How “assertions” can get you Hacked !!

A deep dive into the assert() function and ways to exploit it!

Mayank Pandey · Follow
Published in InfoSec Write-ups
7 min read · Dec 22, 2021

Hello Fellow Hackers and developers, It’s been a while since I posted any blog here. Today I will be discussing **Coding+Hacking**, especially about a feature present in almost every High-Level Language known as “**assertion**”. You will get to know What are assert() functions and different ways to exploit them in the wild. So hang tight and let's dive in.

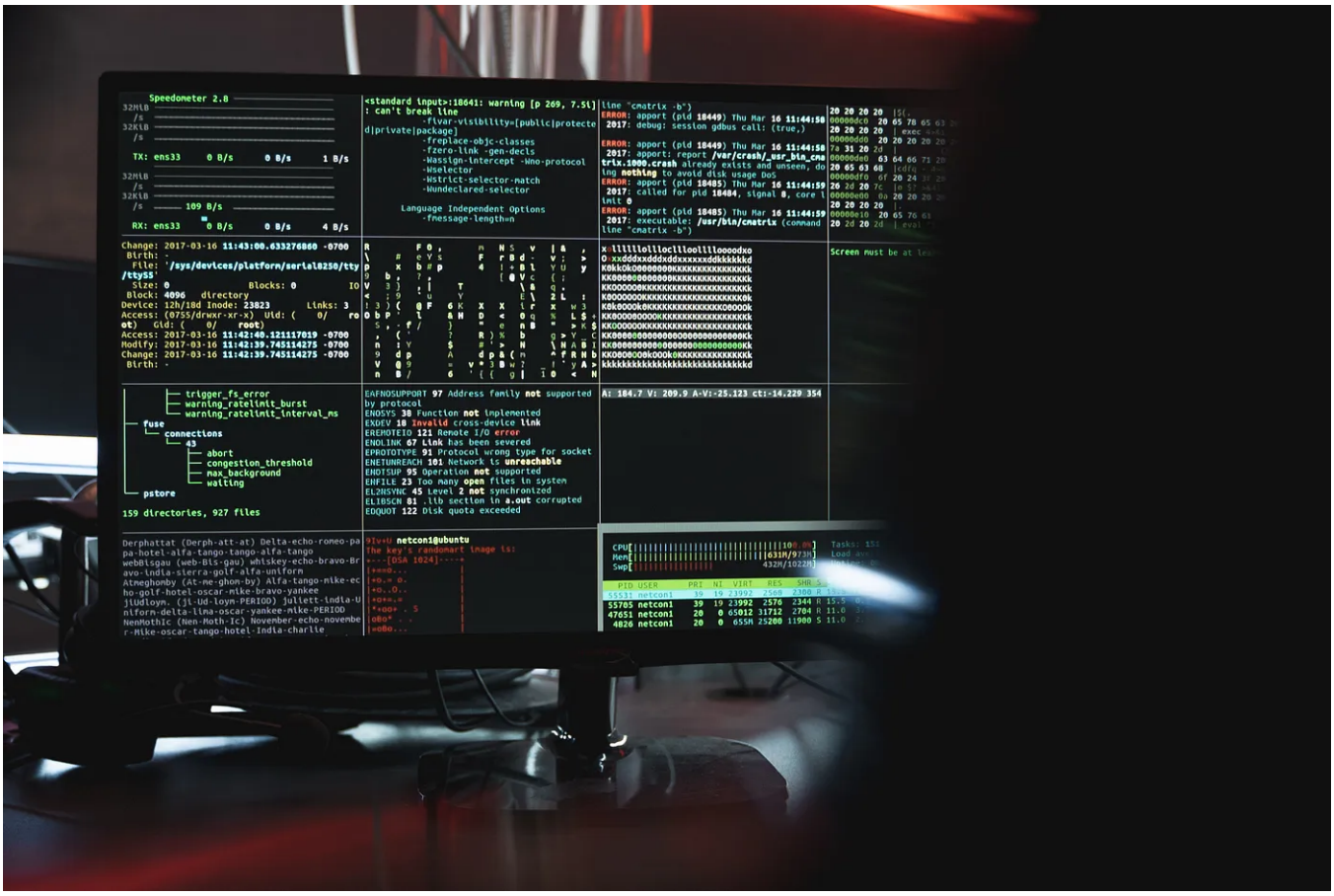

Photo by Tima Miroshnichenko from Pexels

What is an Assertion?

The Literal Meaning of Assertion is *“a statement that says you strongly believe that something is true”,* similarly sometimes in programming, we assume some values to be true and then Proceed with the rest of the code.

For example, we can assert **that any request hitting the local SQL server must be coming from a local host thus no authentication is needed to proceed further,** this whole statement already started to tickle your hacking senses and you can think of many ways in which this assertion can go wrong.

There can be many different other scenarios in which assert() can be used to capture your assumptions in the code.

assert() checks for a condition to be true and if it is not true then it raises an error and closes the program.

How is assert() different from an if-else block?

You may question that all this assert() function is doing that it's checking for something and then acting accordingly, so why not use if-else instead of this

assert is to document your assumptions in the code. if-else statement is to handle different logical scenarios. Let's understand it from a simple example

For example when you write

```
1  void print_number(int* somePtr) {
2    assert (somePtr!=NULL);
3    printf ("%d\n",*somePtr);
4  }
```

assert.cpp hosted with ❤ by GitHub view raw

you mean to say that, In my **print_number** function, I assume that always the pointer coming is not null. I would be very very surprised if this is null. I don’t care to handle this scenario at all in my code.

But, if you write

```
1  void print_number(int* somePtr) {
2    if (somePtr != NULL)
3         printf ("%d\n",*somePtr);
4    // else do something
5  }
```

if-else.cpp hosted with ❤ by GitHub view raw

You seem to say that, in my **print_number** function, I expect people to pass a null pointer. And I know how to handle this situation and I do handle this with an else condition.

So, sometimes you will know how to handle certain situations and you want to do that. Then, use if-else. Sometimes, you assume that something will not happen and you don’t care to handle it. You just express your surprise and stop your program execution there with assert.

Security Pitfalls related to assert()!

assert() works differently in different Programming languages but then also it keeps being a Security related issue.

1: Access Control Bugs

This is found mainly in python due to running code in optimized mode.

Now, by default Python executes with `__debug__` as true, but in a production environment, it’s common to run with optimizations. This will **skip the assert** statement and go straight to the secure code

```
def check_permission(super_user):
        try:
                assert(super_user)
                print("\nYou are a super user\n")
        except AssertionError:
                print(f"\nNot a Super User!!!\n")
```

assert.py hosted with ❤ by GitHub view raw

If this code is executed like `python3 assert.py` , then it works perfectly as expected. But if it is used with the switch `-O` for optimization then a serious issue can arise. According to The Official Website: `-O` *Removes assert statements and any code conditional on the value of* `__debug__`

When you run this Python program in optimized mode, the *assert statement* is ignored and any normal user can be a **Super User.**

2: Denial of Service (DOS) Attack

DoS Attacks have seen a decline in the past few years due to load distribution and better system designs but Application-level DoS is still an exciting bug in the Bug-Bounty Field. Application-level DoS can Still be Achieved due to bad code writing or minor mistakes made by developers like improper error Handling.

In this small example, you can see a Simple C++ Program taking `Year of Birth` as input and checking if it is **>18** or not. At first, it all seems well and good, but in this code, no error handling is performed, it will directly result in the crashing of the program after the failure of the assert check.

This can be easily prevented by using error handling here, but it's a surprise that many developers take this lightly in Languages like C and C++.

```
int main() {
    using namespace std;
    int yob;
    cout << "Enter You Year Of Birth: ";
    cin>>yob;
    assert(2021-yob>18);
    cout<<"You May Proceed";

    return 0;
}
```

assert_dos.cpp hosted with ❤️ by GitHub view raw

3: A Twisted Denial of Service

Sometimes it's not your code that can cause these DoS to happen, it can be some library that is using another library for dependency which is causing the unexpected crashing of the program. You can visualize this as a Domino, if one assert() gets false it will make all the functions inside the code exit and thus crash the entire application.

A similar Scenario happened in **NodeJS,** it was marked as **CVE-2019–15604** In this Vulnerability, **an Attacker can Remotely trigger an assertion on a TLS server with a malformed certificate string causing the server to crash**

Initially, a Report was Submitted to NodeJS Team by **Rogier Schouten** on **Hackerone,** the Team Reviewed this as a **Critical Vulnerability** and the fix was released in a few days.

- **What was the Cause of CVE-2019–15604 ??**

This Vulnerability exploited the `getPeerCertificate()` function in the `socket` library of NodeJS. Anybody can remotely connect to a TLS server and supply a crafted x509 certificate, which causes an assertion and the process to abort.

```
const tls = require("tls");
const fs = require("fs");

let server = tls.createServer({
    ca: fs.readFileSync("./ca.crt"),
    cert: fs.readFileSync("./server.crt"),
    key: fs.readFileSync("./server.key"),
    requestCert: true,
    rejectUnauthorized: true
}, (socket) => {
    socket.setEncoding("utf8");
    socket.on("data", (data) => {
        console.log("server.socket.data", data);
        socket.write(data);
    });
    socket.on("end", () => undefined);
    socket.on("error", () => undefined);

    // THIS CRASHES THE SERVER
    console.log(socket.getPeerCertificate());
});
server.listen({ port: 12345 }, () => {
    console.log("listening!")
});
```

server.js hosted with ❤️ by GitHub view raw

Following Error was thrown when the crafted Certificate was sent to the TLS Server, it clearly shows an assertion being triggered causing the whole server to crash.

```
node[56864]: ../src/node_crypto.cc:1965:Local<v8::Object> node::crypto::X509ToObject(nc
 1: 0x10007c74a node::Abort() [/Users/findutnyy/.node/12.11.0/bin/node]
 2: 0x10007c4f6 node::AppendExceptionLine(node::Environment*, v8::Local<v8::Value>, v8:
 3: 0x100144db9 node::crypto::X509ToObject(node::Environment*, x509_st*) [/Users/findut
 4: 0x10013df92 node::crypto::SSLWrap<node::TLSWrap>::GetPeerCertificate(v8::FunctionCa
 5: 0x1001db0c8 v8::internal::FunctionCallbackArguments::Call(v8::internal::CallHandler
 6: 0x1001da689 v8::internal::MaybeHandle<v8::internal::Object> v8::internal::(anonymou
 7: 0x1001d9db2 v8::internal::Builtin_Impl_HandleApiCall(v8::internal::BuiltinArguments
 8: 0x100954359 Builtins_CEntry_Return1_DontSaveFPRegs_ArgvOnStack_BuiltinExit [/Users/
Abort trap: 6
```

error.cpp hosted with ❤️ by GitHub view raw

The fix explained the issue in more detail.

The key property of the malformed cert is that **subjectAltName** contains a string with a type `23` which cannot be encoded into a string by `X509V3_EXT_print`

```
//bad code

if (!SafeX509ExtPrint(bio.get(), ext)) {
      CHECK_EQ(1, X509V3_EXT_print(bio.get(),


//good code

if (!SafeX509ExtPrint(bio.get(), ext) &&
        X509V3_EXT_print(bio.get(), ext, 0, 0) != 1) {
      info->Set(context, keys[i], Null(env->isolate())).Check();
      USE(BIO_reset(bio.get()));
      continue;
```

bad_vs_good.js hosted with ❤️ by GitHub view raw

X509V3_EXT_print can return a value different from `1` if the **X509** extension does not support printing to a buffer. Instead of failing with an unrecoverable assertion, the code was replaced with the relevant value in the hashmap with a JS null value.

This Vulnerability shows how small negligence in error handling in a small library can become a Critical Point of Failure.

3: Remote Code Execution

PHP has always been a mine of crazy exploits and one of them is Remote Code Execution using the assert() function. In short words **assert()+PHP===Server Pwned.**

Unlike every other programming language PHP gives assert() a special power, **If the `assertion` is given as a string it will be evaluated as PHP code by assert()**

This means if you pass a `string` to assert() it will act as a `eval()` function and can run system commands.

- **Leaking Runtime Variable**

It is sometimes possible to leak a Runtime Variable from memory if that variable is being used to perform an assertion check.

```php
<?php
$flag="Take_the_Flag";
$flag2=NULL;
$t="die($flag)";
assert("$t == $flag2") or die("Dying...!!!!!");
?>
```

flag.php hosted with ❤ by GitHub view raw

Here I made a simple program to compare the input to `NULL` using an assertion check. Now if we pass a malformed input we can see the value of the `flag` variable in the error

Fatal error: Uncaught ParseError: syntax error, unexpected ';' in C:\Users\Mayank\Desktop\test.php(6) : assert code:1 Stack trace: #0 C:\Users\Mayank\Desktop\test.php(6): assert('die(Take_the_Fl...', 'assert("$t == $...') #1 {main} Next Error: Failure evaluating code: assert("$t == $flag2"):"die(Take_the_Flag) == " in C:\Users\Mayank\Desktop\test.php:6 Stack trace: #0 C:\Users\Mayank\Desktop\test.php(6): assert('die(Take_the_Fl...', 'assert("$t == $...') #1 {main} thrown in **C:\Users\Mayank\Desktop\test.php** on line **6**

This happens because the **`die`** function in PHP can run any system command before quitting the flow of the program. We abuse this feature and pass a string inside the assert function. Combining these two issues we can access any random variable from runtime or can **execute system commands.**

We can exploit any Application which might directly take input from the user without sanitizing it. For this, we can send payloads like `"die(system(ls));//"` it will run `ls` on the system and the rest of the code after the `system()` will be treated as comments due to the `;//` present in the payload.

The most important thing to focus on here is that we want to send a string and then run or commands using the `die()` function and then comment out the rest of the code. this can help us to exploit most of the use cases of the assert() function in PHP. We can think of it as **SQL Injection** where we run our Query and then comment out the rest of the query.

- **Bypass LFI checks and strpos() check**

Many Application uses strpos() to check for malicious inputs in the file parameter. **strpos()** finds the position of the first occurrence of a substring in a string, it returns

False if the given substring is not found in the given string.

> *assert(“strpos(‘$file’, ‘..’) === false”) or die(“Detected LFI attempt!”);*

This kind of filter can be found in many CTF’s or even in real life to protect the Application from LFI Attacks. This filter checks the `file` variable for any `..` pattern and if it's found then the program terminates.

We can use the same technique used above but here we have one extra function (strpos) that we need to bypass in order to run our command directly to the assert function.

`' and die(system(ls))or '` , this payload can be used to break out of strpos() and interact with the assert() function directly, and again we can run any arbitrary command on the Server.

Mitigations

- Avoid Using assert() function in the Production
- Take Special care on whether you are running an optimized code or not in python before using assert in the code.
- Assertions should be used as a debugging feature only.
- You may use them for sanity checks that test for conditions that should always be **`true`**
- Assertions should not be used for normal runtime operations like input parameter checks.
- As a rule of thumb, your code should always be able to work correctly if assertion checking is not activated.
- If using a user input directly in the assert function then try to sanitize it.

This will be it for this time and do share your points if you want me to add them here.

Thanks a lot for reading. Share if you like it 😇😇