Assessing a read-after free for security implications: The string comparison

devblogs.microsoft.com/oldnewthing/20220603-00

June 3, 2022

Raymond Chen

A security vulnerability report arrived that reported a read-after-free bug.

> Launch the program under Application Verifier with strict heap enforcement, and it crashes with a use-after-free. This can lead to arbitrary code execution or information disclosure.
>
> Bounty requested

Let's see what we have:

```
std::string GetThemeName();

void Vulnerable()
{
    auto theme = GetThemeName().c_str();
    if (strcmp(theme, "Light") == 0) {
        SetLightTheme();
    } else if (strcmp(theme, "Dark") == 0) {
        SetDarkTheme();
    } else {
        SetDefaultTheme();
    }
}
```

We indeed have a use-after free bug here, because we are using the `c_str()` after the temporary `std::string` has destructed. This leaves us with a pointer to already-freed data, and attempting to compare it to other strings will result in a read-after-free.[1]

The claim is that this is possible remote code execution or data disclosure. Does that make sense?

The value read from the already-freed memory is not used to control execution flow directly. It's not a vtable or a function pointer or index into a jump table, or anything like that, so there's no code injection opportunity here. The only code that can run is code that was written with the intent of being run. At best, you can cause the wrong code to run.

As for information disclosure, the value read from the already-freed memory is compared against two other values. The string in the already-freed memory is not directly disclosed. The only clue you get is that the theme changes if the already-freed memory contains either of the two magic strings. The information disclosure is therefore one bit of information: "Does the freed memory contain one of the magic strings?"

At best, the program gets the wrong answer to the question, and the user gets the wrong theme.

This code is running at program startup, where there are no competing threads, so the memory is not going to be overwritten during the small window between the free and the use. Therefore, in practice, the original string is still there, and the value will still be correct.

If you are very unlucky (or if bad luck is forced upon you by verification tools), the memory will be unmapped, and the read will crash with an access violation. In that case, you have a denial of service.

Okay, let's do our assessment.

Who is the attacker?

The finder never said, but maybe we can say that the attacker is somebody who emails you a file that this program uses, trying to induce the read-after-free.

Who is the victim?

Again, the finder never identified the victim, but the victim appears to be the recipient of the file who tries to open it. If the victim is unlucky, the theme is wrong. If the victim is very unlucky, the program crashes. And if the victim preconfigured their system to force extremely bad luck, the program always crashes at startup.

What has the attacker gained?

As far as I can tell, the attacker has gained nothing.

The victim might get the wrong theme, but they would have gotten the wrong theme even without any help from the attacker, seeing as the attacker has done nothing to make the error more likely than it would have been without any effort from the attacker.

Even if the victim experiences a denial of service due the program crashing at launch, this is the same crash that would have occurred without any action from the attacker. (And in the extreme case of having enabled memory verification tools, it is a *self-inflicted* denial of service.)

On top of all that, the results of the defect are never communicated back to the attacker, so the attacker gains no information.

What they found was a bug, but not a security vulnerability.

This is basically a script-kiddie vulnerability report. They used some tools, followed the instructions from a Web site on how to set up the tools, and with that configuration, they were able to identify a problem and submitted a security vulnerability report without bothering to evaluate what their crash actually told them.

Of the time I spend investigating security vulnerability reports, a good chunk of the time goes into assessing what the attacker actually gained. In many cases, the answer is “Nothing.”

And this is an assessment the finder really should have done before submitting the report. You can’t claim to have found a vulnerability without being able to articulate how it can be exploited.

[1] Let’s ignore the small string optimization. I’ve simplified the code for expository purposes.

Raymond Chen

Follow