NCC Group Whitepaper

Double Fetch Vulnerabilities in C and C++

Version 1.0

Author

Nick Dunn

Abstract

The double fetch vulnerability is a specific type of time-of-check to time-of-use (TOCTOU) bug, generally occurring in shared memory interfaces. It happens when a kernel process, or other privileged process such as a device driver, accesses a less trusted variable more than once, without re-verifying any checks of the variable on the second access.

The paper discusses how they occur, how they can be recognised, and some approaches to mitigate against them.

Table of Contents

Introduction

The double fetch vulnerability is a specific type of time-of-check to time-of-use (TOCTOU) bug, generally occurring in shared memory interfaces. It happens when a kernel process, or other privileged process such as a device driver or ARM TrustZone, reads a less trusted variable more than once, without re-verifying any checks of the variable on the second read. The term was first used by Fermin J. Serna in a post on the Microsoft Security and Defense Blog for MS08-061, in 2008[1] although the bug class was known prior to that blog post. This bug (MS08-061) allowed a local privilege escalation in the then current version of Windows[2] as a result of the kernel accessing a user-controlled memory pool twice, without checking for a change in the size of the memory pool. No public domain exploits exist for this vulnerability, although it was possibly overshadowed at the time by MS08-067, a notorious remote code execution vulnerability with System privileges, that was disclosed two weeks later. Since the discovery and description of MS08-061, variations of the double-fetch vulnerability have been discovered in the Linux Kernel, the Windows kernel, and in various device drivers and virtualized hardware (see the examples later in this paper). Double fetches cause security concerns for various reasons. If device drivers, kernel processes, or other high-privileged code interact with a user-controlled variable that is susceptible to double fetch, then it can result in a vulnerability (such as privilege escalation or buffer overflow). These defects are not easily identifiable by static analysis.

[1] https://msrc-blog.microsoft.com/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/

[2] https://docs.microsoft.com/en-us/security-updates/securitybulletins/2008/ms08-061

Double Fetch Bugs in More Detail

There are variations in the (unintentional) implementation or manifestation of the double fetch bug, but they all access a value from shared variable or memory interface that is accessible by another process. The subsequent fetching of that data without a subsequent check for whether it has been changed by other processes or by hardware that can modify the data. This can have security implications when a privileged process carries out the fetching, using user-controlled data from a less privileged process.

It's also important to note that although the bugs are commonly known by the name 'double fetch', the general class of bug can be caused by other types of access (such as a shared variable or object), and the situation may also occur with more than two accesses.

Double fetch bugs can be caused by access of shared memory with insufficient protections, sometimes due to invalid programmer assumptions, or in other cases, by compiler optimization, which we'll also examine. The outcome is a privileged process such as a kernel, uses data controlled by a less privileged user without a suitable check, potentially leading to privilege escalation.

Not all double fetch instances are exploitable. The level of risk is classified by Wang[3] as follows: * Benign double fetch: A case that will not cause harm, because of other protections, because the double-fetched value is not used twice or the original value is overwritten prior to use. * Harmful double fetch: A case that could cause failures in the kernel/process with the failures limited to performance issues. * Double-fetch vulnerability: A case that can be actively exploited for privilege escalation, arbitrary read/write, or buffer overflows.

While we examine double fetch in general terms in this paper, we concentrate on the security implications and primarily concern ourselves with double fetch vulnerabilities (although any defences and mitigations discussed are equally applicable against the benign and harmful instances).

The non-exploitable instances can still result in sub-optimal performance, and can cause poor performance in terms of speed or resource usage, or potentially a crash.

These bugs exist in low-level code such as the kernel and device drivers, because shared memory interfaces can offer performance benefits over high-level mechanisms such as sockets. Device drivers, by their nature, are susceptible to this issue because they run with elevated privileges while interacting with memory that may be updated by a piece of hardware.

Double fetch vulnerabilities can be caused by poor programming practices or by unexpected compiler optimizations. To illustrate these cases and make the explanations more concrete, we'll look at examples of each in detail in the following sections, and discuss the nature of the vulnerabilities and how they were fixed.

Programming Practices Causing Double Fetch

A double fetch bug may occur due to insecure programming practices. This could happen when, for example, a privileged process uses a variable controlled by a less-privileged process and performs a check on it, before subsequently reusing the same variable without performing further checks.

Although double fetch vulnerabilities existed prior to MS08-061, these were usually described using more traditional terms, such as buffer overflow. Fermin J. Serna first used the term double fetch in a blog post.[4]

In the MS08-061 blog post we are given the following code:

```
// Attacker controls lParam
void win32k_entry_point(...) {
    [...]
    // lParam has already passed successfully the ProbeForRead
```

[3] https://discovery.ucl.ac.uk/id/eprint/1557280/1/Dodier-Lazaro_sec17-wang.pdf
[4] https://msrc-blog.microsoft.com/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/

```
    my_struct = (PMY_STRUCT)lParam;
    if (my_struct ->lpData) {
        cbCapture = sizeof(MY_STRUCT) + my_struct->cbData;  // [1] first fetch
        [...]
        // my_struct ->lpData has already passed successfully the ProbeForRead
        [...]
        if ( my_allocation = UserAllocPoolWithQuota(cbCapture, TAG_SMS_CAPTURE)) != NULL) {
            RtlCopyMemory(my_allocation, my_struct->lpData,
                my_struct->cbData);   // [2] second fetch
        }
    }
    [...]
}
```

In this case, the two fetches are visible in the source code, and a manual code review could conceivably identify the issue, provided the reviewer identified the variable as originating in user space, prior to being used by the kernel. In this example, the first fetch of mystruct->cbData is used to determine the size of my_struct->lpData, and to then determine the size of an area of memory referenced by my_allocation that will hold a copy of my_struct->lpData. The second fetch takes place when the RtlCopyMemory function[5] is used to obtain a new copy of my_struct and copy mystruct->cbData bytes of my_struct->lpData into my_allocation. If an attacker has inserted a payload into the memory referenced by my_struct that is larger than the original value of my_struct, then this payload will overflow the memory referenced by my_allocation.

Generally, these situations involve invariants between two or more variables where one or more of these variables is modified without the invariant being enforced. An example of a specific situation that could impact security, such as the one previously described, is where code for a privileged process reads a user-controlled variable to determine the size of memory allocation and subsequently to perform a copy of the variable, without re-checking that the size is still correct. Detection and possible fixes are described later in this post. The following example shows code from Linux kernel 2.6.9, reported in CVE 2005-2490[6,7]. Although described as a buffer overflow in the CVE, it is similar to the vulnerability described by MS08-061 in that it checks the length once, but fetches it twice, using the length from the first fetch. In the following code, the user-controlled parameters passed into the function are examined in the first while loop and copied in the second while loop. The verification of the data's length only occurs in the first loop, while the second loop copies the data without verifying the length.

```
int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg, unsigned char *stackbuf,
    int stackbuf_size)
{
    struct compat_cmsghdr __user *ucmsg;
    struct cmsghdr *kcmsg, *kcmsg_base;
    compat_size_t ucmlen;

    [...]

    kcmsg_base = kcmsg = (struct cmsghdr *)stackbuf;
    ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);

    while(ucmsg != NULL) {
        if(get_user(ucmlen, &ucmsg->cmsg_len))
            return -EFAULT;
```

[5] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-rtlcopymemory
[6] https://www.cvedetails.com/cve/CVE-2005-2490/
[7] https://discovery.ucl.ac.uk/id/eprint/1557280/1/Dodier-Lazaro_sec17-wang.pdf

```
        if(CMSG_COMPAT_ALIGN(ucmlen) < CMSG_COMPAT_ALIGN(sizeof(struct compat_cmsghdr)))
            return -EINVAL;

        [...]

        if((...)(((char __user *)ucmsg - (char __user*)... + ucmlen) > kmsg->msg_controllen)
            return -EINVAL;

        ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
    }

    if(kcmlen == 0)
        return -EINVAL;

    [...]

    ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);
    while(ucmsg != NULL) {
        __get_user(ucmlen, &ucmsg->cmsg_len);
        tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
        CMSG_ALIGN(sizeof(struct cmsghdr)));
        kcmsg->cmsg_len = tmp;

        if(copy_from_user(CMSG_DATA(kcmsg), CMSG_COMPAT_DATA(ucmsg),
            (ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))))
    }
```

Compiler-Introduced Double Fetch

A double fetch bug can occur in situations where the compiler introduces multiple fetches of a variable, despite a single read within the source code. Unlike the MS08-061 bug previously discussed, this defect arises from a permissible compiler transformation that results in a second read of the user-controlled memory. Consequently, the defect is visible in the object code but not the source code.

Note that this bug type has been referred to as 'compiler induced' in the past. We are using the phrase 'compiler introduced' here to emphasize that this is a valid transformation. This distinction will be discussed in more detail later.

The following code, adapted from Xen Security Advisory CVE-2015-8550 (XSA-155),[8] illustrates a compiler-introduced double fetch. The code is vulnerable to a race condition, in a case where the integer referenced by the ps pointer could be modified by another thread that carried out the modification between the first and second read of the variable.

```
#include <stdio.h>
#include <stdlib.h>

void doStuff(int* ps)
{
        printf("NON-VOLATILE");
        switch(*ps)
        {
                case 0: { printf("0"); break; }
                case 1: { printf("1"); break; }
```

[8] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8550

```
                case 2: { printf("2"); break; }
                case 3: { printf("3"); break; }
                case 4: { printf("4"); break; }
                default: { printf("default"); break; }
        }
        return;
}

void main(int argc, void *argv)
{
        int i = rand();
        doStuff(&i);
}
```

The source code has a single read of *ps, but the compiler produces object code with multiple reads of *ps. This is a result of the "as-if" principle, as described in section 5.1 of the C99 Rationale 2003[9]:

The /as if/ principle is invoked repeatedly in this Rationale. The C89 Committee found that describing various aspects of the C language, library, and environment in terms of concrete models best serves discussion and presentation. Every attempt has been made to craft these models so that implementations are constrained only insofar as they must bring about the same result, /as if/ they had implemented the presentation model; often enough the clearest model would make for the worst implementation.

This bug is a consequence of the way GCC optimizes code for Intel architecture, containing switch statements with five or more cases.[10] The optimization eliminates the use of a register in cases where the default value is hit, but opens a small window for an attacker to modify the variable between reads.

This is apparent in the compiler output below, that shows the switch value held in memory, at a location pointed to by the rbx register, and being dereferenced in the cmp instruction, before being used again by subsequent instructions:

```
doStuff:
.LFB39:
    .cfi_startproc
    endbr64
    push    rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    xor    eax, eax
    mov    rbx, rdi
    lea    rsi, .LC0[rip]
    mov    edi, 1
    call    __printf_chk@PLT
    cmp    DWORD PTR [rbx], 4
    ja    .L2
    mov    eax, DWORD PTR [rbx]
    lea    rdx, .L4[rip]
    movsx    rax, DWORD PTR [rdx+rax*4]
    add    rax, rdx
    notrack jmp    rax
    .section    .rodata
    .align 4
```

[9] http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf

[10] http://tkeetch.co.uk/blog/?p=58

```
    .align 4

[...]

.L2:
    .cfi_restore_state
    lea    rsi, .LC1[rip]
    mov    edi, 1
    xor    eax, eax
    pop    rbx
    .cfi_def_cfa_offset 8
    jmp    __printf_chk@PLT
    .cfi_endproc
```

This is not a compiler bug as this is a valid transformation. The Xen source code is defective in that it fails to disallow this optimization through the use of volatile or other means, as discussed further below.

The bug specifically occurs in an optimized GCC executable. Compiler-introduced vulnerabilities depend on the specific implementation (compiler and flags)[11],.[12] The source code should always be considered incorrect if the transformation that introduces the defect is permissible.

[11] http://tkeetch.co.uk/blog/?p=58

[12] https://docs.huihoo.com/blackhat/usa-2013/us-13-Blanchou-Shattering-Illusions-in-Lock-Free-Worlds.pdf

Static analysis is currently of limited use in detecting double fetch bugs because of the contextual knowledge required outside of the identification of unchecked second fetch. To do this any static analysis tool would require additional information to identify an instance:

- Whether the code being examined runs with elevated privileges
- Whether the variable can be modified outside of the context of the code
- Whether the above modification may be made by a non-administrator user/process

Manual code review could be used to identify double fetch bugs, although this would depend upon the skills and alertness of the reviewer, as well as the reviewer being provided with suitable information or documentation to allow them to draw correct conclusions.

Wang and Krinke give the following breakdown of how their approach to static analysis was used to detect double fetch bugs in the Linux kernel. This seems to have been effective, according to their paper, although it is a refined version of the above general points, adapted for the Linux kernel[13]:

"Our approach examines all source code files of the Linux kernel and checks whether a kernel function contains two or more invocations of transfer functions that fetch data from the same user pointer. From the 39,906 Linux source files, 17,532 files belong to drivers (44%), and 10,398 files belong to non-x86 hardware architectures (26%) which cannot be analyzed with Jurczyk and Coldwind's x86-based technique. We manually analyzed the matched kernel functions to infer knowledge on the characteristics of double fetches, i.e., how the user data is transferred to and used in the kernel, which helped us to carry out a categorization of double-fetch situations, as we discuss in Section 3.2. The manual analysis also helped us refine our pattern matching approach and more precisely detect actual double-fetch bugs⋯"

Dynamic analysis can be used to identify both compiler-introduced double fetches and double fetches resulting from insecure coding practices, with useful work being done by both the Bochspwn project[14] and by Wang.

The Bochspwn project's whitepaper provides a useful pattern for the detection of double fetch bugs[15]: Defining the double-fetch pattern as two consequent reads of the same memory location is insufficient to achieve satisfying results; instead, the pattern must be defined in greater detail. To have a double-fetch vulnerability:

- There must be at least two memory reads from the same location.
- Both read operations take place within a small race window.
- The code instantiating the reads must execute in kernel mode.
- The location subject to multiple reads must reside in memory writable by user-mode code.

Using dynamic analysis to detect double fetch bugs is non-trivial, with considerable time and resource needed to run the test platform, and further time needed to interpret the results[16]:

"Jurczyk et al. presented a dynamic approach for finding double fetches. They used a full CPU emulator to run Windows and log all memory accesses. This requires significant computation and storage resources, as just booting Windows already consumes 15 hours of time, resulting in a log file of more than 100 GB [38]. In the memory access log, they searched for a distinctive double-fetch pattern, e.g., two reads of the same user-space address within a short time frame. They identified 89 double fetches in Windows 7 and Windows 8. However, their work also required manual analysis, in which they found that only 2 out of around 100 unique double fetches were exploitable double-fetch bugs. "

[13] https://discovery.ucl.ac.uk/id/eprint/1557280/1/Dodier-Lazaro_sec17-wang.pdf
[14] https://storage.googleapis.com/pub-tools-public-publication-data/pdf/42189.pdf
[15] https://storage.googleapis.com/pub-tools-public-publication-data/pdf/42189.pdf
[16] http://www.misc0110.net/files/double_fetch.pdf

Impact of Double Fetch on Application Security

A privilege escalation risk is present when memory accessed by a privileged process being shared with, and modified by, a less privileged process. Exploitable instances can result in arbitrary read/write or in code execution in the context of the privileged process. Less harmful instances of the bug can result in crashes or have a performance impact.

Possible Solutions and Mitigations

Since double fetch bugs can have varying causes, we must consider different solutions for the two different subtypes of double fetch. This section suggests both how one can prevent double fetch vulnerabilities introduced through insecure programming practices, as well as how one can prevent double fetch vulnerabilities introduced by compilers.

Programming Practices Double fetch bugs accessing shared memory may be fixed by adding a check against the second fetch, eliminating the second fetch (where practical), or performing the check in a different manner. For instance, shared memory variables can be copied to an automatic variable and/or declared volatile.

The blog post about MS08-061, previously used to illustrate this bug type, gives the following non-vulnerable code as a fix:

```
// Attacker controls lParam
void win32k_entry_point(…) {
    [...]
    // lParam has already passed successfully the ProbeForRead
    my_struct = (PMY_STRUCT)lParam;
    cbData_captured= my_struct->cbData;
    lpData_captured = my_struct->lpData;
    if (lpData_captured) {
        cbCapture = sizeof(MY_STRUCT) + cbData_captured;
        [...]
        // lpData_captured has already passed successfully the ProbeForRead
        [...]
        if (my_allocation = UserAllocPoolWithQuota(cbCapture, TAG_SMS_CAPTURE)) != NULL) {
            RtlCopyMemory(my_allocation, lpData_captured, cbData_captured);
        }
    }
    [...]
}
```

The fixed code now uses both the size and the data from the original fetch instead of repeating the fetch, and has two new variables, lpData_captured and cbData_captured, holding the data and its length respectively. The memory referenced by my_allocation is now the correct size to hold the data from the user-controlled variable, and the privilege escalation attack is eliminated.

Compiler Introduced For compiler-introduced double fetches, the use of volatile variables is one possible solution to the double fetch problem. The volatile keyword prevents a double fetch as each read or write of a volatile-qualified object that appears in the source code must have an equivalent fetch or store in the object code, and additional reads and writes cannot be injected. For non volatile-qualified objects, a compiler may omit an access if the compiler deems it unnecessary (bearing in mind that the compiler does not detect that the variable may have been modified by external processes) or may introduce additional accesses for any reason (such as the optimizations discussed earlier).

A fix that uses volatile for the CVE-2015-8550 bug (used to illustrate the above bug type) is shown below.[17] For this issue, the use of a volatile int pointer eliminate the compiler-introduced double fetch from the object code:

```
#include <stdio.h>
#include <stdlib.h>

void doStuff(volatile int* ps)    // Use volatile to ensure a single read
{
    printf("VOLATILE");
    switch(*ps)
```

[17] http://tkeetch.co.uk/blog/?p=58

```
    {
        case 0: { printf("0"); break; }
        case 1: { printf("1"); break; }
        case 2: { printf("2"); break; }
        case 3: { printf("3"); break; }
        case 4: { printf("4"); break; }
        default: { printf("default"); break; }
    }
    return;
}

void main(int argc, void *argv)
{
    int i = rand();
    doStuff(&i);
}
```

The code above results in the following compiler output, now having a single dereference of the memory location that is stored in rbx, and the value held in that location being stored in the eax register:

```
doStuff:
.LFB39:
    .cfi_startproc
    endbr64
    push    rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    lea    rsi, .LC0[rip]
    mov    rbx, rdi
    xor    eax, eax
    mov    edi, 1
    call     __printf_chk@PLT
    mov    eax, DWORD PTR [rbx]
    cmp    eax, 4
    ja    .L2
    lea    rdx, .L4[rip]
    movsx    rax, DWORD PTR [rdx+rax*4]
    add    rax, rdx
    notrack jmp    rax
    .section    .rodata
    .align 4
    .align 4
```

In more formal terms, the volatile keyword imposes restrictions on access and caching.

According to C23:

"Accesses to objects through the use of lvalues of volatile-qualified types are evaluated strictly according to the rules of the abstract machine. At each sequence point, the value last stored in the object must agree with that prescribed by the abstract machine. In the absence of the volatile qualifier, the contents of the designated location may be assumed to be unchanged, except for possible aliasing."

As the bug is inadvertently introduced by compiler optimization, it might be tempting to take an approach of attempting to eliminate it by switching off compiler optimization. This approach is definitely not recommended. In addition to the performance impact, a solution that relies on the person compiling the code using the appropriate settings is more of a workaround than a repair. It also carries the inherent risk of not being applied if procedures or instructions are not followed correctly.

Performance Considerations when Using Volatile-Qualified Objects As mentioned above, the decision to use shared memory is usually for the purposes of improved performance. Using volatile -qualified objects can inhibit some optimizations, impairing performance. Volatile should not be used carelessly, and should only be applied as necessary to ensure the correctness of the code.

If volatile (or any other solution) is used to repair compiler-inserted double fetch bugs, the object code should be examined to ensure a that only a single read instruction is emitted and test the resulting executable to ensure that the repair is successful.

The use of volatile is unlikely to repair vulnerabilities introduced through insecure coding practices. The use of volatile ensures that each memory access in the source code occurs in the executable, even when such memory accesses are erroneous.

Compilers sometimes have bugs that cause the volatile qualifier to not behave as required by the standard.[18] When using volatile , examine the generated object code to verify that any use of volatile to eliminate double fetch translates to the appropriate object code, and make sure you test the optimized, production builds.

[18] https://www.cs.utah.edu/~regehr/papers/emsoft08-preprint.pdf

Conclusion

Double fetch bugs can result in privilege escalation vulnerabilities that can allow an attacker with a low privilege account to execute code with elevated privileges, although the exploitable vulnerabilities are a relatively small subset of these bugs. They can result from either insecure programming practices or can be caused as an unintended side effect of compiler optimization (which are also combined with slightly less obvious insecure coding practices). There is currently only one documented case where the GCC compiler can introduce such a defect, but to future-proof your code you should assume that any allowable transformation is possible.

The two types of double fetch bug both have the same result, whereby an invariant exists involving two or more variables and one or more of these variables is modified without the invariant being enforced. For example, a privileged process reads a user-controlled variable, allocates memory to be used to hold a copy of the variable and subsequently reads the variable from userland again without carrying out a subsequent check, allowing a potential overflow of the kernel-controlled variable.

The bugs resulting from insecure coding vary, depending on the exact nature of the bug but include code changes to ensure a check of the variable length after the second read, or performing a single read, storing a local copy and eliminating the second read. The compiler introduced bugs can be fixed with the use of the volatile qualifier to prevent the unwanted second read appearing in the object code.

The bugs generally exist due to the use of shared memory communication for the purpose of higher performance, while the fixes generally have some small impact on performance. The fixes generally have a minimal performance impact, and it seems unlikely that there would be any situation where such a small performance degradation would outweigh the advantages of a secure and robust implementation.

Standing on the Shoulders of Giants

This blog post has used information from several sources, and I'm indebted to the excellent work of the following people and organizations:

- Robert C. Seacord for helpful suggestions and for the QA of initial drafts of this paper.
- Software Engineering Institute, Carnegie Mellon University:

https://wiki.sei.cmu.edu/confluence/display/c/CON43-C.+Do+not+allow+data+races+in+multithreaded+code

- Tom Keetch [@tkeetch]:

http://tkeetch.co.uk/blog/?p=58

- Pengfei Wang, National University of Defense Technology; Jens Krinke, University College London; Kai Lu and Gen Li, National University of Defense Technology; Steve Dodier-Lazaro, University College London:

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei