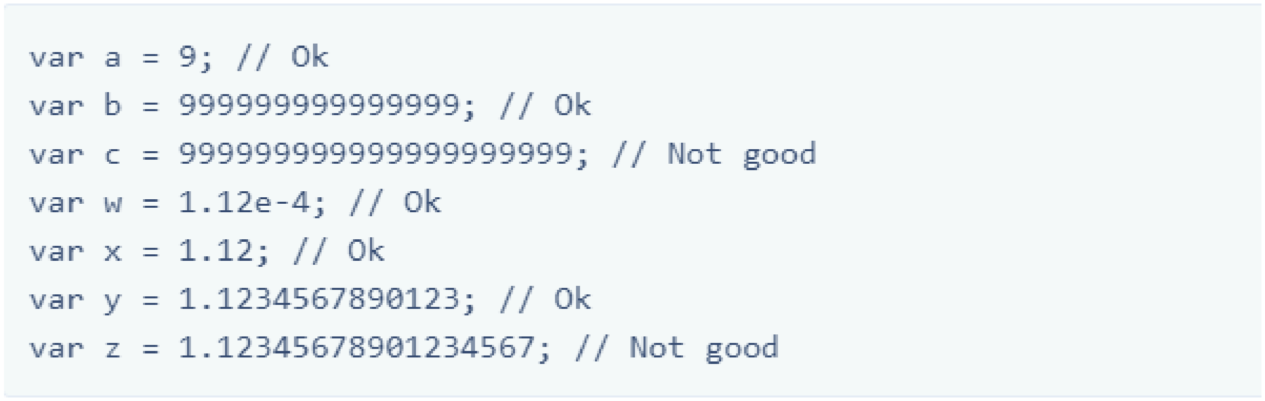


```
!function(e){"use strict";function n(e){function a(e,n){var t,r,i,o,u,s,f=this;if(!(f instanceof a))return j&&L(26,"constructor call without new",e),new a(e,n);if(null!=n&&H(n,2,64,
M,"base")){if(n=0|n,s=e+"",10==n)return f=new a(e instanceof a?e:s),U(f,P+f.e+1,k);if((o="number"==typeof e)&&0*e!=0||!new RegExp("^-?"+(t="["+O.slice(0,n)+"]+")+"(?:\\."+t+")?$",37
>n?"i":"").test(s))return g(f,s,o,n);o?(f.s=0>1/e?(s=s.slice(1),-1):1,j&&s.replace(/^0\.0*|\./,"").length>15&&L(M,b,e),o=!1):f.s=45===s.charCodeAt(0)?
```

### Why is this an issue?

- Since: PMD 5.0
- The numeric literal will have a different value at runtime, which can happen if you provide too much precision in a floating-point number. This may result in numeric calculations being in error.

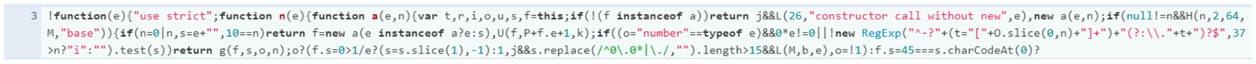
### Example(s):



```
var a = 9; // Ok
var b = 999999999999999; // Ok
var c = 999999999999999999999; // Not good
var w = 1.12e-4; // Ok
var x = 1.12; // Ok
var y = 1.1234567890123; // Ok
var z = 1.12345678901234567; // Not good
```

## 4.2

- *Use ===/!== to compare with true/false or Numbers*



```
!function(e){"use strict";function n(e){function a(e,n){var t,r,i,o,u,s,f=this;if(!(f instanceof a))return j&&L(26,"constructor call without new",e),new a(e,n);if(null!=n&&H(n,2,64,
M,"base")){if(n=0|n,s=e+"",10==n)return f=new a(e instanceof a?e:s),U(f,P+f.e+1,k);if((o="number"==typeof e)&&0*e!=0||!new RegExp("^-?"+(t="["+O.slice(0,n)+"]+")+"(?:\\."+t+")?$",37
>n?"i":"").test(s))return g(f,s,o,n);o?(f.s=0>1/e?(s=s.slice(1),-1):1,j&&s.replace(/^0\.0*|\./,"").length>15&&L(M,b,e),o=!1):f.s=45===s.charCodeAt(0)?
```

## Why is this an issue?

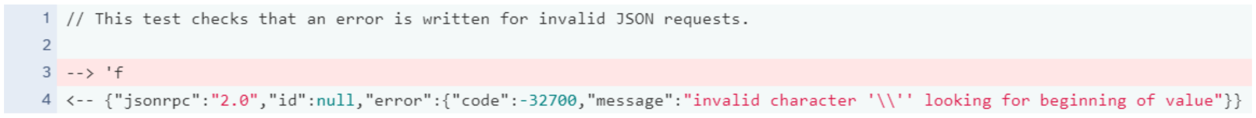
- Since: PMD 5.0
- Using == in condition may lead to unexpected results, as the variables are automatically casted to be of the same type. The === operator avoids the casting.

## Example(s):

```
// Ok
if (someVar === true) {
  ...
}
// Ok
if (someVar !== 3) {
  ...
}
// Bad
if (someVar == true) {
  ...
}
// Bad
if (someVar != 3) {
  ...
}
```

5. rpc/testdata/invalid-syntax.json

- *Unexpected character ('-' (code 45)) in numeric value: expected digit (0-9) to follow minus sign, for valid numeric value*
- *Prohibit invalid JSON files -- Prohibits unparsable or invalid JSON files*



```
// This test checks that an error is written for invalid JSON requests.

--> 'f
<-- {"jsonrpc":"2.0","id":null,"error":{"code":-32700,"message":"invalid character '\\'' looking for beginning of value"}}
```

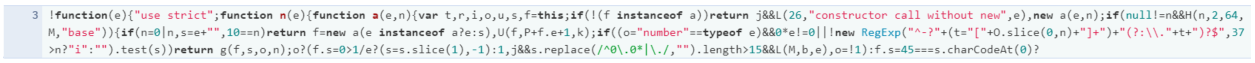
### Why is this an issue?

- ➢ At Codacy we strive to provide great descriptions for our patterns. With good explanations developers can better understand issues and even learn how to fix them.
- ➢ For this tool we are not yet meeting this standard but you can help us improve the docs. To know more, take a look at our tool documentation guide.
- ➢ You can also visit the tool's website to find useful tips about the patterns.

## 6. signer/rules/deps/bignumber.js

### 6.1

- *Use ===/!== to compare with true/false or Numbers*



```
!function(e){"use strict";function n(e){function a(e,n){var t,r,i,o,u,s,f=this;if(!(f instanceof a))return j&&L(26,"constructor call without new",e),new a(e,n);if(null!=n&&H(n,2,64,
M,"base")){if(n=0|n,s=e+"",10==n)return f=new a(e instanceof a?e:s),U(f,P+f.e+1,k);if((o="number"==typeof e)&&0*e!=0||!new RegExp("^-?"+(t="["+O.slice(0,n)+"]+")+"(?:\\."+t+")?$",37
>n?"i":"").test(s))return g(f,s,o,n);o?(f.s=0>1/e?(s=s.slice(1),-1):1,j&&s.replace(/^0\.0*|\./,"").length>15&&L(M,b,e),o=!1):f.s=45===s.charCodeAt(0)?
```

### Why is this an issue?

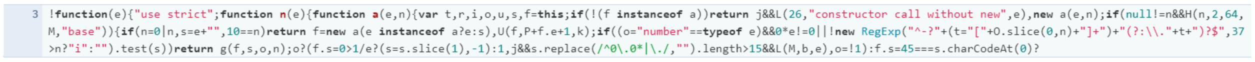
- Since: PMD 5.0
- Using == in condition may lead to unexpected results, as the variables are automatically casted to be of the same type. The === operator avoids the casting.

### Example(s):

```
// Ok
if (someVar === true) {
  ...
}
// Ok
if (someVar !== 3) {
  ...
}
// Bad
if (someVar == true) {
  ...
}
// Bad
if (someVar != 3) {
  ...
}
```

## 6.2

- *The numeric literal '9e15' will have at different value at runtime.*
- *Inaccurate Numeric Literal -- The numeric literal ''{0}'' will have at different value at runtime.*

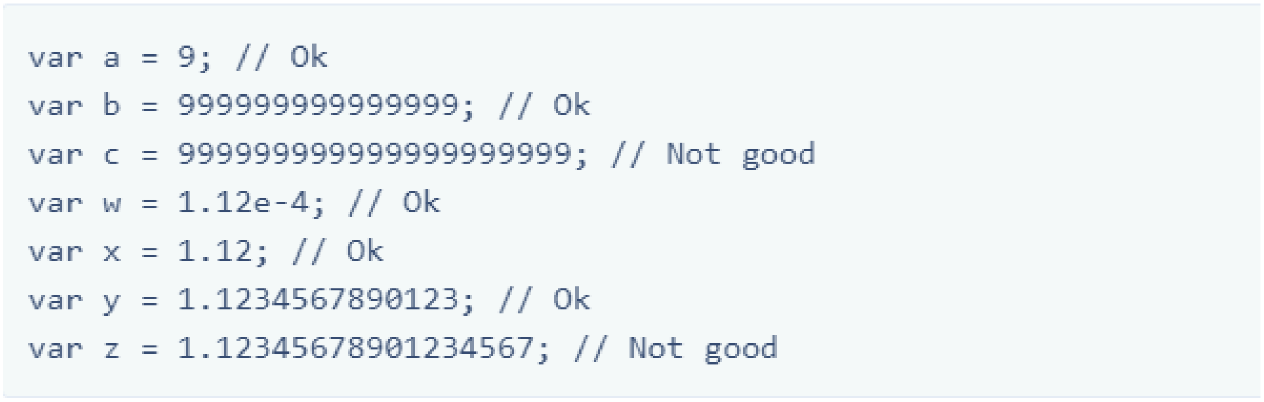


```
!function(e){"use strict";function n(e){function a(e,n){var t,r,i,o,u,s,f=this;if(!(f instanceof a))return j&&L(26,"constructor call without new",e),new a(e,n);if(null!=n&&H(n,2,64,
M,"base")){if(n=0|n,s=e+"",10==n)return f=new a(e instanceof a?e:s),U(f,P+f.e+1,k);if((o="number"==typeof e)&&0*e!=0||!new RegExp("^-?"+(t="["+O.slice(0,n)+"]+")+"(?:\\."+t+")?$",37
>n?"i":"").test(s))return g(f,s,o,n);o?(f.s=0>1/e?(s=s.slice(1),-1):1,j&&s.replace(/^0\.0*|\./,"").length>15&&L(M,b,e),o=!1):f.s=45===s.charCodeAt(0)?
```

### Why is this an issue?

- Since: PMD 5.0
- The numeric literal will have a different value at runtime, which can happen if you provide too much precision in a floating-point number. This may result in numeric calculations being in error.

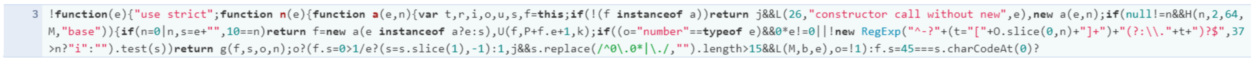
### Example(s):



```
var a = 9; // Ok
var b = 999999999999999; // Ok
var c = 999999999999999999999; // Not good
var w = 1.12e-4; // Ok
var x = 1.12; // Ok
var y = 1.1234567890123; // Ok
var z = 1.12345678901234567; // Not good
```

6.3

- *Empty block statement.*
- *No empty -- Disallow empty block statements*



```
!function(e){"use strict";function n(e){function a(e,n){var t,r,i,o,u,s,f=this;if(!(f instanceof a))return j&&L(26,"constructor call without new",e),new a(e,n);if(null!=n&&H(n,2,64,
M,"base")){if(n=0|n,s=e+"",10==n)return f=new a(e instanceof a?e:s),U(f,P+f.e+1,k);if((o="number"==typeof e)&&0*e!=0||!new RegExp("^-?"+(t="["+O.slice(0,n)+"]+")+"(?:\\."+t+")?$",37
>n?"i":"").test(s))return g(f,s,o,n);o?(f.s=0>1/e?(s=s.slice(1),-1):1,j&&s.replace(/^0\.0*|\./,"").length>15&&L(M,b,e),o=!1):f.s=45===s.charCodeAt(0)?
```

## Why is this an issue?

### disallow empty block statements (no-empty)

Empty block statements, while not technically errors, usually occur due to refactoring that wasn't completed. They can cause confusion when reading code.

### Rule Details

This rule disallows empty block statements. This rule ignores block statements which contain a comment (for example, in an empty catch or finally block of a try statement to indicate that execution should continue regardless of errors).

### Examples of incorrect code for this rule:

```
/*eslint no-empty: "error"*/

if (foo) {
}

while (foo) {
}

switch(foo) {
}

try {
    doSomething();
} catch(ex) {

} finally {

}
```

Examples of correct code for this rule:

```
/*eslint no-empty: "error"*/

if (foo) {
    // empty
}

while (foo) {
    /* empty */
}

try {
    doSomething();
} catch (ex) {
    // continue regardless of error
}

try {
    doSomething();
} finally {
    /* continue regardless of error */
}
```

## Options

This rule has an object option for exceptions:

- "allowEmptyCatch": true allows empty catch clauses (that is, which do not contain a comment)

## allowEmptyCatch

**Examples of additional correct code for this rule with the { "allowEmptyCatch": true } option:**

```
/* eslint no-empty: ["error", { "allowEmptyCatch": true }] */
try {
    doSomething();
} catch (ex) {}

try {
    doSomething();
}
catch (ex) {}
finally {
    /* continue regardless of error */
}
```

## When Not to Use It

If you intentionally use empty block statements then you can disable this rule.

## Related Rules

no-empty-function

# About eNebula Solutions

We believe that people have a fundamental need to security and that the use of secure solutions enables every person to more freely use the Internet and every other connected technology. We aim to provide security consulting service to help others make their solutions more resistant to unauthorized access to data & inadvertent manipulation of the system. We support teams from the design phase through the production to launch and surely after.

The eNebula Solutions team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities & specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including incryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize varioustools to scan code & networks and build custom tools as necessary.

Although we are a small team, we surely believe that we can have a momentous impact on the world by being translucent and open about the work we do.

For more information about our security consulting, please
mail us at – contact@enebula.in.

# Our Methodology

We wish to work with a clear method and build our reviews a cooperative effort. The goals of our security audits are to boost the standard of systems we tend to review and aim for adequate remediation to assist protect users. The subsequent is that the methodology suggested by synopsys (synopsys.com) we tend to use in our security code review audit method.

1. **Finalize the tool.** Select a static analysis tool that can perform code reviews of applications written in the programming languages you use. The tool should also be able to comprehend the underlying framework used by your software.

2. **Create the scanning infrastructure, and deploy the tool.** This step involves handling the licensing requirements, setting up access control and authorization, and procuring the resources required (e.g., servers and databases) to deploy the tool.

3. **Customize the tool.** Fine-tune the tool to suit the needs of the organization. For example, you might configure it to reduce false positives or find additional security vulnerabilities by writing new rules or updating existing ones. Integrate the tool into the build environment, create dashboards for tracking scan results, and build custom reports.

4. **Prioritize and onboard applications.** Once the tool is ready, onboard your applications. If you have a large number of applications, prioritize the high-risk applications to scan first. Eventually, all your applications should be onboarded and scanned regularly, with application scans synced with

release cycles, daily or monthly builds, or code check-ins.

5. **Analyze scan results.** This step involves triaging the results of the scan to remove false positives. Once the set of issues is finalized, they should be tracked and provided to the deployment teams for proper and timely remediation.

6. **Provide governance and training.** Proper governance ensures that your development teams are employing the scanning tools properly. The software security touchpoints should be present within the SDLC. SAST should be incorporated as part of your application development and deployment process.