Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels

Meng Xu, Chenxiong Qian, Kangjie Lu+ , Michael Backes*, Taesoo Kim

Georgia Tech | University of Minnesota+ | CISPA, Germany*
What is Double-Fetch?
Address Space Separation

A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space
No Dereference on Userspace Pointers

A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

```c
void kfunc (int __user *uptr, int *kptr) {
    ......
}
```
No Dereference on Userspace Pointers

A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

```c
void kfunc (int __user *uptr, int *kptr) {
    ......
}
```
No Dereference on Userspace Pointers

A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

```c
void kfunc (int __user *uptr, int *kptr) {
    *kptr = *uptr; /* X */
    ....
}
```
No Dereference on Userspace Pointers

A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

```c
void kfunc(int __user *uptr, int *kptr) {
    copy_from_user(kptr, uptr, 4);  // ✔
    ....
}
```

User / Program Address Space

Kernel Address Space

- No Dereference on Userspace Pointers

- Uninitialized:
  ```c
  0xDEADBEEF
  ```

- Kernel Addresses:
  ```c
  0xFFFFFFFF
  ```

- User Memory:
  ```c
  0xC0000000
  ```

- User Memory:
  ```c
  0x00000000
  ```

- Total User Memory:
  ```c
  1 GB
  ```

- Total Kernel Memory:
  ```c
  3 GB
  ```

- Total Memory:
  ```c
  4 GB
  ```
Shared Userspace Pointer Across Threads

A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

```c
void kfunc (int __user *uptr, int *kptr) {
    copy_from_user(kptr, uptr, 4);
    .......
}
```
Shared Userspace Pointer Across Threads

A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

```c
void kfunc (int __user *uptr, int *kptr) {
    copy_from_user(kptr, uptr, 4);
    ......
}
```
Why Double-Fetch?

```c
static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

// first fetch
if (get_user(size, &uattr->size))
    return -EFAULT;

// sanity checks
if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

// second fetch
if (copy_from_user(attr, uattr, size))
    return -EFAULT;

......
}

// BUG: when attr->size is used later
memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
Why Double-Fetch?

```c
static int perf_copy_attr_simplified(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *attr)
{
    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE ||
        size < PERF_ATTR_SIZE_VER0)
        return -EINVAL;

    // second fetch
    if (copy_from_user(attr, uattr, size))
        return -EFAULT;

    .......
}
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
Why Double-Fetch?

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
Why Double-Fetch?

static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

u32 size;

// first fetch
if (get_user(size, &uattr->size))
    return -EFAULT;

// sanity checks
if (size > PAGE_SIZE ||
    size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

......
Why Double-Fetch?

```c
static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE ||
        size < PERF_ATTR_SIZE_VER0)
        return -EINVAL;

    ......}
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
### Why Double-Fetch?

```c
static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

        u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE ||
        size < PERF_ATTR_SIZE_VER0)
        return -EINVAL;

    // second fetch
    if (copy_from_user(attr, uattr, size))
        return -EFAULT;
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
Why Double-Fetch?

```
static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

  u32 size;

  // first fetch
  if (get_user(size, &uattr->size))
    return -EFAULT;

  // sanity checks
  if (size > PAGE_SIZE ||
      size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

  // second fetch
  if (copy_from_user(attr, uattr, size))
    return -EFAULT;

  .......
}
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
What Goes Wrong in This Process?
Adapted from `perf_copy_attr` in file `kernel/events/core.c`

```c
static int perf_copy_attr_simplified
    (struct perf_event_attr __user *uattr,
     struct perf_event_attr *attr) {

    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    ......
}
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
Wrong Assumption: Atomicity in Syscall

static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

  u32 size;

  // first fetch
  if (get_user(size, &uattr->size))
    return -EFAULT;

  // sanity checks
  if (size > PAGE_SIZE ||
      size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

  // BUG: when attr->size is used later
  memcpy(buf, attr, attr->size);

Adapted from perf_copy_attr in file kernel/events/core.c
Wrong Assumption: Atomicity in Syscall

```c
static int perf_copy_attr_simplified(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *attr) {

    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE ||
        size < PERF_ATTR_SIZE_VER0)
        return -EINVAL;

    // second fetch
    if (copy_from_user(attr, uattr, size))
        return -EFAULT;

    ......
}
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`
When The Exploit Happens

Adapted from perf_copy_attr in file kernel/events/core.c
Why Double-Fetch is Prevalent in Kernels?

1. Size checking
2. Dependency look-up
3. Protocol/signature check
4. Information guessing
5. ……
Double-Fetch: Dependency Lookup

```c
void mptctl_simplified(unsigned long arg) {
    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
    MPT_ADAPTER *iocp = NULL;

    // first fetch
    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
    return -EFAULT;

    // dependency lookup
    if (mpt_verify_adapter(khdr.ioctnum, &iocp) < 0 || iocp == NULL)
    return -EFAULT;

    // dependency usage
    mutex_lock(&iocp->ioctl_cmds.mutex);
    struct mpt_fw_xfer kfwdl, __user *ufwldl = (void __user *) arg;

    // second fetch
    if (copy_from_user(&kfwdl, ufwldl, sizeof(struct mpt_fw_xfer)))
    return -EFAULT;

    // BUG: kfwdl.ioctnum might not equal to khdr.ioctnum
    mptctl_do_fw_download(kfwdl.ioctnum, ......);
    mutex_unlock(&iocp->ioctl_cmds.mutex);
}
```

Adapted from __mptctl_ioctl in file drivers/message/fusion/mptctl.c
Double-Fetch: Dependency Lookup

```c
void mptctl_simplified(unsigned long arg) {
    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
    MPT_ADAPTER *iocp = NULL;

    // first fetch
    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
        return -EFAULT;

    // dependency lookup
    if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
        return -EFAULT;

    // dependency usage
    Acquire mutex lock for ioc 01

    // second fetch
    if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
        return -EFAULT;

    Do do_fw_download for ioc 02

    Release mutex lock for ioc 01
}
```

Adapted from __mptctl_ioctl in file drivers/message/fusion/mptctl.c
Double-Fetch: Protocol/Signature Check

```c
void tls_setsockopt_simplified(char __user *arg) {
    struct tls_crypto_info header, *full = /* allocated before */;

    // first fetch
    if (copy_from_user(&header, arg, sizeof(struct tls_crypto_info)))
        return -EFAULT;

    // protocol check
    if (header.version != TLS_1_2_VERSION)
        return -ENOTSUPP;

    // second fetch
    if (copy_from_user(full, arg, sizeof(struct tls12_crypto_info_aes_gcm_128)))
        return -EFAULT;

    // BUG: full->version might not be TLS_1_2_VERSION
    do_something_with(full);
}
```

Adapted from do_tls_setsockopt_txZ in file net/tls/tls_main.c
## Prior Works

<table>
<thead>
<tr>
<th></th>
<th>Bochspwn (BlackHat’13)</th>
<th>DECAF (arXiv’17)</th>
<th>Pengfei et. al., (Security’17)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel</strong></td>
<td>Windows</td>
<td>Linux</td>
<td>Linux and FreeBSD</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td>VMI</td>
<td>Kernel fuzzing</td>
<td>Lexical Code Matching</td>
</tr>
<tr>
<td><strong>Patten</strong></td>
<td>Memory access timing</td>
<td>Cache side channel</td>
<td>Size checking</td>
</tr>
<tr>
<td><strong>Code Coverage</strong></td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Manual Effort</strong></td>
<td>Large</td>
<td>Large</td>
<td>Large</td>
</tr>
</tbody>
</table>
## Prior Works

<table>
<thead>
<tr>
<th></th>
<th>Bochspwn (BlackHat’13)</th>
<th>DECAF (arXiv’17)</th>
<th>Pengfei et. al., (Security’17)</th>
<th>Deadline (Our work)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel</strong></td>
<td>Windows</td>
<td>Linux</td>
<td>Linux and FreeBSD</td>
<td>Linux and FreeBSD</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td>VMI</td>
<td>Kernel fuzzing</td>
<td>Lexical Code Matching</td>
<td>Symbolic Checking</td>
</tr>
<tr>
<td><strong>Pattern</strong></td>
<td>Memory access timing</td>
<td>Cache side channel</td>
<td>Size checking</td>
<td>Formal Definitions</td>
</tr>
<tr>
<td><strong>Code Coverage</strong></td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td><strong>Manual Effort</strong></td>
<td>Large</td>
<td>Large</td>
<td>Large</td>
<td>Small</td>
</tr>
</tbody>
</table>
Double-Fetch Bugs: Towards A Formal Definition

**Fetch:** A pair \((A, S)\), where
- \(A\) - the starting address of the fetch,
- \(S\) - the size of memory copied into kernel.

**Overlapped-fetch:** Two fetches, \((A_0, S_0)\) and \((A_1, S_1)\), where
\[ A_0 \leq A_1 < A_0 + S_0 \quad \| \quad A_1 \leq A_0 < A_1 + S_1 \]
- The overlapped memory region is marked as \((A_{01}, S_{01})\).
- The copied value during 1st fetch is \((A_{01}, S_{01}, 0)\).
- The copied value during 2nd fetch is \((A_{01}, S_{01}, 1)\).
Overlapped-Fetch Case 1

get_user(attr, &uptr->attr)

copy_from_user(kptr, uptr, size)
Overlapped-Fetch Case 2

```c
copy_from_user(
  khdr, uptr, sizeof(struct hdr)
)

copy_from_user(
  kmsg, uptr, khdr->size
)
```

(A₀₁, S₀₁, 0) khdr->size, khdr->type, ...

(A₀₁, S₀₁, 1) kmsg->size, kmsg->type, ...
Double-Fetch Bugs: Towards A Formal Definition

**Control dependence:** A variable $V \in (A_{01}, S_{01})$ and $V$ must satisfy a set of constraints before the second fetch can happen.
Double-Fetch Bugs: Towards A Formal Definition

Control dependence: A variable $V \in (A_{01}, S_{01})$ and $V$ must satisfy a set of constraints before the second fetch can happen.

```c
void tls_setsockopt_simplified(char __user *arg) {
    struct tls_crypto_info header, *full = /* allocated before */;

    // first fetch
    if (copy_from_user(&header, arg, sizeof(struct tls_crypto_info)))
        return -EFAULT;

    // protocol check
    if (header.version != TLS_1_2_VERSION)
        return -ENOTSUPP;

    // second fetch
    if (copy_from_user(full, arg,
                        sizeof(struct tls12_crypto_info_aes_gcm_128)))
        return -EFAULT;

    // BUG: full->version might not be TLS_1_2_VERSION
    do_something_with(full);
}
```

Overlapped variable $V$: header.version

The constraint it must satisfy:
header.version == TLS_1_2_VERSION

Expect:
full->version == TLS_1_2_VERSION
Double-Fetch Bugs: Towards A Formal Definition

**Data dependence**: A variable $V \in (A_{01}, S_{01})$ and $V$ is consumed before or on the second fetch (e.g., involved in calculation, passed to function calls, etc).
Data dependence: A variable $V \in (A_01, S_01)$ and $V$ is consumed before or on the second fetch.

```c
void mptctl_simplified(unsigned long arg) {
    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
    MPT_ADAPTER *iocp = NULL;

    // first fetch
    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
        return -EFAULT;

    // dependency lookup
    if (mpt_verify_adapter(khdr.iocnum, &iocp) <
        return -EFAULT;

    // dependency usage
    mutex_lock(&iocp->ioctl_cmds.mutex);
    struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;

    // second fetch
    if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
        return -EFAULT;

    // BUG: kfwdl.iocnum might not equal to khdr.iocnum
    mptctl_do_fw_download(kfwdl.iocnum, ......);
    mutex_unlock(&iocp->ioctl_cmds.mutex);
}
```

Overlapped variable $V$: khdr.iocnum

Data dependence: mpt_verify_adapter(khdr.iocnum, &iocp)

Expect: kfwdl.iocnum == khdr.iocnum
Double-Fetch Bugs: Towards A Formal Definition

1. Two fetches from userspace memory that cover an overlapped region.

2. A relation must exist on the overlapped region between the two fetches. The relation can be either control-dependence or data-dependence.

3. We cannot prove that the relation established after first fetch still holds after the second fetch.

If all conditions are satisfied: a user thread might race condition to change the content in the overlapped region, and thus, to destroy the relation.
How to Find Double-Fetch Bugs?
How to Find Double-Fetch Bugs?

1. Find as many double-fetch pairs as possible, construct the code paths associated with each pair.

2. Symbolically check each code path and determine whether the two fetches makes a double-fetch bug.
Fetch Pair Collection

**Goal**: Statically enumerate all pairs of fetches that could possibly occur.
**Goal:** Statically enumerate all pairs of fetches that could possibly occur.

**Ideal solution (top-down):**
1. Identify all fetches in the kernel
2. Construct a complete, inter-procedural CFG for the whole kernel
3. Perform pair-wise reachability tests for each pair of fetches

**Our solution (bottom-up):**
1. Identify all fetches in the kernel
2. For each fetch, within the function it resides in, scan its reaching instructions for fetches or fetch-involved functions
Fetch Pairs Collection

**Goal:** Statically enumerate all pairs of fetches that could possibly occur.

**Ideal solution (top-down):**
- ✔️ 1. Identify all fetches in the kernel
- ✗ 2. Construct a complete, inter-procedural CFG for the whole kernel
- ✗ 3. Perform pair-wise reachability tests for each pair of fetches

**Our solution (bottom-up):**
- ✔️ 1. Identify all fetches in the kernel
- ✔️ 2. For each fetch, within the function it resides in, scan its reaching instructions for fetches or fetch-involved functions
static void enclosing_function(
  struct msg_hdr __user *uptr,
  struct msg_full *kptr
) {
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  if (copy_from_user(kptr, uptr, size))
    return -EFAULT;
  ...
}
Bottom-up Fetch Pairs Collection

```c
static void enclosing_function(
    struct msg_hdr __user *uptr,
    struct msg_full *kptr
) {
    ...
    ...
    ...
    ...
    ...
    ...
    if (copy_from_user(kptr, uptr, size))
        return -EFAULT;
    ...
}
```

Search through the reaching instructions
Bottom-up Fetch Pairs Collection

static void enclosing_function(
    struct msg_hdr __user *uptr,
    struct msg_full *kptr
) {
    ...
    ...
    if (get_user(size, &uptr->size))
        return -EFAULT;
    ...
    ...
    ...
    if (copy_from_user(kptr, uptr, size))
        return -EFAULT;
    ...
}

[Case 1]
Found another fetch
==>
found a fetch pair
Bottom-up Fetch Pairs Collection

[Case 2]
Found a fetch-involved function
  ==> inline the function, found a fetch pair

static void enclosing_function(
  struct msg_hdr __user *uptr,
  struct msg_full *kptr
) {
  ...
  ...
  size = get_size_from_user(uptr);
  ...
  ...
  ...
  ...
  if (copy_from_user(kptr, uptr, size))
    return -EFAULT;
  ...
}
Bottom-up Fetch Pairs Collection

static void enclosing_function(
    struct msg_hdr __user *uptr,
    struct msg_full *kptr
) {
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    if (copy_from_user(kptr, uptr, size))
        return -EFAULT;
    ...
}

[Case 3]
No fetch-related instruction
==> Not a double-fetch
How to Find Double-Fetch Bugs?

1. Find as many double-fetch pairs as possible, construct the code paths associated with each pair.

2. Symbolically check each code path and determine whether the two fetches makes a double-fetch bug.
Symbolic Checking

**Goal:** Symbolically execute the code path that connects two fetches and determine whether the two fetches satisfy all the criteria set in formal definition of double-fetch bug, i.e.

- Overlap
- Have a relation (control or data dependence)
- We cannot prove the relation still holds after second fetch
Symbolic Checking

```c
static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

  u32 size;

  // first fetch
  if (get_user(size, &uattr->size))
    return -EFAULT;

  // sanity checks
  if (size > PAGE_SIZE ||
    size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

  // second fetch
  if (copy_from_user(attr, uattr, size))
    return -EFAULT;

  .......

  // BUG: when attr->size is used later
  memcpy(buf, attr, attr->size);
```
Symbolic Checking

```c
static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {
  u32 size;

  // first fetch
  if (get_user(size, &uattr->size))
    return -EFAULT;

  // sanity checks
  if (size > PAGE_SIZE ||
      size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

  // second fetch
  if (copy_from_user(attr, uattr, size))
    return -EFAULT;

  .......
}

// init root SR
$0 = PARM(0), @0 = UMEM(0) // uattr
$1 = PARM(1), @1 = KMEM(1) // attr

// BUG: when attr->size is used later
memcpy(buf, attr, attr->size);
```
Symbolic Checking

```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {
    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
      return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE ||
        size < PERF_ATTR_SIZE_VER0)
      return -EINVAL;

    // second fetch
    if (copy_from_user(attr, uattr, size))
      return -EFAULT;

    ......
  }

  // init root SR
  $0 = PARM(0), @0 = UMEM(0) // uattr
  $1 = PARM(1), @1 = KMEM(1) // attr
  ---
  // first fetch
  fetch(F1): {A = $0 + 4, S = 4}
  $2 = @0(4, 7, U0), @2 = nil // size
  ---

  // check fetch overlap

  [solve] --> satisfiable with @0(4, 7, U)

  // check double-fetch bug
  [prove] @0(4, 7, U0) == @0(4, 7, U1)

  --> fail: no constraints on @0(4, 7, U1)

  // BUG: when attr->size is used later
  memcpy(buf, attr, attr->size);
```
static int perf_copy_attr_simplified (struct perf_event_attr __user *uattr, struct perf_event_attr *attr) {
    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE ||
        size < PERF_ATTR_SIZE_VER0)
        return -EINVAL;

    // second fetch
    if (copy_from_user(attr, uattr, size))
        return -EFAULT;

    .......
}

// BUG: when attr->size is used later
memcpy(buf, attr, attr->size);

// init root SR
$0 = PARM(0), @0 = UMEM(0)  // uattr
$1 = PARM(1), @1 = KMEM(1)  // attr
---
// first fetch
fetch(F1): {A = $0 + 4, S = 4}
$2 = @0(4, 7, U0), @2 = nil  // size
---
// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0
---
// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0
---
// Sanity checks
[prove] @0(4, 7, U0) == @0(4, 7, U1)
---
// Sanity checks
[prove] @0(4, 7, U0) == @0(4, 7, U1)
---
// Sanity checks
[prove] @0(4, 7, U0) == @0(4, 7, U1)
---
Symbolic Checking

```c
static int perf_copy_attr_simplified
(struct perf_event_attr __user *uattr,
 struct perf_event_attr *attr) {

  u32 size;

  // first fetch
  if (get_user(size, &uattr->size))
    return -EFAULT;

  // sanity checks
  if (size > PAGE_SIZE ||
     size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

  // second fetch
  if (copy_from_user(attr, uattr, size))
    return -EFAULT;

  ......}

// init root SR
$0 = PARM(0), @0 = UMEM(0) // uattr
$1 = PARM(1), @1 = KMEM(1) // attr

---

// first fetch
fetch(F1): {A = $0 + 4, S = 4}
@2 = nil // size

---

// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0

---

// second fetch
fetch(F2): {A = $0, S = $2}
@1(0, $2 - 1, K) = @0(0, $2 - 1, U1)

---

// BUG: when attr->size is used later
memcpy(buf, attr, attr->size);
```
Symbolic Checking

```c
static int perf_copy_attr_simplified
    (struct perf_event_attr __user *uattr,
     struct perf_event_attr *attr) {
    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE ||
        size < PERF_ATTR_SIZE_VER0)
        return -EINVAL;

    // second fetch
    if (copy_from_user(attr, uattr, size))
        return -EFAULT;

    ......

    // BUG: when attr->size is used later
    memcpy(buf, attr, attr->size);
```
Symbolic Checking

```c
static int perf_copy_attr_simplified(
    struct perf_event_attr __user *uattr,
    struct perf_event_attr *attr) {

    u32 size;

    // first fetch
    if (get_user(size, &uattr->size))
        return -EFAULT;

    // sanity checks
    if (size > PAGE_SIZE || size < PERF_ATTR_SIZE_VER0)
        return -EINVAL;

    // second fetch
    if (copy_from_user(attr, uattr, size))
        return -EFAULT;

    ......
}
```

```c
// init root SR
$0 = PARM(0), @0 = UMEM(0)  // uattr
$1 = PARM(1), @1 = KMEM(1)  // attr

// first fetch
fetch(F1): {A = $0 + 4, S = 4}
    $2 = @0(4, 7, U0), @2 = nil  // size

// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0

// second fetch
fetch(F2): {A = $0, S = $2}
    @1(0, $2 - 1, K) = @0(0, $2 - 1, U1)

// check fetch overlap
[prove]
    -- fail: no constraints on @0(4, 7, U1)
```

// BUG: when attr->size is used later
memcpy(buf, attr, attr->size);
```

54
Please refer to our paper for a comprehensive demonstration on how Deadline handles

1. Loop unrolling
2. Pointer resolving
Findings

- 24 bugs found in total
  - 23 bugs in Linux kernel and 1 in FreeBSD kernel
- 9 bugs have been patched with the fix we provide
- 4 bugs are acknowledged, we are still working on the fix
- 9 bugs are pending for review
- 2 bugs are marked as “won’t fix”
Double-Fetch Bug Mitigations

- The basic idea is to re-assure the control-dependence and data-dependence between the two fetches. In other words, the automaticity in user space memory fetches during the execution of the syscall.
Double-Fetch Bug Mitigations

• The basic idea is to re-assure the control-dependence and data-dependence between the two fetches. In other words, the automaticity in user space memory fetches during the execution of the syscall.

• Based on our experience and our communications with kernel developers, we found four patterns in patching double-fetch bugs.
Double-Fetch Bug Mitigations

1. Override after second fetch.

```c
diff --git a/kernel/events/core.c b/kernel/events/core.c
index ee20d4c..c0d7946 100644
--- a/kernel/events/core.c
+++ b/kernel/events/core.c
@@ -9611,6 +9611,8 @@ static int perf_copy_attr(struct perf_event_attr __user *uattr,
   if (ret)
     return -EFAULT;
+   attr->size = size;
+ if (attr->__reserved_1)
   return -EINVAL;
```

Override the overlapped memory (attr->size) with the value from the first fetch (size).
Double-Fetch Bug Mitigations

2. Abort on change detected.

```
+    /* check the length of messages copied in is the same as the
+     * what we get from the first loop
+     */
+    if ((char *)kcmsg - (char *)kcmsg_base != kcmlen)
+        goto Einval;
+    /* Ok, looks like we made it. Hook it up and return success. */
+    kmsg->msg_control = kcmsg_base;
+    kmsg->msg_controllen = kcmlen;
```
Double-Fetch Bug Mitigations

3. Refactor overlapped copies into incremental copies.

```c
block/scsi_ioctl.c | 8 ++++
1 file changed, 7 insertions(+), 1 deletion(-)
diff --git a/block/scsi_ioctl.c b/block/scsi_ioctl.c
index 7440de4..8fe1e05 100644
--- a/block/scsi_ioctl.c
+++ b/block/scsi_ioctl.c
@@ -463,7 +463,13 @@ int sg_scsi_ioctl(struct request_queue *q, struct gendisk *disk, fmode_t mode,
  * /
  err = -EFAULT;
  req->cmd_len = cmdlen;
- if (copy_from_user(req->cmd, sic->data, cmdlen))
+  + /* avoid copying the opcode twice
+     * /
+    memcpy(req->cmd, &opcode, sizeof(opcode));
+    if (copy_from_user(req->cmd + sizeof(opcode),
+                        sic->data + sizeof(opcode), cmdlen - sizeof(opcode)))
+       goto error;
+  if (in_len && copy_from_user(buffer, sic->data + cmdlen, in_len))
```

When copying the whole message, skip the information copied in the first fetch (+ sizeof(opcode)).
Double-Fetch Bug Mitigations

4. Refactor overlapped copies into a single-fetch.
Such a strategy is usually very complex and requires careful refactoring.
Double-Fetch Bug Mitigations

Unfortunately, not all double-fetch bugs can be patched with these patterns. Some requires heavy refactoring of existing codebase or re-designing of structs, which requires substantial manual effort.
Double-Fetch Bug Mitigations

Unfortunately, not all double-fetch bugs can be patched with these patterns. Some requires heavy refactoring of existing codebase or re-designing of structs, which requires substantial manual effort.

Recently, DECAF has provided a promising solution in using TSX-based techniques to ensure user space memory access automaticity in syscall execution.
Limitations of Deadline

- **Source code coverage**
  - Files not compilable under LLVM.
  - Special combination of kernel configs (e.g., `CONFIG_*`).

- **Execution path construction**
  - Limit on total number of paths explored per fetch pair (4096).
  - Loop unrolling (limited to unroll once only).

- **Symbolic checking**
  - Ignores inline assemblies.
  - Imprecise pointer to memory object mapping.
  - Assumption on enclosing function.
Conclusion

• Detecting double-fetch bugs without a precise and formal definition has led to many false alerts and tremendous manual effort.

• Deadline is based on a precise modeling of double-fetch bugs and achieves both high accuracy and high scalability.

• Application beyond kernels: hypervisors, browsers, TEE, etc.

• Logic bugs are on the rise! We hope that more logic bugs can be modeled and checked systematically.

https://github.com/sslab-gatech/deadline