TikTok: Kernel TOCTTOU Protection

Uroš Tešić
EPFL

Mathias Payer
EPFL

Abstract

Your abstract text goes here. Just a few facts. Whet our appetites. Not more than 200 words, if possible, and preferably closer to 150.

1 Introduction

System call wrappers enable administrators to define system call execution policies. Such policies could prevent the execution of a system call based on the ID of the call and its arguments. By setting only necessary access policies for all processes, administrators would reduce the damage in case of an attack. Filtering could also restrict calls to the exploitable system calls. By excluding some combinations of arguments, the administrator could mitigate certain vulnerabilities until a patch is available. Many embedded devices have binary-only drivers that prevent them from updating the kernel. Malicious input filtering could be used as a permanent solution in those cases.

Unfortunately, system call wrappers suffer from a design flaw. Filters execute before system calls. After the filter reads arguments and validates them, the system call reads them the second time. In-between these two reads, the attacker could change values, leading to an execution of a forbidden call. This is called a *time-of-check to time-of-use* attack (TOCTTOU). It is a consequence of a *double fetch* from the userland and is notoriously hard to detect. Unlike other bugs, double-fetches are benign until the adversary decides to change the arguments.

The contributions of this paper are:

- TikTok - mitigation for time-of-check to time-of-use attacks on system call arguments in the Linux kernel
- A technique to postpone the writes from userland

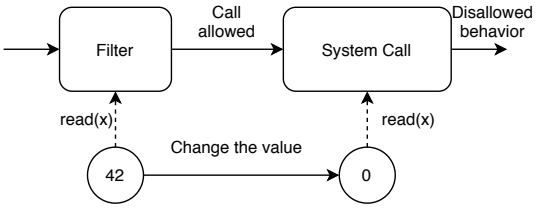


Figure 1: Bypassing a system call filter using a TOCTTOU attack

- A technique to defer the writes from the kernel, while continuing normal kernel execution
- Performance analysis of our system and its security guarantees

The rest of the paper is organized as follows: Section 2 explains inter-process communication, paging, double fetch bugs, and the related background. Section 3 describes how *TikTok* works in theory, while section 4 elaborates on the x86-64 implementation. Related work is discussed in section 5.

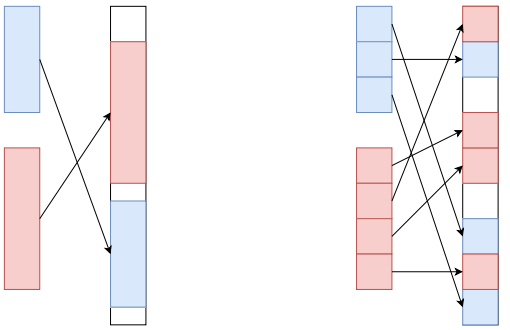
2 Background

2.1 Interprocess Communication

The two main types of communication between processes are *shared memory* and *message passing* [11].

Shared memory relies on processes having a section of memory that both can access. Data transfer is fast. However, synchronization is problematic. Processes must monitor shared memory for changes, leading to unnecessary polling.

Message passing consists of one process calling send, and another one calling receive to fetch the message. Synchronization is guaranteed, with parties waiting for their calls to be served. However, unnecessary message copying



(a) Segmented (b) Paged

Figure 2: Two virtual memory spaces mapped to physical memory using segmented and paged approaches

can occur between processes on the same system.

Modern operating systems support both of these approaches. The downsides are usually offset by adding the bare minimum of the other approach (e.g. shared memory with semaphores, or message passing with shared buffers). Two processes usually use only one paradigm to communicate. Sending messages and writing to the shared memory at the same time is exceptionally rare.

2.2 Virtual Memory and Page Tables

Operating system (OS) provides an illusion that every process is executing alone on the processor. To accomplish this, the OS needs to restore the program state on the context switch between two processes (e.g. CPU registers) and to prevent processes from accessing each other's memory. Memory is protected by *virtualization*. Processes use *virtual addresses* that get mapped to the *physical addresses*. When the OS moves data to a different physical address, the virtual address referring to the data remains the same.

Virtual memory can be implemented by storing different processes' data at different offsets in physical memory and limiting the access to corresponding memory chunks. Each process's memory chunk is called a segment and the implementation is called *segmented virtual memory*. The translation is accomplished by adding an offset to the virtual address. Considering that segments need to be continuous, the free physical memory can be fragmented such that the OS cannot find a part large enough to store a new process.

Paged virtual memory is more flexible. The physical memory is partitioned into fixed-size pages (usually 4 kB). A page in the virtual memory space gets mapped to the corresponding page in the physical memory. The mapping function is defined for each process by a *page-table*.

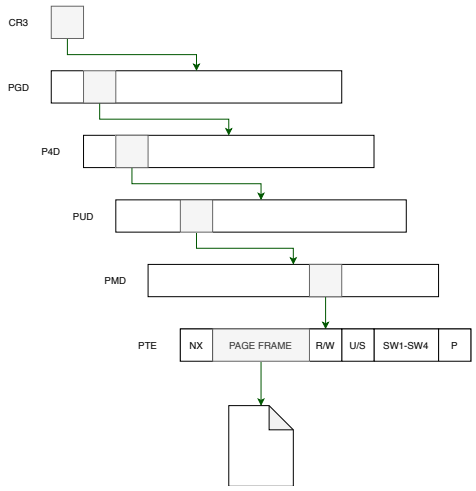


Figure 3: Page Table Structure on x86. Only relevant data has been included.

Page-tables store the reference to the mapped physical memory (*page frame*). They also keep the access permissions for each page (*read*, *write*, *execute*, *user*, *superuser*). In case of low memory, rarely accessed pages can be moved to the disk, and replaced by immediately needed pages. This process is called *swapping*. Similarly, when processing a file, pages do not need to be loaded immediately, but on the first access (*on-demand paging*). The *present* bit is added to differentiate between present and absent pages.

Virtual memory spaces are quite large (2^{64} bytes on a 64-bit processor). A table containing all one-to-one mappings would be impossibly large to store. Page tables are therefore stored as trees with only the allocated memory being present (fig. 3). However, instead of just reading the corresponding physical address on memory access, the processor now needs to perform a tree traversal. Different bits in a virtual address encode the path the processor needs to take to obtain the page frame number (e.g. the first 8 bits tell CPU which entry on the first level it needs to dereference). Traversal needs to be fast. It is implemented in hardware by the *memory management unit* (MMU). Reading the page table from memory is slow, so a small cache is added to the MMU to store frequently accessed page entries - *translation-lookaside buffer* (TLB). On modern processors, TLB consists of several levels, and can even be backed by another MMU cache.

2.3 x86-64 Page Tables

x86-64 architecture officially supports paged virtual memory model with a 5 level page table (fig. 3):

PGD Page Global Directory

P4D Page Fourth-level Directory

PUD Page Upper Directory

PMD Page Middle Directory

PTE Page Table Entry

Every level corresponds to 8 bits in the virtual address, with the remaining 12 bits identifying the offset in the actual page frame. A page table entry includes the following information:

Present bit (P) is set if the page is present in memory

Read/Write bit (R/W) denotes if the page is writable or just readable

User/Superuser bit (U/S) represents if the page can be accessed by the user, or only by the superuser

Not Executable bit (NX) is set if the code stored on the page cannot be executed

Page Frame Number denotes the page frame the entry points to

SW1-SW4 Four bits free for the OS to use

2.4 Page-Faults

On invalid access (e.g. wrong permissions, page not present) the MMU will trigger a page-fault. The fault is a synchronous interrupt that executes in the context of the faulting (accessing) thread. The page-fault handler loads an absent page from the disk. In the case of a write to a temporarily shared page, it creates an independent copy of the page (*copy-on-write*). On permission iolation, the page-fault handler kills the thread. After the fault finishes executing, the faulting instruction is re-executed.

With the advent of cloud computing, userland page-fault handling has been added to the Linux kernel. Users can define their routines to load swapped-out data. This is particularly useful on computer farms when migrating virtual machines between physical nodes. One only needs to migrate the code that is executing. Accesses to the unmigrated memory will be passed to the userland page-fault handler. It will then fetch them over the network and continue the execution.

2.5 Copy-from-User and Copy-to-User

Linux uses swapping only for user memory. When executing in the kernel context, kernel memory is mapped and present. A page fault on kernel memory access is therefore considered fatal. However, the kernel needs to access potentially paged-out user memory. The user memory is also limited to the lower half of the virtual memory space. On every access to the user-pointer, the kernel needs to verify this constraint.

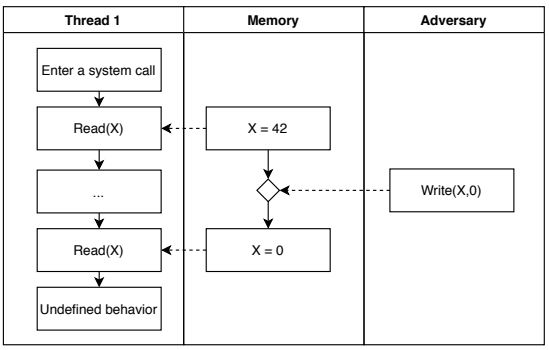


Figure 4: Diagram of a double fetch bug

Linux provides functions and macros for user-memory access from the kernel to enforce the checks. A page-fault generated by them is treated as a fault in the user process which had invoked the executing system call.

The interface for communication with userspace:

`copy_(from/to)_user`

`__copy_(from/to)_user`

`(get/put)_user`

`__(get/put)_user`

`user_strcpy`

`user_strlen`

BSD also provides a similar interface using `copy_in` and `copy_out` functions.

2.6 Double Fetch Bugs

Double fetch bugs occur when a privileged environment (such as the kernel) reads untrusted memory two or more times (fig. 4). In between those two reads, memory could have been changed by an unprivileged adversary. Considering that this bug relies on carefully timed accesses for two different threads, it is a variant of a race condition. The situation where the first fetch validates the value of the fetched variable, but the computation is only performed on the second fetch, is called a *time-of-check to time-of-use* (TOCTTOU) bug. TOCTTOU bugs have been widely studied in file systems, where the API makes it possible to swap the file after validating the access rights [7, 8, 14, 18].

Wang et al. explain in [16] that double fetches appear not only in kernels, but wherever there is a trust boundary to cross (e.g. kernel – hypervisor, hardware – kernel). Double fetches have been responsible for many vulnerabilities in the kernel.

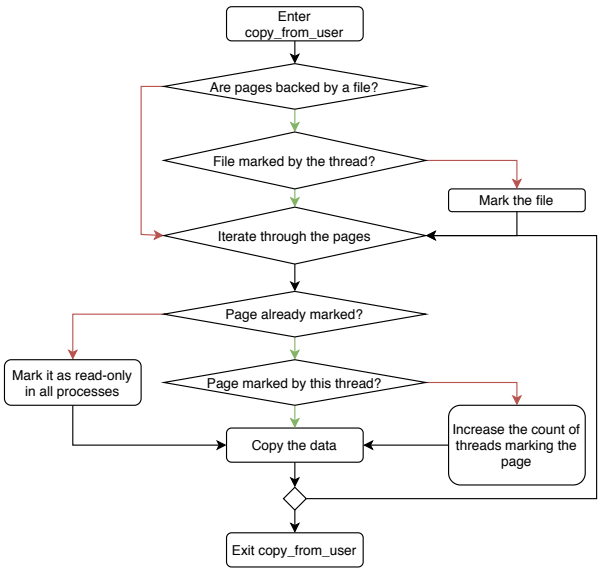


Figure 5: `copy_from_user` marks both the file and the pages before reading in the data

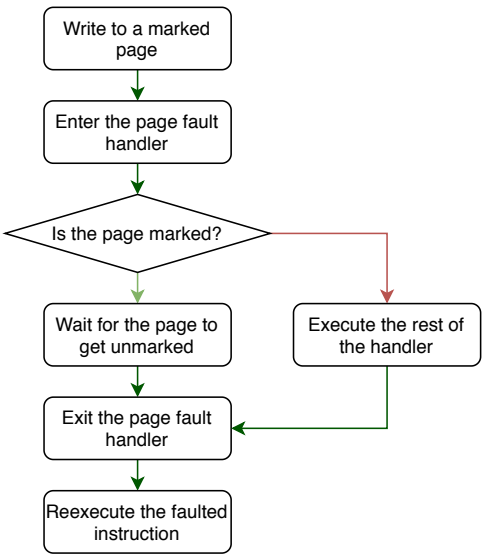


Figure 6: TikTok's handling of the writes to a marked page

3 Design

In this section we describe a high-level overview of TikTok. We start with the security model in section 3.1. Afterward we describe how TikTok protects system call arguments from userland (section 3.2) and the kernel (section 3.3) writes.

3.1 The Security Model

The administrator has set up DAC preventing the adversary from accessing files mapping to physical devices. They also control which file-systems are mounted on the machine and have disabled userland page-fault handling.

The adversary has access to a user account on a target machine. They can execute arbitrary code (including system call) and want to obtain root access by triggering a TOCTTOU bug in the kernel.

3.2 Protecting System Call Arguments from Writes by the User

System calls access the user memory via `copy_from_user` and its variants. When that happens, TikTok *marks* the entire page storing the argument as *read-only* in all virtual memory spaces mapping it (fig. 5). Multiple system calls can mark a page at the same time. Marking a page does not affect readsfrom userspace in any way. When all the system calls that use the page finish executing, the page is *unmarked* – the previous permissions are restored.

Writes to the marked pages will trigger a page-fault (fig. 6). In the page-fault handler we intercept these writes and make them wait for the page to get unmarked. The faulting thread attempts to perform the write again.

3.3 Protecting System Call Aguments from Writes by the Kernel

Watson mentions in [17] that the system call wrappers he analyzed do not handle both reads and writes in the same call properly. Unlike those solutions, TikTok does not copy arguments to separate memory, leading to complex redirection of userspace pointers. TikTok defers the writes until it is safe to commit the writes.

However, deferring kernel writes the same way as user writes would lead to deadlocks. System call `rt_sigaction` needs to write to a page it previously marked. Pausing execution would leave the thread in a state where it is waiting for itself to exit unmark the page. Temporarily unmarking the page would enable an adversary to edit arbitrary data on it.

Allowing the writes for the kernel is not an option. The adversary could abuse this to change the marked memory. They would execute a read system call into the marked page. System calls execute in the kernel context and would be able to bypass protection. The read system call would then write arbitrary data into the protected area.

Our solution is based on the fact that we already provide partial checkpointing of the system call's view of RAM. During its execution, the system call can only see the state of the memory as it was at the beginning of the call. All writes from userspace become visible only when the call has

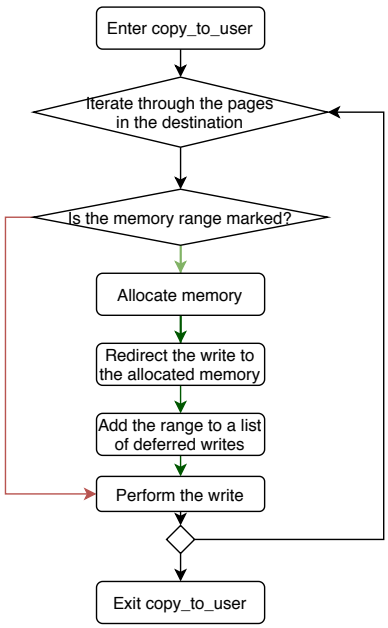


Figure 7: `copy_to_user` defers the writes from kernel until the end of the system call

finished execution. TikTok extends this policy to writes from the kernel. Considering that we need to continue execution after a write to a marked page, we buffer all the writes until the system call finish. At that point we unmark the pages and allow them to proceed normally.

3.4 Ignored System Calls

Some system calls (e.g. `pollfd` and `futex`) rely on writes from userspace for some of their functionality. Marking their arguments would lead to deadlocks, so they are ignored. Considering that attacking these system calls would to expected behavior, the authors do not consider this a deficiency.

Other system calls can be ignored as an optimization. Any unformatted data that is passed to the kernel does not need to be marked by TikTok. Overwriting this data is equivalent to passing different data to the call. Considering that the write call takes unformatted data as one of its arguments, this frequently called interface does not need to be protected.

3.5 Two Axes of the Linux Memory

Memory in Linux can be *file-backed* and *anonymous*. File-backed pages have map to a corresponding file. Anonymous pages do not have a backing file (e.g. stack and heap).

Another classification is based on privacy: *private* and *shared*. Private memory is part of only one virtual memory space. This memory space can be accessed by multiple threads in a process, but no threads outside the process have access. Shared memory can be accessed by multiple processes.

	Anonymous	File-backed
Private	/	Copy-on-write
Shared	Inherited by fork	Can be (un)mapped at any time

Table 1: Properties of the different types of memory in Linux

Unlike private memory and shared anonymous memory, shared file-backed memory can be mapped and unmapped at will (table 1). It also preserves its state. This enables an adversary to map a marked page as writable and edit it. TikTok intercepts mapping of memory and checks the page frames being mapped. The pages are then mapped with appropriate permissions into the virtual memory space.

Devices are treated as files in Linux and can be memory mapped. However, hardware may change its registers at will. There are no conceivable ways from protecting from TOCTTOU attacks if the adversary stores his arguments in device mapped memory. Considering that mapping device memory to userland is considered bad practice, we rely on *Descretionary Access Control* (**DAC**) to prevent users from mapping devices in the first place.

3.6 File Writes

Files in Linux can be accessed in two ways:

- by mapping the file to memory
- by using system calls to modify the file (e.g. `write`)

Watson has noticed that protected file-backed pages could still be edited by a `write` call. TikTok prevents this attacks by pausing the write to the corresponding file as long as it has any marked mapped pages.

3.7 TikTok Deadlocks

TikTok adds additional synchronization points to multi-threaded programs. It is possible for these points to introduce previously non-existant deadlocks to programs. However, deadlocking threads would need to communicate using both shared memory (for TikTok to stop one of them) and message passing system calls (for TikTok to mark memory).

fig. 8 shows an example of a such communication pattern. Thread 1 enters the system call S and marks a shared page A (**1**). The system call S blocks until the corresponding call `unblock_S` is called in Thread 2 ((3)). While the page `A` is

```
1: S(A,T1);
```

(a) Thread 1

```
2: write(A);
3: unblock_S(T2);
```

(b) Thread 2

Figure 8: Executing instructions in the specified order causes a deadlock with TikTok

still marked, Thread 2 attempts to write to it, causing it to wait for `S` to finish (**2**). This situation is perculiar:

1. The page **A** is shared between Thread 1 and Thread 2
2. Access to page **A** is not protected by a mutex, or a semaphore
3. System call **S** is a blocking system call that receives a signal from another thread
4. System call **S** reads its arguments from the page **A**
5. Thread 2 needs to write to the same page where the arguments for **S** are stored (page **A**)
6. Even though Threads 1 and 2 can communicate using shared memory, Thread 1 needs to also invoke **S**

While a a synchronization call (locking or signaling) would be a good candidate for S, they are lightweight and their arguments are passed in registers, not in memory. A message-passing call fits the description better. Data from page A would need to be marked, as it is read by the call. Message-passing can also be synchronous, requiring the other thread to receive the message before proceeding. However, why would two threads communicate using both message passing and shared memory?

While it is possible to create deadlocking sequences, they require mixing different inter-process communication paradigms for the same data. During testing we have not encountered a single deadlock caused by TikTok.

Similar sequences can be constructed using the write system call protection presented in **??**. The same argument can be applied in that case - the program would need to write to the same data to the file using both memory mapping and a system call. We have not encountered such a problem.

4 Implementation

fig. 9 illustrates some of the most important information stored on x86. section 4.1 describes the data pertaining to the physical page frame. The data stored in the page table is explained in section 4.2.

4.1 Storing the Page Frame Information

Linux divides physical memory into page frames. Each page frame is represented by a `struct page`. Considering that that this structure is replicated millions of times, every additional field has a tremendeous impact on memory consumption.

To keep the memory consumption low, TikTok uses a single bit in `struct page` to mark page frames. Considering that x86-64 has enough bits in the flag field, we have decided to use one of the flag bits for this purpose. Architectures which have fewer flag bits (such as x86) can instead use some of the bits used by other features (e.g. Kernel Shared Memory or NUMA domains). On fig. 9 this field is denoted by *Page Marked Flag*. `struct page` also stores a pointer to the *reverse mapping* information. Reverse mapping is used to find all PTEs *TikTok* needs to (un)mark.

The marking metadata is stored in a hashmap based on the *page frame number*. The access to these entries is protected by separate mutexes to improve the scalability of the system. The metadata comprises: the page frame number, a number of threads marking the page (*owners*), a number of threads waiting for the page to get unmarked (*guests*), and a *queue* they are waiting on.

4.2 PTE Information

When we mark a page we change some of the flags in the PTEs mapping it (fig. 9):

R/W gets set to *read-only* to prevent writes to the page

SW2 gets the old value of **R/W**

SW3 gets set to 1

Copy-on-write pages are an exception. We mark only the PTE in the process requesting the marking. Writes from other processes will trigger a copy-on-write mechanism, preventing them from changing the data.

TikTok does a partial flush of the TLB and updates the MMU cache to make these changes visible immediately.

5 Related Work

Literature related to TikTok can be broadly divided in 2 groups. The first group are system call wrappers and filters whose main vulnerability TikTok is mitigating. The second one are the mitigations and solutions for double-fetches, which are a superclass of TOCTTOU bugs. We discuss both groups in this section and describe the benefits TikTok brings to the first group, and the advantages over the second group.

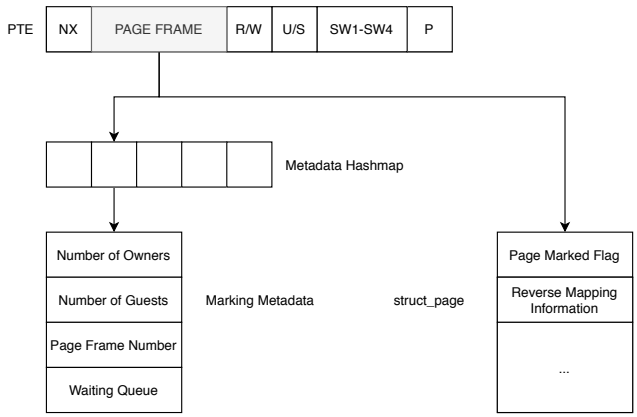


Figure 9: The most important marking information on x86

5.1 System Call Wrappers and Filters

Watson in [17] scrutinized the security of many system call wrappers. Not only that he found that all of them were insecure, Watson described the different types of TOCTTOU bugs and discussed potential fixes. In a short paragraph he mentions that Pawel Dawidek, the creator of CerbNG [20], has experimented with marking arguments read-only. CerbNG was an early system call filtering system for BSD that used copying to protect the arguments. To our knowledge nothing came out of those experiments.

Afterward, Watson briefly discusses problems the such memory marking systems need to solve:

- unnecessary page-faults
- bypassing memory marking using IO system calls
- mapping shared memory late
- handling system calls that write to memory correctly

TikTok addresses all of these issues. Unnecessary page-faults are rare and they are used to make the offending threads wait for unmarking. After the page has been unmarked, the write proceeds without any consequences. Write system call does not proceed until there are no marked pages of the file. If needed, pages are marked when they are mapped. TikTok also postpones all writes to marked pages coming from kernelspace, while allowing the system calls to execute correctly.

Modern system call wrappers can be classified in two groups, based on how they approach the TOCTTOU attack. The first group eliminates all functionality vulnerable to the attack. Linux's *SecComp* [2] and *eBPF* [1] belong to this group. The second group moves the filter checks deeper into the system calls, eliminating the need to read the arguments twice. *Landlock Linux* [9] and Google's *Kernel Runtime Security Instrumentation* (KRSI) [12] embrace this technique.

5.1.1 Partial Solutions

SecComp [2] uses *Berkeley Packet Filter* (BPF) to provide small, programmable filters that execute before the system call. Based on the values in registers, Linux can decide whether to allow, or to prevent a system call. However, BPF cannot dereference pointers because an adversary would by able to bypass those checks using a TOCTTOU attack. *Extended Berkeley Packet Filter* (eBPF) [1] provides larger filters which can also dereference user pointers. However, eBPF cannot be used for security purposes - it cannot stop system calls from executing. EBPF is completely read-only and can only be used for tracing.

5.1.2 LSM-based Solutions

Landlock [9] and KRSI [12] use *Linux Security Module* [6] (LSM) hooks to call filter checks after the arguments have already been copied into the kernel. LSM hooks have been imagined as a set of places where arbitrary checks can be performed before accessing a kernel resource. Execution proceeds only if the execution has been successful. Different security modules can provide different hooks to provide different guarantees (e.g. *SELinux* [13] and *AppArmor* [3]).

Both Landlock and KPSI attach eBPF filters to hooks, allowing users to provide custom rules for system calls. For this solution to work everywhere for perfect syscall filtering, LSM hooks would need to be manually added to all Linux drivers and ioctls. Unfortunately, this is highly impractical and requires a considerable effort from a large group of developers. Considering that LSM focuses on access control to kernel objects, it is questionable if an LSM module can be used to mitigate bugs based on the system call arguments. Some of the bugs could manifest themselves before a hook has been executed. TikTok is a generic solution that does not require modifying the drivers, nor the use of LSM hooks. Once it is deployed, all double-fetch bugs are eliminated from all the drivers. A system call wrapper that uses TikTok can be completely independent from the implementation of the system calls it is filtering.

5.2 Double-Fetch Solutions

Solutions for double fetch bugs can be divided into *static* and *dynamic* techniques. Static techniques do not execute the program, but analyze the source code. Dynamic techniques analyze the execution of the program to find any violations.

5.2.1 Static Analysis Work

Static analysis techniques analyze the source code to find double-fetch bugs. Wang et al. [15] used pattern matching to

find potential double-fetches. They implemented a tool that patches certain double fetches automatically. However, their method in the general case produces false positives which need to be inspected manually. Xu et al. [19] improved on this work by proposing Deadline. Deadline does not use the pattern analysis on the source files to detect double fetches, but a compiler's intermediate representation and constraint solving to eliminate false positives.

Static analysis techniques such as these have the benefit of being able to find bugs in the code that we cannot actually run (e.g. we are missing hardware to test the drivers). However, they are meant for bug detection, not mitigation. Even though the tools can fix some bugs automatically, this is not always possible. The TOCTTOU bug is in system call wrappers by design. Another problem are the double fetches which are not visible in the source, nor in the intermediate representation. Compilers can introduce such invisible double fetches when allocating registers to variables. TikTok is a mitigation technique that works even in such cases.

5.2.2 Dynamic Analysis Work

Google Project Zero's Bochspwn [5] uses an emulator to detect double-fetches. It found an quite a large number of bugs in the Windows kernel. Bochpwn works on binaries, does not require access to the source code and it detects bugs introduced by compilers. However, it is also limited to the detection of double-fetches. Similarly to work prestented in section 5.2.1, developers need to manually fix the bugs. However, with dynamic analysis a double fetch also needs to be executed, limiting this techniques to the core kernel and to the drivers with the available hardware.

A big leap in dynamic analysis techniques has been presented by Schwartz et al. [10]. The first part of the paper introduces DECAF - a framework that uses side-channel attacks to create a fuzzing oracle for double-fetch bugs. While Bochspwn relies on emulation, slowing it down significantly, DECAF runs natively. It also eliminates false positives by automatically exploiting found bugs.
Schwartz et al. then discuss a real-time mitigation technique for double fetches - DropIt. DropIt uses Intel's *Transactional Synchronization Extensions* (TSX) [4] in a creative way to prevent double fetch bugs. By encapsulating the code in a TSX transaction, writes from other threads to the addresses we have previously read from or written to will result in the transaction being aborted. However, the code executing inside a TSX transaction is severly limited. All reads must fit in the L3 cache, and all writes in L1, with some instructions being forbidden. TikTok has none of those limitations. It works on non-Intel processors and relies on page tables for protection - a technique which has been present for several decades.

Acknowledgments

Thanks, mom!

Availability

The source code of TikTok is available at LINK. It has been released under the GNU Public Licence.

References

[1] ebpf.

[2] Seccomp.

[3] Andreas Gruenbacher and Seth Arnold. Apparmor technical documentation, 2007.

[4] Intel. Intel 64 and ia-32 architectures developer's manual. 1, 2019.

[5] GC Mateusz Jurczyk and Gynvael Coldwind. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. *Black Hat USA Briefings (Black Hat USA)*, 2013.

[6] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, pages 17–31. ACM Berkeley, CA, 2002.

[7] Mathias Payer and Thomas R Gross. Protecting applications against tocttou races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 215–226, 2012.

[8] Calton Pu and Jinpeng Wei. A methodical defense against tocttou attacks: The edgi approach. In *Proceedings of the 2006 International Symposium on Secure Software Engineering*, 2006.

[9] Mickaël Salaün. Landlock lsm.

[10] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 587–600, 2018.

[11] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system principles*. John Wiley & Sons, 10 edition, 2018.

[12] K. P. Singh. Kernel runtime security instrumentation.

[13] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[14] Dan Tsafrir, Tomer Hertz, David A Wagner, and Dilma Da Silva. Portably solving file tocttou races with hardness amplification. In *FAST*, volume 8, pages 1–18, 2008.

[15] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1–16, 2017.

[16] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6):e4345, 2018.

[17] Robert NM Watson. Exploiting concurrency vulnerabilities in system call wrappers. *WOOT*, 7:1–8, 2007.

[18] Jinpeng Wei and Calton Pu. Modeling and preventing tocttou vulnerabilities in unix-style file systems. *computers & security*, 29(8):815–830, 2010.

[19] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[20] Slawek Zak, Przemyslaw Frasunek, and Pawel Jakub Dawidek. Cerbng.