Assessing the attack complexity of a race condition security vulnerability

devblogs.microsoft.com/oldnewthing/20241129-00

November 29, 2024

When assessing the attack complexity of a race condition security vulnerability, you have to look not only at how small the race window is but also how easy it is to hit the window.

Consider the following time-of-check-to-time-of-use (TOCTTOU) race condition. Suppose this code runs in kernel mode, and receives an `InfoStruct` from user mode that specifies where to put the information.

```
struct InfoStruct
{
    uint32_t size;
    char* buffer;
};

void GetInfo(InfoStruct* info)
{
    __try
    {
        // If the buffer does not point to user mode, then fail.
        if (!ValidateUserModeBuffer(info->buffer, info->size)) {
            return ERROR_INVALID_PARAMETER;
        }

        FillBufferWithData(info->buffer, info->size);
        return ERROR_SUCCESS;
    }
    __except (⟦ invalid user-mode pointer provided ⟧)
    {
        return ERROR_INVALID_PARAMETER;
    }
}
```

The race condition occurs if the user-mode buffer pointer changes after it is validated and before it is used.[1]

An attacker exploits this race condition by creating a second thread that modifies the `InfoStruct` after it has been validated but before it has been used.

On the other hand, consider this function, which returns two results in a user-provided buffer. The first result goes at the start of the buffer, and the second result goes after it. The size of the information is dynamic, so we tell the caller where we put the second result.

```
struct InfoStruct
{
    uint32_t size; // in
    char* buffer; // in
    char* result2; // out
};

void GetInfo(InfoStruct* info)
{
    __try
    {
        InfoStruct captured = *info;

        // If the buffer does not point to user mode, then fail.
        if (!ValidateUserModeBuffer(captured.buffer, captured.size)) {
            return ERROR_INVALID_PARAMETER;
        }

        // Copy the first result.
        uint32_t actual;
        if (!FillBufferWithResult1(captured.buffer, captured.size, &actual)) {
            return ERROR_INSUFFICIENT_BUFFER;
        }

        // Copy the second result.
        info->result2 = info->result1 + actual;
        uint32_t remaining = captured.size - actual;
        if (!FillBufferWithResult2(info->result2, remaining, &actual)) {
            return ERROR_INSUFFICIENT_BUFFER;
        }

        return ERROR_SUCCESS;
    }
    __except (⟦ invalid user-mode pointer provided ⟧)
    {
        return ERROR_INVALID_PARAMETER;
    }
}
```

This code is careful to capture the `InfoStruct` from user mode, but it incorrectly assumes that the value written to `info->result2` at the line `info->result2 = info->result1 + actual;` will be the same value read back when it is passed to `FillBufferWithResult2`.[2]

The `info` points to user mode, and an attacker exploits this race condition by creating a second thread that modifies the `InfoStruct`'s `result2` memory after its value has been set and before it is used.

Both of these race conditions involve a user-mode attacker modifying user-mode member during a small race window. Do they have the same attack complexity?

No. One of them is easy to exploit, and the other is hard to exploit.

Superficially, they look like they have similar attack profiles.

Timing of write	First vulnerability	Second vulnerability
Early	Rejected by validation: Fail	Overwritten before being used: Fail
On time	Exploit	Exploit
Late	No effect: Fail	No effect: Fail

However, this assessment assumes that the attacker is attempting only one malicious write. But what if they try multiple writes?

In the first vulnerability, an early malicious write poisons the scenario because `Validate-UserModeBuffer` reads a malicious value, which it rejects as invalid, so the attack fails. The exploitability of the first vulnerability is low because the first write must occur inside the narrow window.

In the second vulnerability, an early malicious write leaves the scenario still vulnerable. The kernel will just overwrite the malicious value at the line `info->result2 = info->result1 + actual;`, and the attacker gets another chance to overwrite the value before the kernel reads back from it on the next line.

So the attacker need only set up another thread that is in a tight loop:

```
while (true) {
    info->result2 = malicious_value;
}
```

and let that thread run while calling `GetInfo` from another thread. There will be a huge number of premature writes, which have no effect, and then the attack window opens, and you are already in a tight write loop, so the odds are not too bad that your malicious write will occur during the small race window.

By analogy, consider the case of trying to buy tickets to a popular event. You might try calling the ticket line repeatedly, hoping to call after the tickets go on sale but before they sell out. One possibly policy of the ticket agency is that if you call before the tickets go on sale, your phone number gets put on a list of “malicious callers”, and they won’t sell you tickets to that event. In that case, the cost of a premature call is very high, and your chances of success relies entirely on how well you can predict when the tickets go on sale. Another possible policy is that the ticket agency just says, “Sorry, they’re not on sale yet”, but you can still call again. In this case, the cost of a premature call is very low, and you may as well just put the ticket agency on speed dial and mash it as fast as you can.

[1] The correct behavior is to capture the values from user mode into kernel mode, and then operate exclusively on the captured values. The captured values cannot be manipulated from user mode, so the values you validate are also the values that you use.

[2] The correct behavior is not to read back the value written to user mode but rather remember the address in a place the attacker cannot modify, and use that remembered address.

```
char* result2 = info->result1 + actual;
info->result2 = result2;
if (!FillBufferWithResult2(result2, remaining, &actual)) {
```