Lec05: Advanced Topics in Security

Taesoo Kim

Goals and Lessons

- Understand *modern* attack/defenses: **ROP** and **CFI**
- Understand three classes of vulnerabilities
  - **Type confusion**: e.g., bad casting in C++
  - **Race condition**: e.g., double fetching
  - **Uninitialized read**: e.g., struct padding issues
- Learn them from real-world examples (e.g., Chrome, Linux)

Modern Exploit against DEP (NX)

- Return-oriented Programming (ROP)
- Reusing code snippets (called gadgets) instead of injecting shellcode
  - e.g., ret-to-libc: ret → system("/bin/sh")

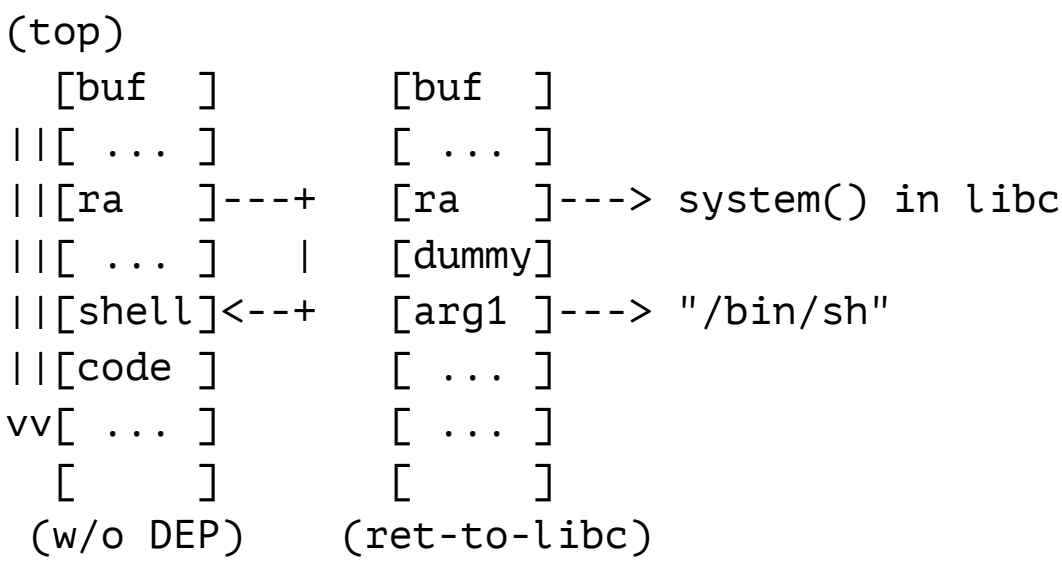
Example: Stack Smashing (w/o DEP)

```
1   main() {
2     char buf[16];
3     scanf("%s", buf);
4   }
```

```
(top)
  [buf  ]
||[ ... ]
||[ra   ]---+
||[ ... ]   |
||[shell]<--+ (inject shellcode)
||[code ]
vv[ ... ]
  [     ]
 (w/o DEP)
```

Example: Stack Smashing vis Ret-to-libc

```
1   main() {
2     char buf[16];
3     scanf("%s", buf);
4   }
```



Example: Stack Smashing vis Ret-to-libc

- Q. What happens when system() returns?
- Q. Is there any way to “gracefully” terminate?

```
(top)
  [buf  ]
  [ ... ]
  [ra   ]---> system() in libc
  [dummy]
  [arg1 ]---> "/bin/sh"
  [ ... ]
  [ ... ]
  [     ]
(ret-to-libc)
```

Example: Stack Smashing vis Ret-to-libc

- A. “dummy” → exit()
- A. its first argument → 0

```
(top)                                  system("/bin/sh")
  [buf  ]                              exit(0)
  [ ... ]
  [ra1  ]---> system() in libc
  [ra2  ]---> exit()
  [arg1 ]---> "/bin/sh"
  [arg2 ]---> 0
  [ ... ]
  [     ]
(ret-to-libc)
```

Example: Executing Two Functions

- The return address of main() is smashed by the buffer overflow
- It returns to system() instead of the original caller

```
main(): ret
```

```
        (top)
           [buf  ]
           [ ... ]
esp =>     [ra1  ]---> system() in libc
           [ra2  ]---> exit()
           [arg1 ]---> "/bin/sh"
           [arg2 ]---> 0
           [ ... ]
           [     ]
         (ret-to-libc)
```

Example: Executing Two Functions

- dummy (i.e., ptr to exit()) is now considered as caller of system()
- arg1 is the first argument of system, “/bin/sh”

```
main(): ret
system(): ..
 caller: exit()
 arg: "/bin/sh"
```

```
    (top)
       [buf  ]
       [ ... ]
       [     ]
esp =>[ra2  ]---> exit()
      [arg1 ]---> "/bin/sh"
      [arg2 ]---> 0
      [ ... ]
      [     ]
     (ret-to-libc)
```

Example: Executing Two Functions

- dummy (i.e., ptr to exit()) is now considered as caller of system()
- arg1 is the first argument of system, “/bin/sh”

```
main(): ret
system(): ret
```

```
        (top)
           [buf  ]
           [ ... ]
           [     ]
esp =>[ra2  ]---> exit()
      [arg1 ]---> "/bin/sh"
      [arg2 ]---> 0
      [ ... ]
      [     ]
       (ret-to-libc)
```

Example: Executing Two Functions

- dummy (i.e., ptr to exit()) is now considered as caller of system()
- arg1 is the first argument of system, “/bin/sh”

```
main(): ret
system(): ret
exit():
 caller: "/bin/sh" (!!)
 arg: 0
```

```
      (top)
         [buf  ]
         [ ... ]
         [     ]
         [ra2  ]---> exit()
esp =>[arg1 ]---> "/bin/sh"
      [arg2 ]---> 0
         [ ... ]
         [     ]
       (ret-to-libc)
```

Example: Execution More than Two Funcs?

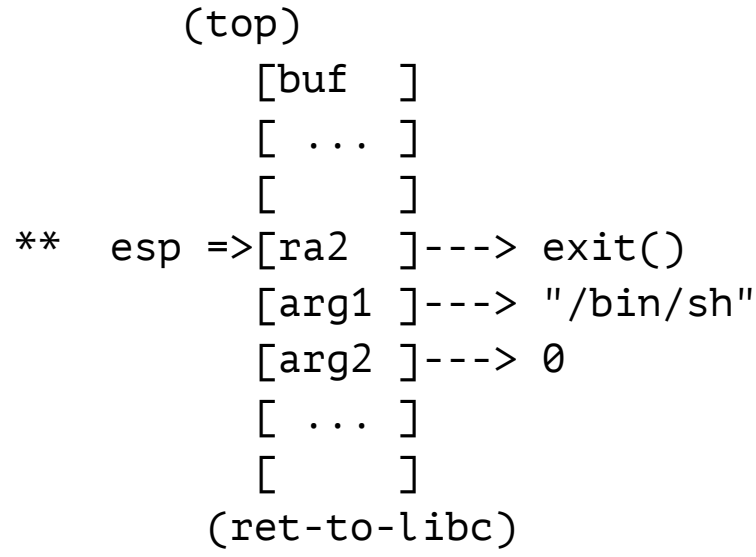
- Can we chain three functions in this way? No!
- ROP generalizes this approach by using

```
(top)                                  system("/bin/sh")
  [buf  ]                              exit(0)
  [ ... ]
  [ra1  ]---> system() in libc
  [ra2  ]---> exit()
  [arg1 ]---> "/bin/sh"          <- "/bin/sh" vs func3?
  [arg2 ]---> 0
  [ ... ]
  [     ]
(ret-to-libc)
```

Example: ROP

- Cleaning up stacks by using pop/ret gadgets
- Chaining them in a general fashion!

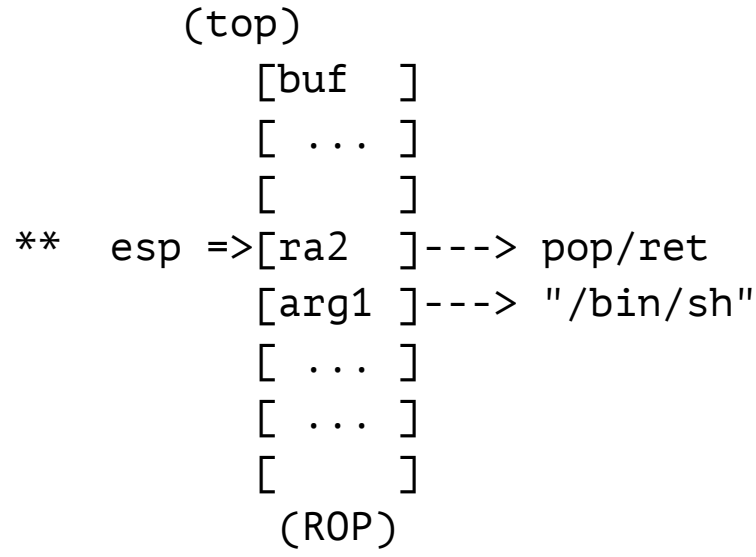
```
main(): ret
system(): ret
```



Example: ROP

- Cleaning up stacks by using pop/ret gadgets
- Chaining them in a general fashion!

```
main(): ret
system(): ret
```



Example: ROP

- Cleaning up stacks by using pop/ret gadgets
- Chaining them in a general fashion!

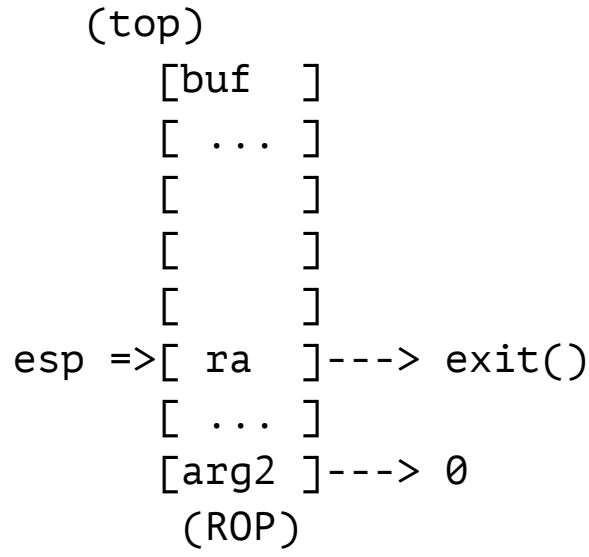
```
main(): ret
system(): ret
pop
```

```
        (top)
           [buf  ]
           [ ... ]
           [     ]
           [     ]
esp =>     [arg1 ]---> "/bin/sh"
           [ ra  ]---> exit()
           [ ... ]
           [arg2 ]---> 0
            (ROP)
```

Example: ROP

- Cleaning up stacks by using pop/ret gadgets
- Chaining them in a general fashion!

```
main(): ret
system(): ret
pop
ret
```



Example: Beyond Two Functions!

- Cleaning up stacks by using pop/ret gadgets
- Chaining them in a general fashion!

```
(top)                                  system("/bin/sh")
  [buf  ]                              exit(0)
  [ ... ]                              func3(arg3, arg4)
  [ra1  ]---> system() in libc |
* [gadgt]---> pop/ret          | system()
  [arg1 ]---> "/bin/sh"        |
  [ra2  ]---> exit()           =
* [gadgt]---> pop/ret          = exit()
  [arg2 ]---> 0                =
  [ra3  ]---> func3()          +
* [gadgt]---> pop/pop/ret      + func3()
  [arg3 ] ...                  +
```

Defenses: Control-flow Integrity

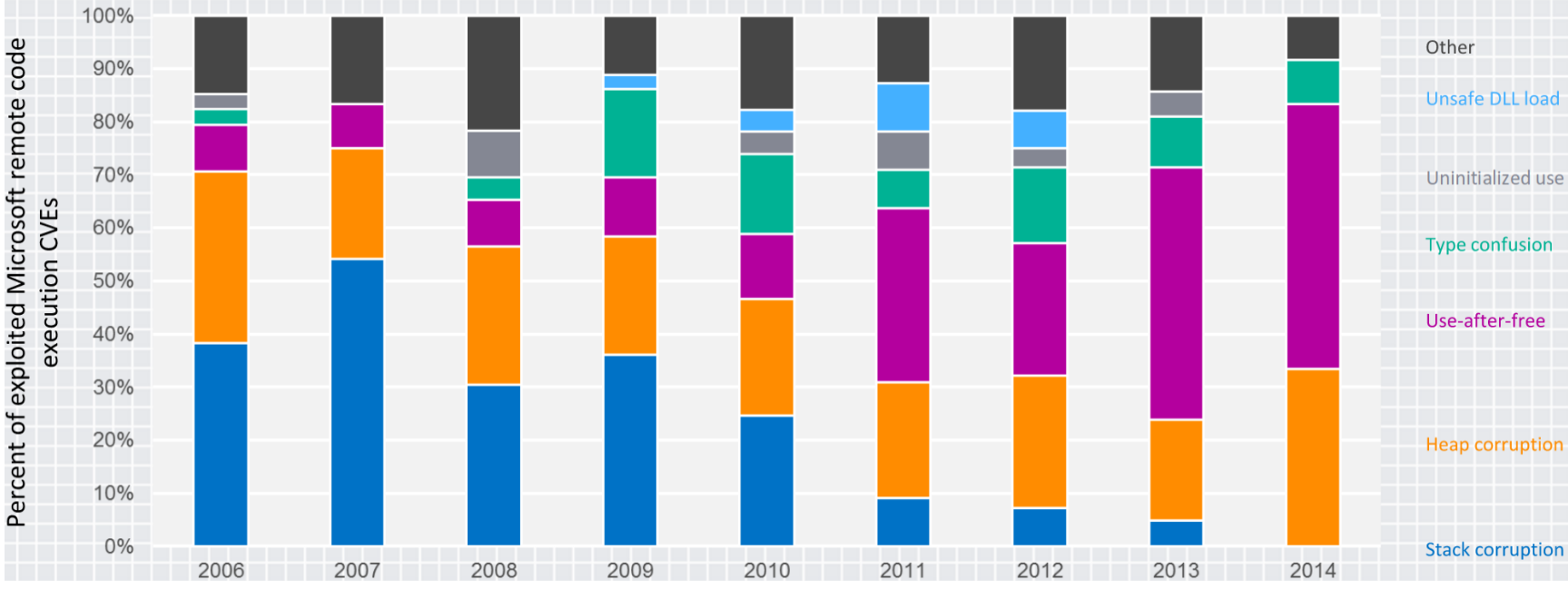
- Control-flow at the compilation time should be enforced at runtime!
- Vendors adoptions:
  - Control-flow Guard (CFG) by Microsoft from Windows 10/8.1
  - LLVM forward CFI by Android (Pixel 3)
  - LLVM forward CFI by Google Chrome (dev)
- Hardware solutions:
  - Intel: Control-flow Enforcement (CET)
  - ARM v8,3: PAC in iOS 12 by Apple

Basic Idea: Enforcing Control-flow Graphs

- Forward CFI: protecting indirect calls (e.g., jmp eax)
  - Course-grained: static call graphs
  - Finer-grained: type-based, Input-based, etc.
- Backward CFI: protecting returns (e.g., ret)
  - Safe/shadow stack

→ CPU overheads: 5-20% in forward CFIs, 1-5% in backwards CFIs

Trends of Vulnerability Classes

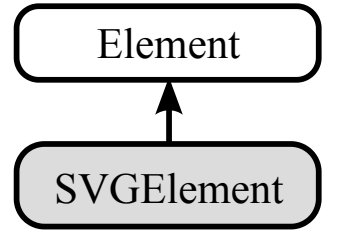


Ref. Exploitation Trends: From Potential Risk to Actual Risk, RSA 2015

Three Emerging Classes of Vulnerabilities

1. Type confusion: e.g., bad casting in C++
2. Race condition: e.g., double fetching
3. Uninitialized read: e.g., struct padding issues

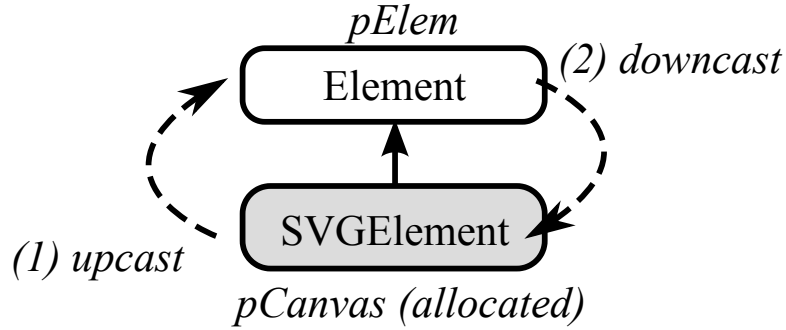
Type Casting in C++



pCanvas (allocated)

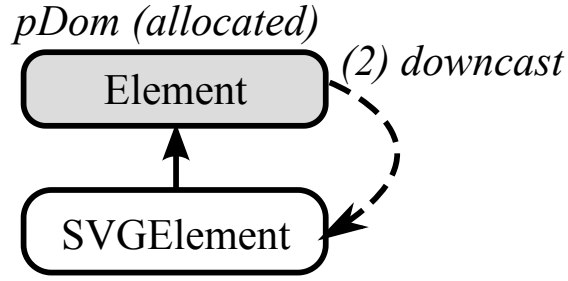
```
1  class SVGElement: public Element { ... };
2
3  SVGElement *pCanvas = new SVGElement();
```

Upcasting and Downcasting in C++



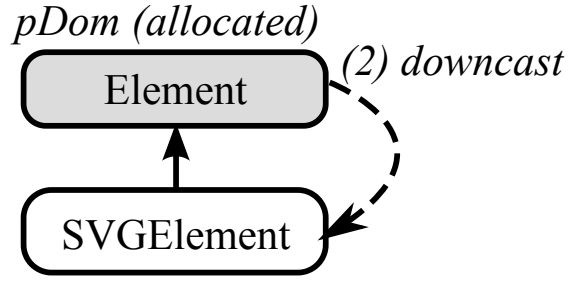

```
1  // (1) valid upcast from pCanvas to pElem
2  Element *pElem = static_cast<Element*>(pCanvas);
3
4  // (2) valid downcast from pElem to pCanvasAgain (== pCanvas
5  SVGElement *pCanvasAgain = static_cast<SVGElement*>(pElem);
```

Dynamic Casting in C++ (Downcasting)




```
1  Element *pDom = new Element();
2
3  // (3) invalid downcast with dynamic_cast, but no corruption
4  SVGElement *p = dynamic_cast<SVGElement*>(pDom);
5  if (p) {
6      p->m_className = "my-canvas";
7  }
```

Static Casting in C++ (Downcasting)

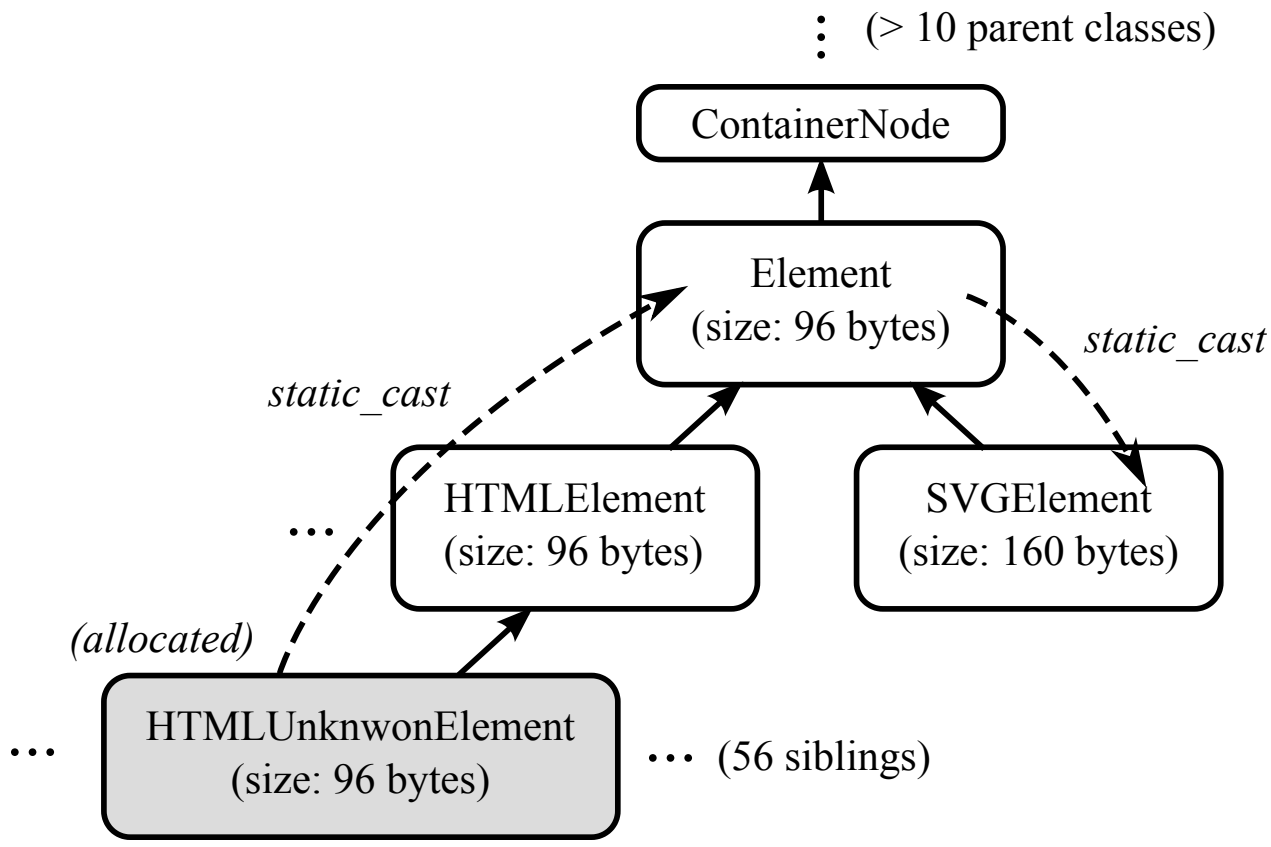



```
1  // (4) BUG: invalid downcast
2  SVGElement *p = static_cast<SVGElement*>(pDom);
3
4  // (5) leads to memory corruption!
5  p->m_className = "my-canvas";
```

Type Confusion: Bad Casting in C++

- Two (important) type castings:
    1. static_cast<>: verify the correctness at compilation
    2. dynamic_cast<>: verify the correctness at runtime
- Incorrect downcasting (via static_cast<>) results in memory violation
- dynamic_cast<> requires RTTI, incurring non-trivial perf. overheads

CVE-2013-0912 and Pwn2Own 2013



Exercise: Real-world Examples

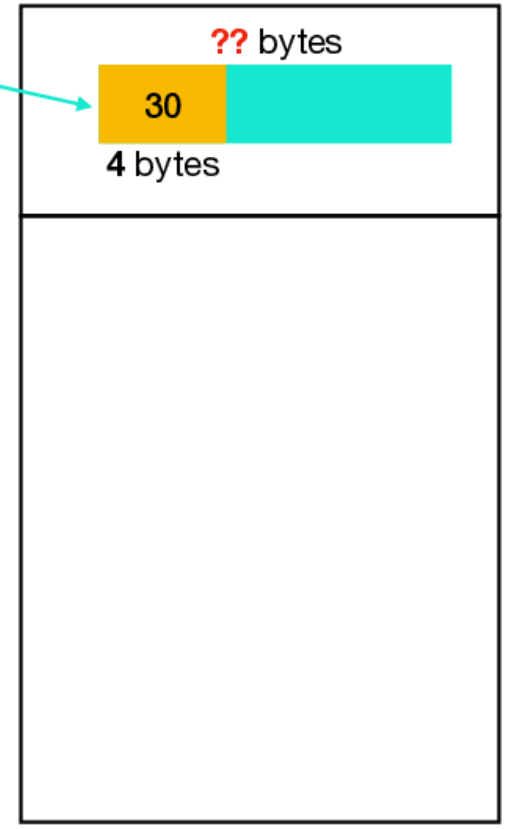
- Ex1. Linux Perf (CVE-2009-3234/+)
  - → Race condition: double fetching in Linux
- Ex2. Linux USB (CVE-2016-4482)
  - → Uninitialized read via struct padding

Double Fetching Vulnerabilities

- A special form of race conditions: user vs. kernel spaces
- Leading to information leaks, buffer overflows, etc.

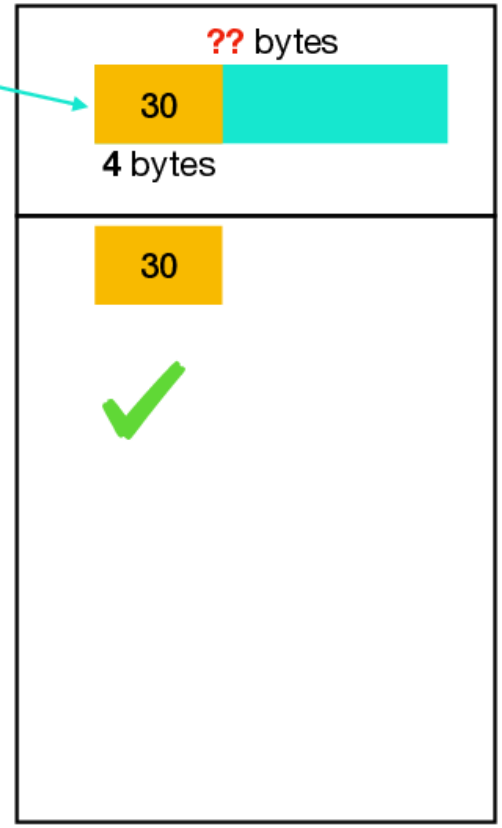
Example: perf_copy_attr() Explained

```
1 static int perf_copy_attr_simplified
2   (struct perf_event_attr __user *uattr,
3    struct perf_event_attr *attr) {
4
5   u32 size;
```



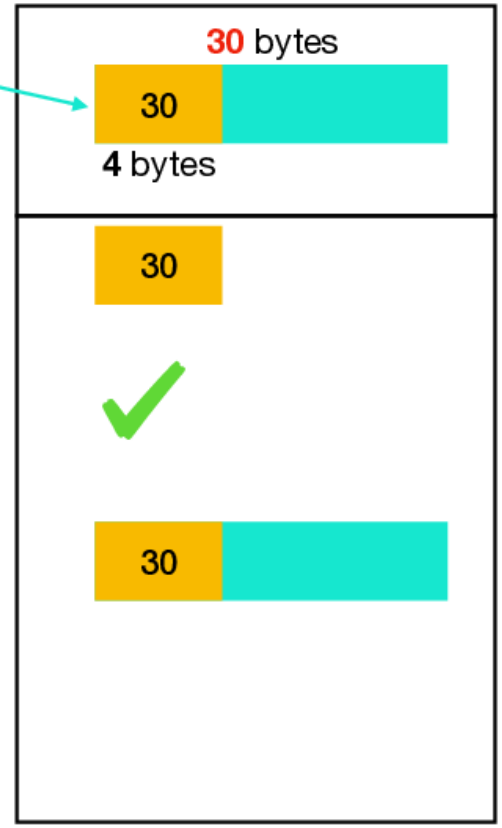
Example: perf_copy_attr() Explained

```
1  static int perf_copy_attr_simplified
2    (struct perf_event_attr __user *uattr,
3     struct perf_event_attr *attr) {
4
5    u32 size;
6
7    // first fetch
8    if (get_user(size, &uattr->size))
9      return -EFAULT;
10
11   // sanity checks
12   if (size > PAGE_SIZE ||
13       size < PERF_ATTR_SIZE_VER0)
14     return -EINVAL;
```



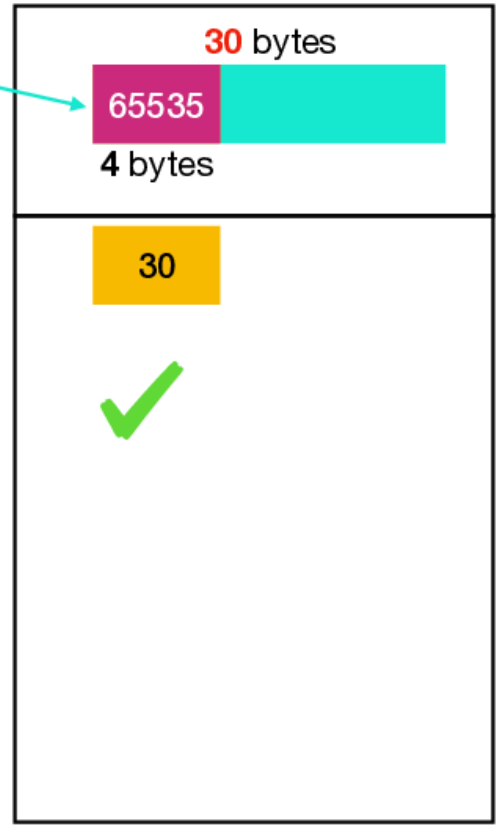
Example: perf_copy_attr() Explained

```
1  static int perf_copy_attr_simplified
2    (struct perf_event_attr __user *uattr,
3     struct perf_event_attr *attr) {
4
5    u32 size;
6
7    // first fetch
8    if (get_user(size, &uattr->size))
9      return -EFAULT;
10
11   // sanity checks
12   if (size > PAGE_SIZE ||
13       size < PERF_ATTR_SIZE_VER0)
14     return -EINVAL;
15
16   // second fetch
17   if (copy_from_user(attr, uattr, size))
18     return -EFAULT;
19
20   ......
21 }
```



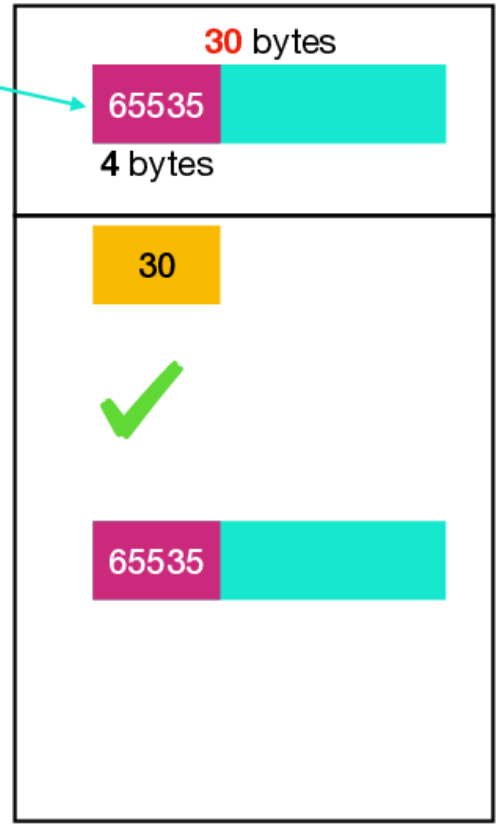
BUG: Racing in Userspace

```
1  static int perf_copy_attr_simplified
2    (struct perf_event_attr __user *uattr,
3     struct perf_event_attr *attr) {
4
5    u32 size;
6
7    // first fetch
8    if (get_user(size, &uattr->size))
9      return -EFAULT;
10
11   // sanity checks
12   if (size > PAGE_SIZE ||
13       size < PERF_ATTR_SIZE_VER0)
14     return -EINVAL;
```



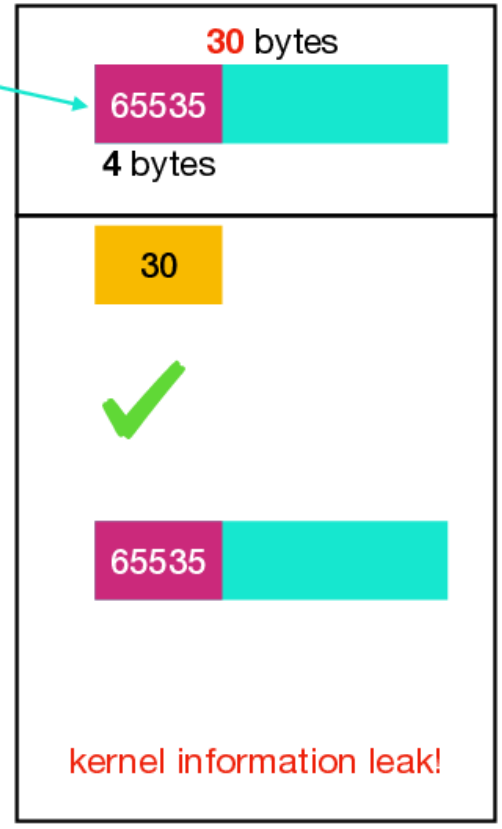
BUG: "Double" Fetching from Kernel

```
1  static int perf_copy_attr_simplified
2    (struct perf_event_attr __user *uattr,
3     struct perf_event_attr *attr) {
4
5    u32 size;
6
7    // first fetch
8    if (get_user(size, &uattr->size))
9      return -EFAULT;
10
11   // sanity checks
12   if (size > PAGE_SIZE ||
13       size < PERF_ATTR_SIZE_VER0)
14     return -EINVAL;
15
16   // second fetch
17   if (copy_from_user(attr, uattr, size))
18     return -EFAULT;
19
20   ......
21 }
```



BUG: Trigger Incorrect Memory Copy

```
1  static int perf_copy_attr_simplified
2    (struct perf_event_attr __user *uattr,
3     struct perf_event_attr *attr) {
4
5    u32 size;
6
7    // first fetch
8    if (get_user(size, &uattr->size))
9      return -EFAULT;
10
11   // sanity checks
12   if (size > PAGE_SIZE ||
13       size < PERF_ATTR_SIZE_VER0)
14     return -EINVAL;
15
16   // second fetch
17   if (copy_from_user(attr, uattr, size))
18     return -EFAULT;
19
20   ......
21 }
22
23 // BUG: when attr->size is used later
24 copy_to_user(ubuf, attr, attr->size);
```



Fixing Double Fetches

1. Partialy reading after the size attribute
2. Ensuring the atomicity of previous checked size

```
1    // @f12f42acdbb577a12eecfcebbbec41c81505c4dc
2    ret = get_user(size, &uattr->size);
3    ret = copy_from_user(attr, uattr, size);
4    ...
5
6    // overwrite with the sanitiy-checked size
7  + attr->size = size;
```

CVE-2009-3234: Buffer Overflow!

```
1   /* If we're handed a bigger struct than we know of,
2    * ensure all the unknown bits are 0.  */
3   if (size > sizeof(*attr)) {
4     ...
5     for (; addr < end; addr += sizeof(unsigned long)) {
6       ret = get_user(val, addr);
7       if (ret)
8         return ret;
9       if (val)
10        goto err_size;
11    }
12  }
13  ret = copy_from_user(attr, uattr, size); // Q. size?
```

CVE-2009-3234: Buffer Overflow!

```
1    /* If we're handed a bigger struct than we know of,
2     * ensure all the unknown bits are 0.  */
3    if (size > sizeof(*attr)) {
4      ...
5      for (; addr < end; addr += sizeof(unsigned long)) {
6        ret = get_user(val, addr);
7        if (ret)
8          return ret;
9        if (val)
10         goto err_size;
11     }
12 +   size = sizeof(*attr);
13   }
14   ret = copy_from_user(attr, uattr, size); // Q. size?
```

Exercise: Real-world Examples

- Ex1. Linux Perf (CVE-2009-3234/+)
  - → Race condition: double fetching in Linux
- Ex2. Linux USB (CVE-2016-4482)
  - → Uninitialized read via struct padding

CVE-2016-4482: Linux USB

```
1  static int proc_connectinfo(struct usb_dev_state *ps,
2                              void __user *arg) {
3    struct usbdevfs_connectinfo ci = {
4      .devnum = ps->dev->devnum,
5      .slow = ps->dev->speed == USB_SPEED_LOW
6    };
7
8    if (copy_to_user(arg, &ci, sizeof(ci)))
9      return -EFAULT;
10   return 0;
11 }
```

Padding Issues in Struct

```
struct usbdevfs_connectinfo {
  unsigned int devnum;  // 4 bytes
  unsigned char slow;   // 1 bytes
};

sizeof(struct usbdevfs_connectinfo) == 8
```

```
|<---------- 8B ---------->|
[devnum      ][slw][padding]
|<--  4B  -->|<1B>|<--3B-->|
```

Ref. Proactive Kernel Memory Initialization to Eliminate Data Leakages

Struct Padding: No Proper Way to Initialize

> *§6.2.6.1/6 (C11, ISO/IEC 9899:201x)*
>
> *When a value is stored in an object of structure (...), the bytes of the object representation that correspond to any padding values.*

```
struct usbdevfs_connectinfo ci = {
  .devnum = ps->dev->devnum,
  .slow = ps->dev->speed == USB_SPEED_LOW
};
```

CVE-2016-4482: Patch

- Zero-ing out via memset(): initializing two times!

```
1  static int proc_connectinfo(struct usb_dev_state *ps,
2                              void __user *arg) {
3  +   struct usbdevfs_connectinfo ci;
4  +
5  +   memset(&ci, 0, sizeof(ci));
6  +   ci.devnum = ps->dev->devnum;
7  +   ci.slow = ps->dev->speed == USB_SPEED_LOW;
8      ...
9  }
```

Summary

- Modern attacks are using **ROP** to defeat DEP/NX
- Learn three emerging, critical classes of vulnerabilities
  - **Type confusion**: e.g., bad casting in C++
  - **Race condition**: e.g., double fetching
  - **Uninitialized read**: e.g., struct padding issues

References

- CVE-2016-4482
- CVE-2013-0912
- CVE-2009-3234
- Proactive Kernel Memory Initialization to Eliminate Data Leakages
- Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels
- Return-oriented Programming
- Unisan: paper/slides