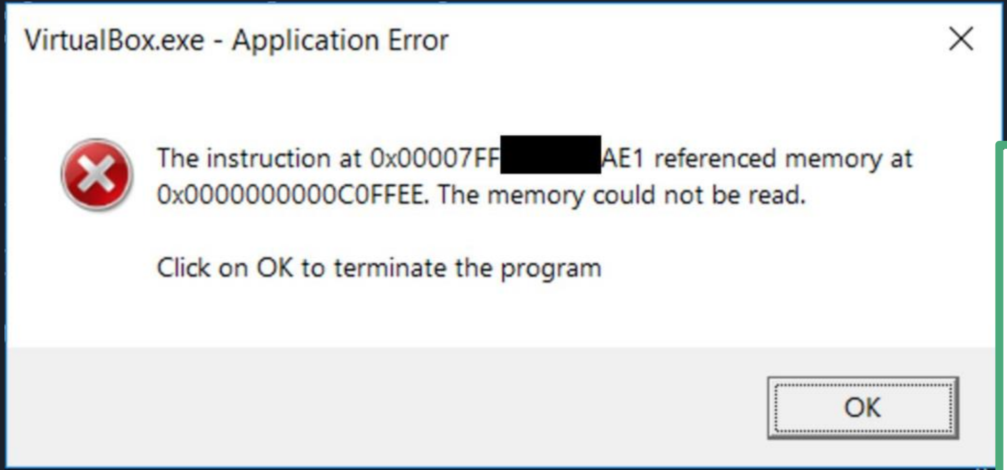
From UB to RCE

Memory Unsafety from an attacker's point of view

Niklas Baumstark

WHOIS

- Co-Founder @ CASHLINK GmbH
- Passionate about security research
  - Reported & exploited bugs in Safari, macOS, and VirtualBox
  - Pwn2Own ’17 & ’18
  - Capture-the-Flag player & orga with KITCTF and Eat Sleep Pwn Repeat
- Blogging about exploitation @ phoenhex.re
- Contact: @_niklasb on Twitter



Source: https://twitter.com/alisaesage

```
Stopped reason: SIGSEGV
0x00000000deadbeef in ?? ()
gdb-peda$ x/10i $rip
=> 0xdeadbeef:  Cannot access memory at address 0xdeadbeef
```

Why offensive security in software?

- To kill bugs
- To identify risky attack surface
- To evaluate the start of the art of exploitation
  - Learn about bug classes and techniques
  - Design effective mitigations that make exploitation harder

How to exploit <SOME_C(++)_SOFTWARE>

1. Find 1+ memory corruption bugs *reachable by the attacker*
2. Turn bugs into useful *exploit primitives*
3. Upgrade primitives
4. Overwrite a function pointer and forge some data structures
5. Hijack program control flow and get *arbitrary code execution*
6. Post-exploitation payload

Agenda

1. Common exploit primitives & root causes
2. Common exploit mitigations
3. How modern exploits work
4. Exploit demo

Common exploit primitives & root causes

Important exploit primitives

- Absolute memory read/write
- Relative read/write: Overflows, underflows and OOB
- Pointer type confusion

Absolute read/write

*ptr = data

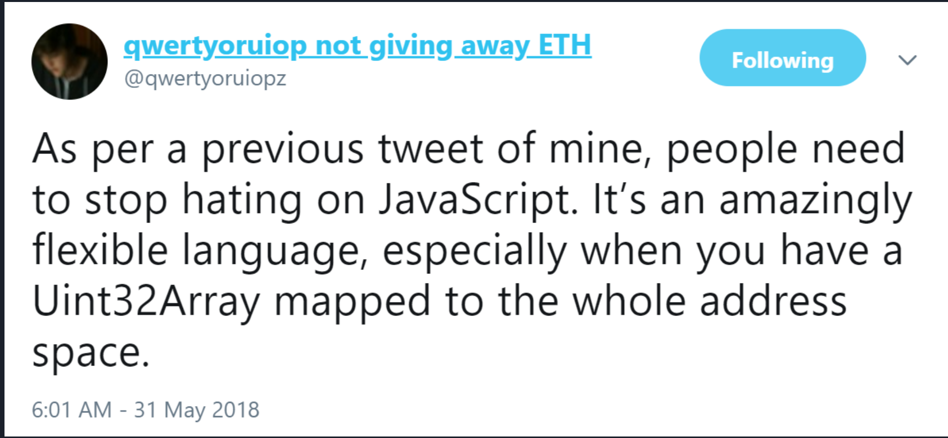
vec.push_back(x)

str += c

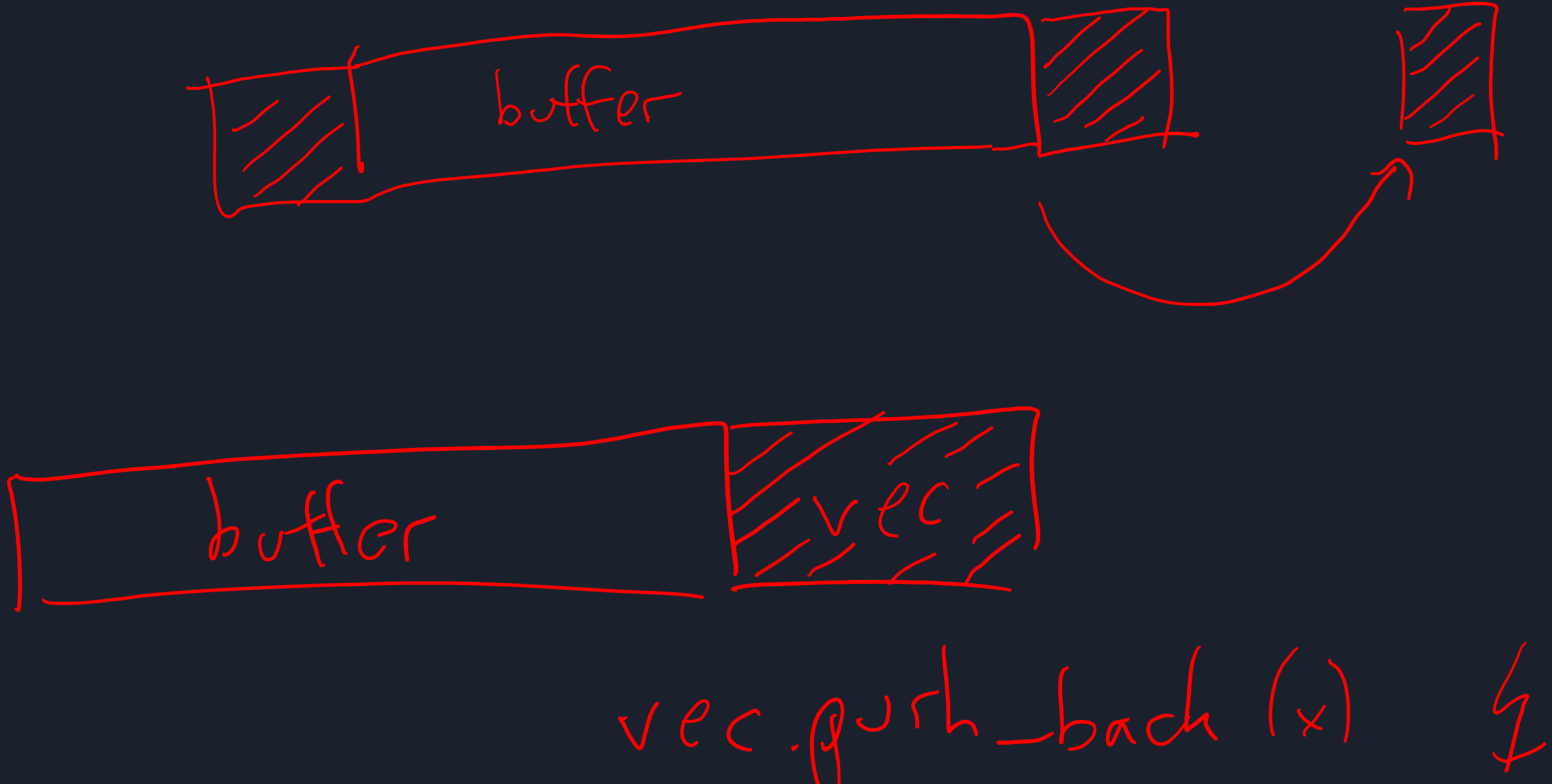
```
std::vector {
    char
    T* data
    T* end
    T* allocation_end
}
```

Absolute read/write

- We control a pointer that is used to read data given back to us
- We control a pointer that is used to write data controlled by us
- General rule of thumb: arbitrary read & write = GAME OVER



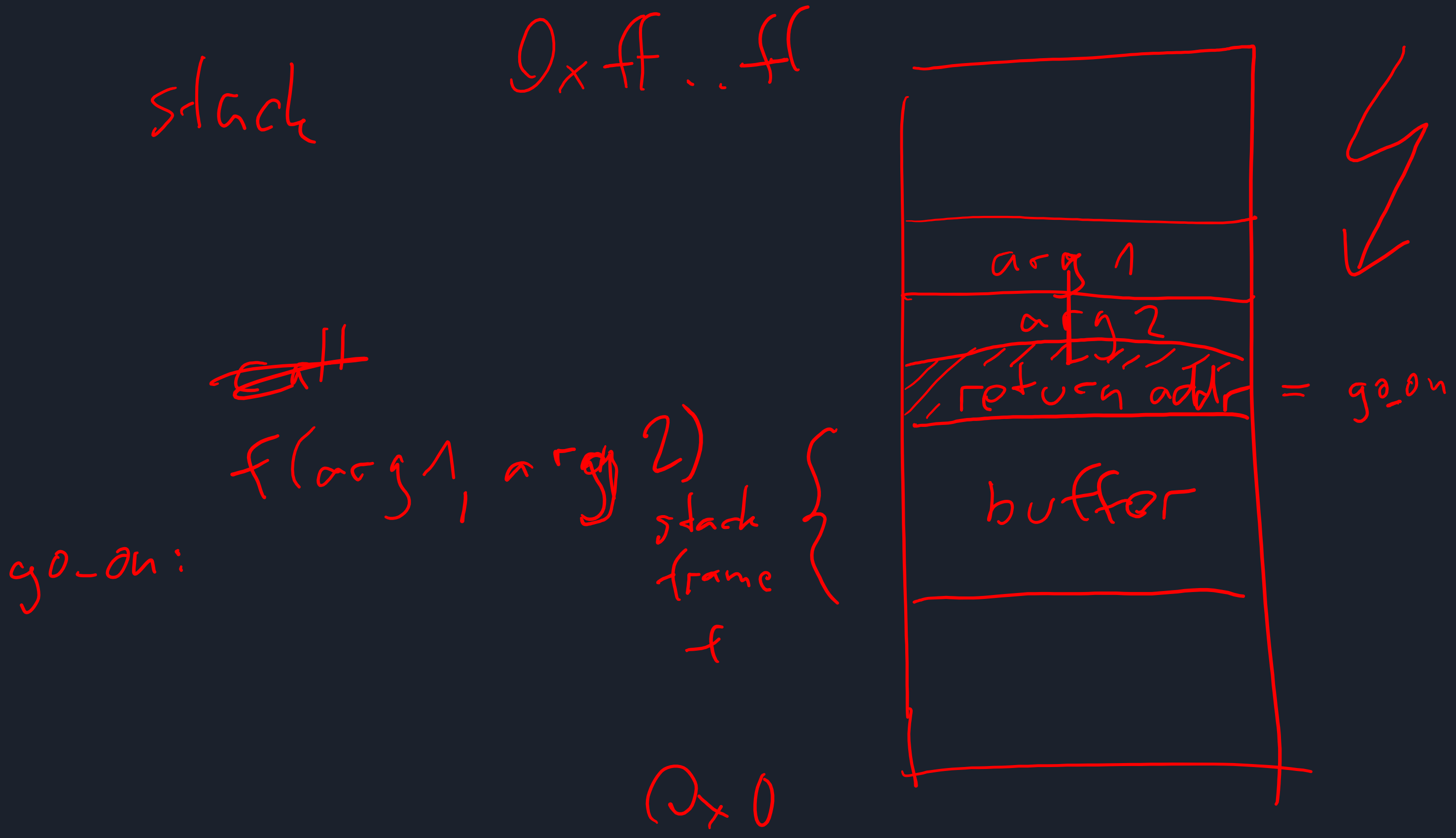
Relative read/write: Over/underflow, OOB



Relative read/write: Over/underflow, OOB

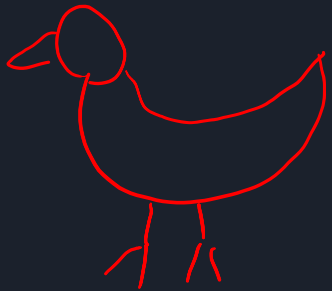
- Buffer bounds are not checked correctly
- Linear (buffer over/underflow) vs non-linear (at offset)
- Useful both as read or write
- Useful on the stack, heap or in global memory
- Often used to corrupt pointers, offsets or length fields

Relative read/write: Over/underflow, OOB



Type confusion

Dog* dog

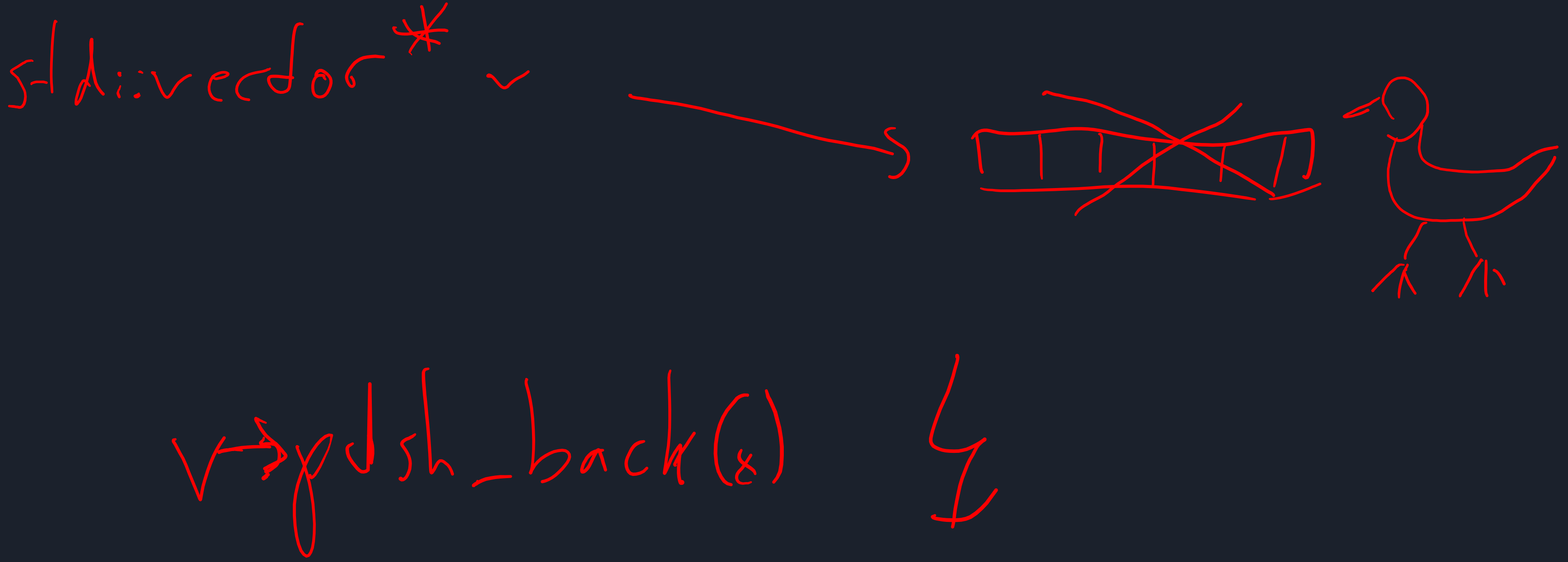


dog.bark(); ↯

Type confusion

- Pointer of type X* points to
  - Object of type Y != X
  - Some buffer containing unrelated data
  - The middle of some object
- We can often choose the pointed-to object from a set of possible types (ideally type is arbitrary)

Type confusion



Common root causes of memory corruption

- Missing bounds checking of untrusted input (sizes, offsets)
- Integer overflow/truncation
- Dangling pointers (use-after-free)
- Race conditions
- Uninitialized memory

Missing bounds checking of user input

When spotted in a code base, feels like

¯_(ツ)_/¯

When you miss one and somebody else pipes /dev/urandom into the program:

(╯°□°）╯︵ ┻━┻

- Occasional oversights or unaudited legacy code
- Mostly during parsing of complex binary data such as image files

Missing bounds checks: Example

```
static int vboxVDMACmdExecBpbTransfer(PVBOXVDMAHOST pVdma,
        const PVBOXVDMACMD_DMA_BPB_TRANSFER pTransfer, /*...*/)
{
    // ...
    uint32_t cbTransfer = pTransfer->cbTransferSize;
    uint32_t cbTransfered = 0;
    // ...
    do
    {
        uint32_t cbSubTransfer = cbTransfer;
        // ...
            pvSrc  = pvRam + pTransfer->Src.offVramBuf + cbTransfered;
        // ...
            pvDst  = pvRam + pTransfer->Dst.offVramBuf + cbTransfered;
        // ...
            memcpy(pvDst, pvSrc, cbSubTransfer);
            cbTransfer -= cbSubTransfer;
            cbTransfered += cbSubTransfer;
        // ...
    } while (cbTransfer);
    // ...
```

Integer overflows

- Integers are hard
  - Unsigned integer overflow (well-defined)
  - Signed integer overflow (UB)
  - Truncation / signedness issues during conversions & comparisons
- Can lead to
  - Unexpected negative offsets
  - Unexpected huge values (e.g. signed -> unsigned cast)
  - Incorrect length computations => overflow
- Static analysis helps a lot, but yields false positives
  - Turn on all the compiler warnings you can, early on

https://googleprojectzero.blogspot.de/2014/07/pwn4fun-spring-2014-safari-part-i_24.html

Integer overflows: Example

```
/* Code used when joining strings in WebKit Javascript, e.g.
["Integers", "are", "hard"].join(" ") == "Integers are hard"
*/

unsigned m_cumulatedStringsLength;
// [...]
inline void JSStringJoiner::append(const String& str)
{
    if (!m_isValid) return;
    m_strings.uncheckedAppend(str);
    if (!str.isNull()) {
        m_cumulatedStringsLength += str.length();
        // [...]
    }
}
// [...]
JSValue JSStringJoiner::build(ExecState* exec)
{
    // [...]
    size_t outputStringSize = totalSeparactorsLength + m_cumulatedStringsLength;
    // [...]
        // this uses outputStringSize to allocate the result buffer
        outputStringImpl = joinStrings<LChar>(m_strings, m_separator, outputStringSize);
    // [...]
```

https://googleprojectzero.blogspot.de/2014/07/pwn4fun-spring-2014-safari-part-i_24.html

Integer overflows: Example

```
var longString = "A".repeat(0x10000)
var longAry = []
for (var i = 0; i < 0x10001; ++i)
    longAry.push(longString)
longAry.join("")

/*
0x10000'th iteration: m_cumulatedStringsLength = 0xffff0000 + 0x10000 = 0
0x10001'th iteration: m_cumulatedStringsLength = 0          + 0x10000 = 0x10000

*Slightly* too small to hold 4 GB of strings.
*/
```

https://googleprojectzero.blogspot.de/2014/07/pwn4fun-spring-2014-safari-part-i_24.html

Integer overflows: Example

```
Checked<unsigned, RecordOverflow> m_accumulatedStringsLength;
// [...]
JSValue JSStringJoiner::build(ExecState* exec)
{
    // [...]
    Checked<size_t, RecordOverflow> totalSeparactorsLength = /*...*/;
    Checked<size_t, RecordOverflow> outputStringSize = totalSeparactorsLength + m_accumulatedStringsLength;

    size_t finalSize;
    if (outputStringSize.safeGet(finalSize) == CheckedState::DidOverflow)
        return throwOutOfMemoryError(exec);

    // [...]
        outputStringImpl = joinStrings<LChar>(m_strings, m_separator, finalSize);
    // [...]
}

template</*...*/>
static inline PassRefPtr<StringImpl> joinStrings(
    const Vector<String>& strings, const String& separator,
    unsigned outputLength)
```

Use-after-Free

- Pointer is dereferenced after pointed-to object has been deleted
- Special case: Double free
- Mostly occurs when raw pointers / iterators are stored in memory

the grugq
@thegrugq

Following

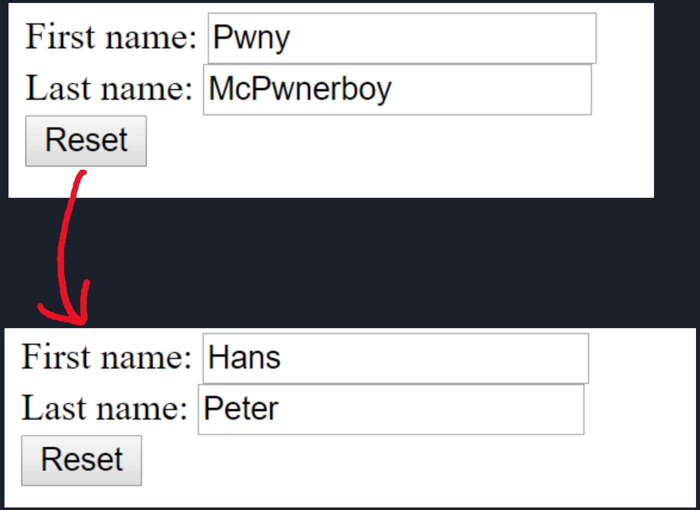
WebKit is basically a collection of use-after-frees that somehow manages to render HTML (probably via a buffer overflow in WebGL)

6:34 PM - 29 Apr 2014

https://bugs.chromium.org/p/project-zero/issues/detail?id=1044

Use-after-Free: Example

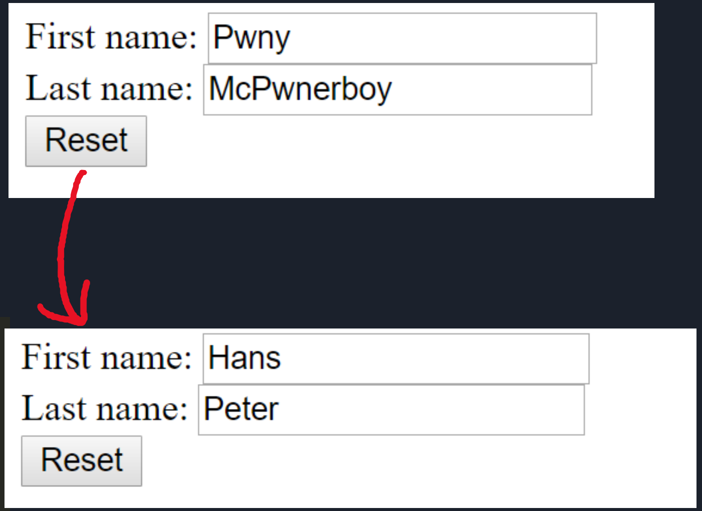
```
<form id="form" onchange="eventhandler()">
First name: <input type="text" value="Hans"><br>
Last name: <input type="text" value="Peter"><br>
<button onclick="form.reset()">Reset</button>
</form>
```



https://bugs.chromium.org/p/project-zero/issues/detail?id=1044

Use-after-Free: Example

```html
<form id="form" onchange="eventhandler()">
First name: <input type="text" value="Hans"><br>
Last name: <input type="text" value="Peter"><br>
<button onclick="form.reset()">Reset</button>
</form>
```




```cpp
void HTMLFormElement::reset()
{
    // [...]
    for (auto& associatedElement : m_associatedElements) {
        if (is<HTMLFormControlElement>(*associatedElement))
            downcast<HTMLFormControlElement>(*associatedElement).reset();
    }
    // [...]
}
```

https://bugs.chromium.org/p/project-zero/issues/detail?id=1044

Use-after-Free: Example

```
<form id="form">
    <output id="output">
        <div id="inner"></div>
    </output>
    <button onclick="form.reset()"></button>
</form>
```

Inside `HTMLOutputElement::reset`, DOM will be modified:
inner `<div>` will be removed

https://bugs.chromium.org/p/project-zero/issues/detail?id=1044

Use-after-Free: Example

```
<script>
function go() {
  output.addEventListener('DOMSubtreeModified', function () {
    for (var i = 0; i < 100; i++)
      form.appendChild(document.createElement("input"));
  }, false);

  form.reset();
}
</script>

<form id="form">
    <output id="output">
        <div id="inner"></div>
    </output>
    <button onclick="go()"></button>
</form>
```

https://bugs.chromium.org/p/project-zero/issues/detail?id=1044

Use-after-Free: Example

```
void HTMLFormElement::reset()
{
    // [...]
    for (auto& associatedElement : m_associatedElements) {
        if (is<HTMLFormControlElement>(*associatedElement))
            downcast<HTMLFormControlElement>(*associatedElement).reset();
    }
    // [...]
}
```

Race condition

- Data structure is modified by one thread and can be observed in inconsistent state by a separate thread
- Special case: *double fetch* of memory from a less privileged context
  - Usually in operating systems or hypervisors
  - Enables time-of-check vs. time-of-use (TOCTOU) errors

```
int syscall_handler(struct* user_arg) {
    char buf[1024];
    if (user_arg->length <= 1024) {
        memcpy(buf, user_arg->buffer, user_arg->length);
    }
}
```

Double fetch: Example

```
static int vboxVDMACmdExec(PVBOXVDMAHOST pVdma, const uint8_t *pvBuffer, uint32_t cbBuffer)
{
    // ...
        PVBOXVDMACMD pCmd = (PVBOXVDMACMD)pvBuffer;
        switch (pCmd->enmType)
        {
            case VBOXVDMACMD_TYPE_CHROMIUM_CMD:
                /* do something */
            case VBOXVDMACMD_TYPE_DMA_PRESENT_BLT:
                /* do something */
            case VBOXVDMACMD_TYPE_DMA_BPB_TRANSFER:
            // ... some more cases
        }
    // ...
}
```

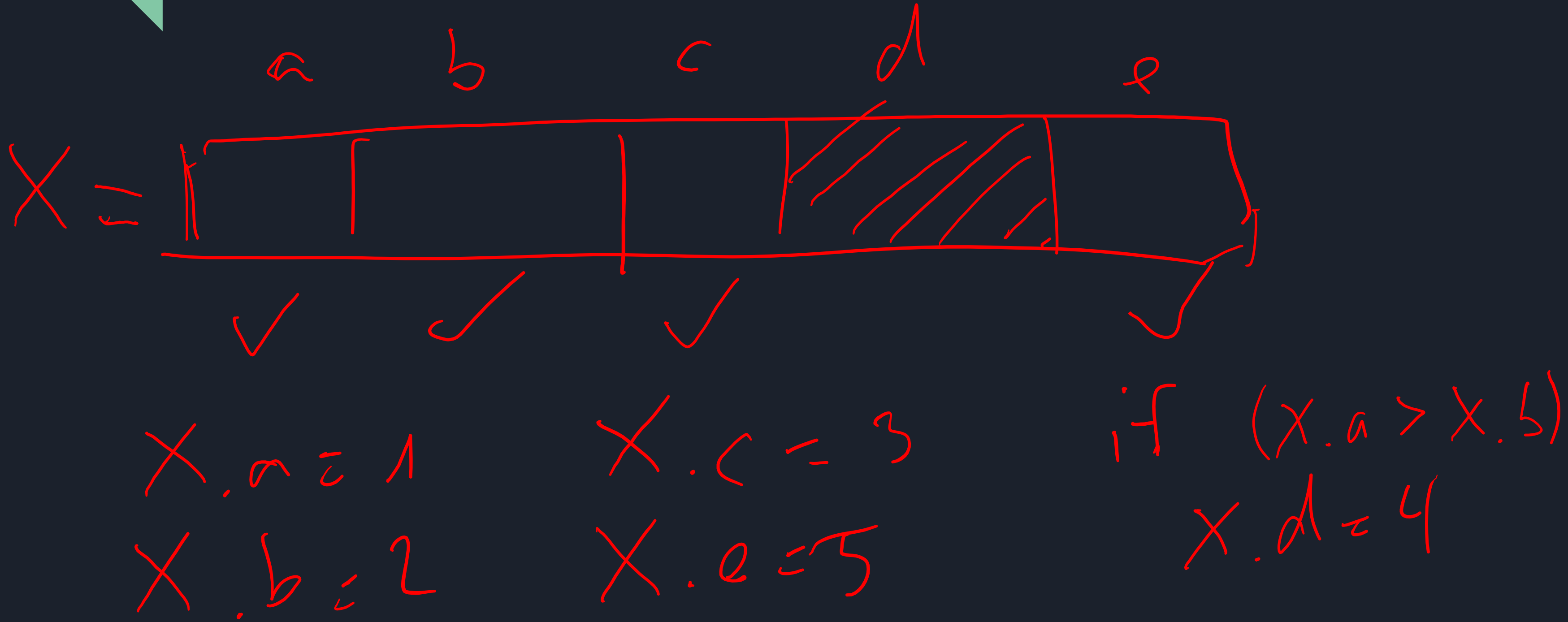
Double fetch: Example

```
cmp     dword ptr [rbx], 0Ah ; switch 11 cases
ja      invalid_offset   ; jumptable 00000000001144EE
mov     eax, [rbx]
lea     rcx, jump_table
movsxd  rax, ds:(jump_table - 2C9C38h)[rcx+rax*4]
add     rax, rcx
jmp     rax              ; switch jump
```

Uninitialized memory

- Object or buffer is left partially uninitialized before usage
- Often leads to other primitives

Uninitialized memory



Common exploit mitigations

Exploit mitigations in 2018

- Address Space Layout Randomization (ASLR)
- No eXecute (NX)
- Stack canaries
- Code Flow Integrity (CFI)
- Heap metadata hardening

Address Space Layout Randomization (ASLR)

```
int x;
std::cout
    << "code @ "  << (void*)&main << " | "
    << "stack @ " << &x << " | "
    << "heap @ "  << (void*)new char[10] << "\n\n";
sendfile(1, open("/proc/self/maps", 0), 0, 0x10000);
```

Address Space Layout Randomization (ASLR)

```
$ ./a.out
code @ 0x1397e7859ba | stack @ 0x7dc39968ae94 | heap @ 0x139a4699cf0

1397e785000-1397e786000 r-xp 00000000 fe:02 39354511                   /home/niklas/ub_to_rce/aslr/a.out
...
139a4687000-139a46a9000 rw-p 00000000 00:00 0                          [heap]
...
6ae510776000-6ae5108ee000 r-xp 00000000 00:16 12105392                 /usr/lib/libstdc++.so.6.0.24
...
7dc39966c000-7dc39968e000 rw-p 00000000 00:00 0                        [stack]

$ ./a.out
code @ 0x9b3726489ba | stack @ 0x7793ca3e8074 | heap @ 0x9b373a09960

9b372648000-9b372649000 r-xp 00000000 fe:02 39354511                   /home/niklas/ub_to_rce/aslr/a.out
...
9b3739f7000-9b373a19000 rw-p 00000000 00:00 0                          [heap]
...
67080a5e7000-67080a75f000 r-xp 00000000 00:16 12105392                 /usr/lib/libstdc++.so.6.0.24
...
7793ca3c8000-7793ca3ea000 rw-p 00000000 00:00 0                        [stack]
```

No eXecute

- CPU-enforced security mechanism
- Every virtual memory page has read, write and execute bits
- Data (stack, heap, global memory) is **RW-**
- Code is **R-X**

⇒ Can't overwrite code, or execute data[1]

[1] Except when there is RWX memory

No eXecute

```c
uint8_t shellcode[] = "\xcc\xcc\xcc";  // cc = int3 instruction = software breakpoint
((void(*)())shellcode)();
```

```
Stopped reason: SIGSEGV
0x00007fffffffcf74 in ?? ()
gdb-peda$ x/3i $rip
=> 0x7fffffffcf74:      int3
   0x7fffffffcf75:      int3
   0x7fffffffcf76:      int3
gdb-peda$ vmmap 0x00007fffffffcf74
Start              End                Perm      Name
0x00007ffffffdd000 0x00007ffffffff000 rw-p      [stack]
```

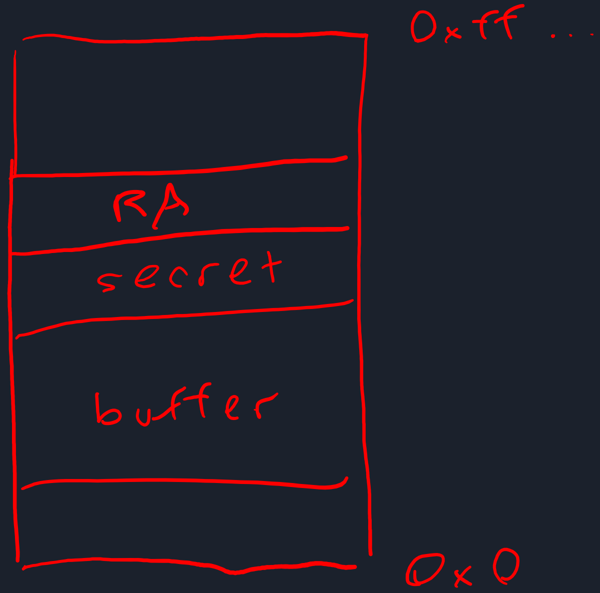
No eXecute

```c
uint8_t shellcode[] = "\xcc\xcc\xcc";  // cc = int3 instruction = software breakpoint
mprotect((void*)(((uintptr_t)shellcode) & ~0xfff), 0x2000, 7);  // 7 = RWX
((void(*)())shellcode)();
```

```
Stopped reason: SIGTRAP
0x00007ffffffd4f5 in ?? ()
gdb-peda$ x/3i $rip
=> 0x7ffffffd4f5:        int3
   0x7ffffffd4f6:        int3
   0x7ffffffd4f7:        add    BYTE PTR [rax],al
gdb-peda$ vmmap 0x7ffffffd4f5
Start              End                Perm      Name
0x00007ffffffd000 0x00007ffffffff000 rwxp      [stack]
```

Stack canaries

- Secret value placed between stack buffers and return address
- Checked before return



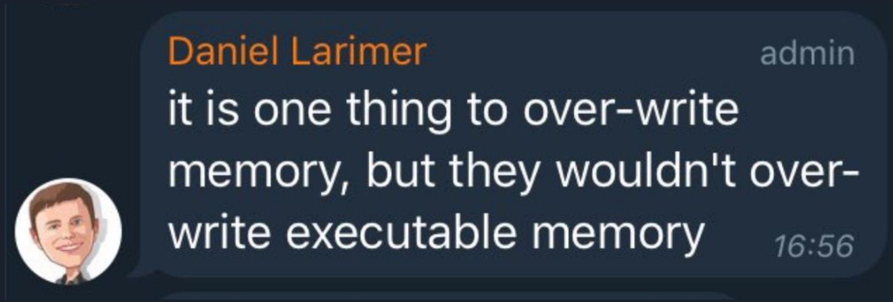
Modern exploit basics

Defeating ASLR & canaries: infoleaks

- Mitigations based on secret values
- In most cases, requires *info leaks* for exploitation
  - Exploit needs to incorporate the secrets
  - Usually, via interaction between target and exploit
  - In rare cases, file format parsers can be tricked to do perform non-trivial computation
- Can require additional bugs, or exploiting bugs multiple times in different ways

Defeating NX: Code reuse

- In non-trivial binaries, useful code exists and can be used
- A single function call is usually enough (`mprotect` / `VirtualProtect` / `system` / `longjmp`)
- Common technique: *Return oriented programming (ROP)*



Heap-based exploitation

- Typical heap-based primitives:
  - Overflows
  - OOB writes
  - Use after free
  - Double free
- Exploitation requires predicting allocation patterns
  - Info leaks
  - Deterministic allocator behaviour

Heap spraying

- Works well for deterministic allocators
- Force allocator into predictable behaviour by “spraying” objects
- Make holes to get allocation in predictable places

Corrupting heap metadata

- Many allocators store metadata in between chunks (e.g. sizes)
- Corrupting those can lead to other primitives such as overlapping allocations or completely controlled allocations
- Can get arbitrarily complex [1]
- Some allocators try to protect against tampering using secrets
    - e.g. Windows *Low fragmentation heap*

[1] https://github.com/how2heap

Vtables

- C++ supports virtual method calls through polymorphic pointers
- Implementation via virtual tables

```
object->method(arguments)
```

Becomes

```
object->p_vtable->p_method(object, arguments)
```

The prototypical C++ exploit

- info leak
- Optional: arbitrary read/write
- Control flow hijack
  - Function pointer
  - Vtable pointer
  - Stack
  - RWX memory
- Make payload executable
- Jump into payload

Demo time!

Exploitability Quiz

Off-by-one heap overflow

```
(new char[x])[x] = y
(new char[x])[x] = 0
```

Off-by-one heap overflow

```
(new char[x])[x] = y
(new char[x])[x] = 0
```

- ChromeOS sandbox escape via shill TCP proxy (https://crbug.com/648971)
- RCE in Exim mail server (CVE-2018-6789)
- Off-by-one NULL byte in glibc `iconv_open` (CVE-2014-5119)

Memory / reference count leak

```
x->incrementRefcount();
// no decrement, e.g. because of error condition
```

Memory / reference count leak

```
x->incrementRefcount();
// no decrement, e.g. because of error condition
```

- Can be exploitable if refcounts are not checked for overflow
- Firefox: refcount leak -> 32-bit overflow -> use after free -> RCE

  `https://phoenhex.re/2017-06-21/firefox-structuredclone-refleak`

Double free without intermediate code

```
x->decrementRefcount();
x->decrementRefcount();
```

Double free without intermediate code

```
x->decrementRefcount();
x->decrementRefcount();
```

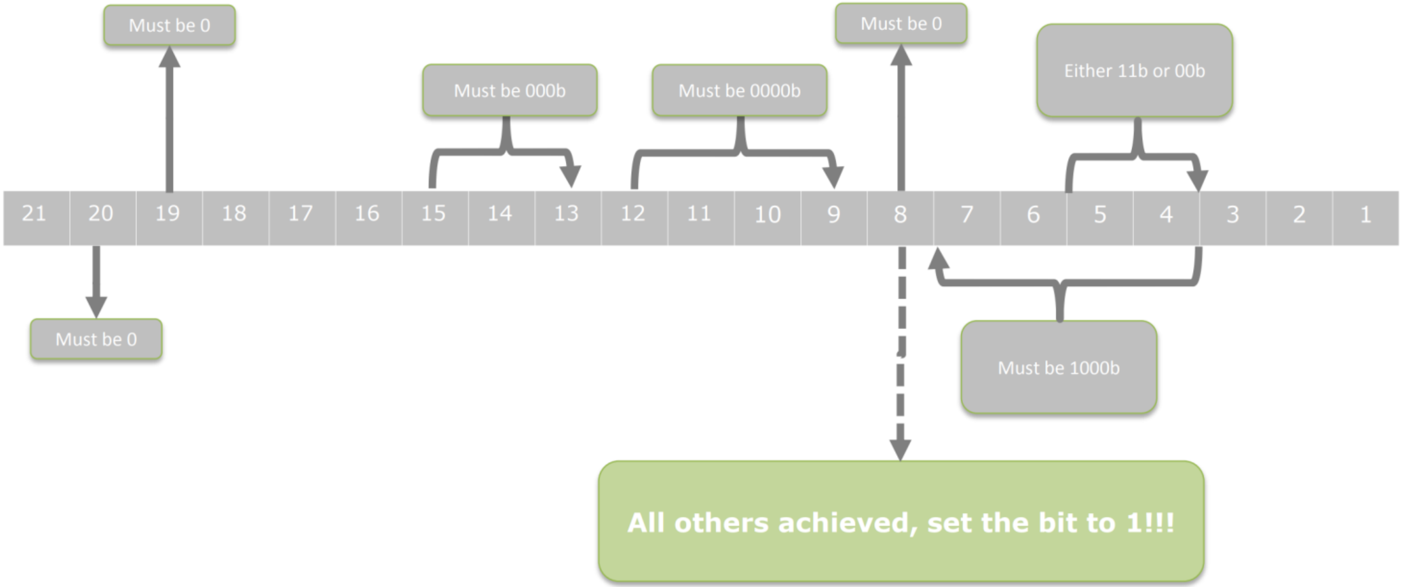
- Can be exploitable if attacker can interleave an allocation in separate thread
- CVE-2016-1804: macOS sandbox escape via WindowServer

Heap out-of-bounds 1-bit write

```
uint64_t* x = new uint64_t[10];
size_t idx = <attacker value>;
if (some complex condition on x[idx]) {
     x[idx] |= 0x40;
}
```

Restrictive 1-bit write

- 1-bit write can be reached when…



KEEN

Source: https://cansecwest.com/slides/2015/Liang_CanSecWest2015.pdf

NULL pointer dereference

```
X* x = nullptr;
x->some_struct->y = 0x1337;
```

NULL pointer dereference

```
X* x = nullptr;
x->some_struct->y = 0x1337;
```

- Exploitable if the attacker can map the zero page
- Possible in almost every OS until a few years ago
- Still possible in Windows 7
- Still possible if x is small but non-zero (i.e. offset provided into nullptr)

Thank you!

Time for questions :)