Chrome

Stuart Nevans Locke

Question to the audience:

What do you know about browsers and browser exploitation?

Disclaimer

- I am very far from an expert in this area
  - I probably will not know the answer to at least some questions

Overview

- Browser Security
  - An overview
    - Sandboxing
  - Javascript Engines
- V8
  - Dynamic Typing
  - Type Speculation
  - More type stuff!
  - TurboFan’s Sea of Nodes
  - Optimization Phases
  - Ranges

Browser Security

- Browsers are huge pieces of software
  - Chrome is made up of ~24 million LOC
- Pretty much all input they take is untrusted
  - Browsers basically spend all their time processing potentially malicious data
    - And users want them to be really fast
- We’re not going to be talking about XSS, but actually pwning the browser.
  - With a browser exploit, imagine how the impact of xss goes up.

Sandboxing

- Exploits just kept coming out unrelentlessly
  - Browser developers realized that they needed a way to reduce the number of important exploits
- Enter: Sandboxing
  - Browsers have a bunch of subprocesses
    - Often 1 per tab
  - These are things like the HTML renderer and the javascript engine.
- The HTML renderer and javascript engine are sandboxed
  - Often grouped into “renderers”

Sandboxed Processes

- What exactly can these sandboxed processes do?
  - Pretty much nothing
    - No file access
    - No display access
    - No keyboard access
- What can they do?
  - Make (some) syscalls
    - If you have a kernel exploit, you can chain it with a browser exploit
- How do they communicate with the rest of the browser?
  - IPC
    - Also a good place to find vulns

Javascript Engines

- A Javascript Engine is what executes javascript
- These things are also incredibly complex
  - Chrome's engine is v8
  - V8 is ~1.6 million LOC
- Because people are now writing web apps, javascript needs to run very fast
  - That means these engines need to be really optimized
- Often contain interpreters and JITs
  - Interpreter - A program that directly executes javascript code
  - JIT - just in time compiler
    - Compiles javascript to machine code, runs that
- Some engines actually have more than one JIT
  - Differing amounts of time spent optimizing based on how many times code is run

v8

- Because I know pretty much nothing about any engines other than v8, we'll be talking about v8 specifically
- Much of this stuff (should) apply to other engines

v8

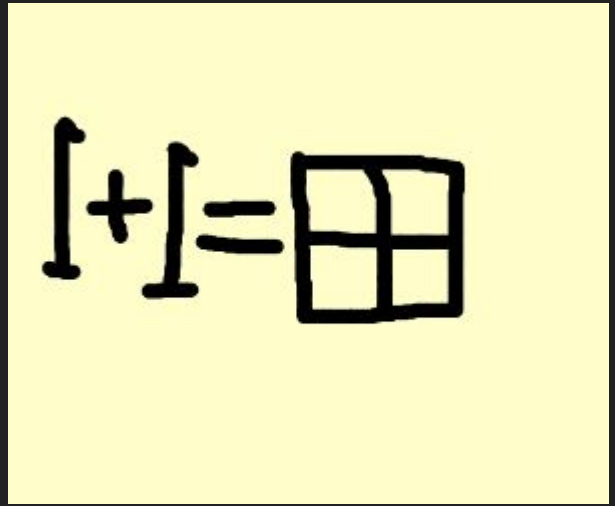
- V8 has an interpreter and a JIT
  - Interpreter runs first
  - If code is “hot”, the JIT runs
    - V8’s JIT is TurboFan

Dynamic Typing Issues

- Javascript is a dynamically typed language
- This means we don’t have type information at compile time
  - Type information is incredibly important to optimization
- Take the function that is def add(a,b){return a+b;}
  - In javascript, we could pass any objects to this function

Add function

- If we knew that add was only passed integers, we could optimize it to ~1 assembly instruction
  - add reg1, reg2
- If any objects could be passed, it will probably be 100s of instructions if not more.
  - Slower to run
  - Uses more memory



12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ? ToString(*lprim*).
    b. Let *rstr* be ? ToString(*rprim*).
    c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

Type speculation

- The solution to this is to guess (speculate) on the types of the objects passed
- V8 just guesses that integers would be passed to the function
- The way this speculation is done is by looking at historical data
  - When a function is interpreted, type information is collected
  - If add was called only with integers while being interpreted, TurboFan would compile it to expect integers
- This can vastly speed up code

Deoptimization

- Sometimes we speculate wrong
  - Maybe a function is just *usually* called with integers
  - If v8 guesses wrong it “deoptimizes”
- Deoptimization means it basically throws away it’s speculated type information and just makes a generic function

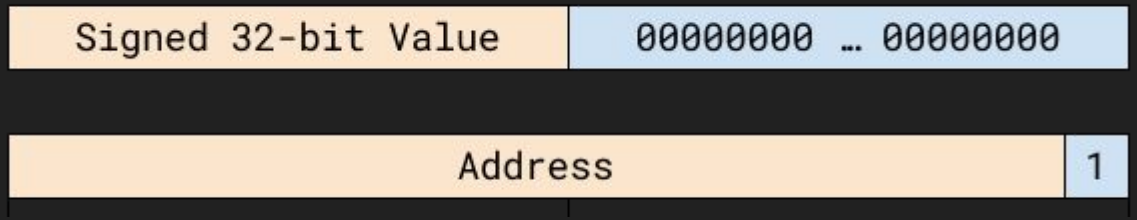
Demo time

Type Information

- I don’t know the official name of this, so I might refer to it as a “shape”
- In javascript, objects have some type
  - The type contains information such as where to find property names
  - For example, if some object “obj” has a field named “x”, it would have an entry in the shape saying where to find obj.x
- Typically these shapes are immutable and global
  - New ones are made if an object is given new properties or old properties are removed
  - Often shapes are shared by many objects

Pointer Tagging

- Note: V8 specific
- In v8, there are Small Integers (SMI) and Objects
- Smi’s are <32 bit integers
  - Weirdly, the value is stored in the upper 32 bits of a 64 bit value.
  - Meaning 0x**1234**00000000 has a value of 0x**1234**
- Everything else is an object, meaning it is accessed via a pointer
  - All pointers end in 1.
  - The value **0xff00018000**1 would be a pointer to **0xff00018000**1



https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8

Back to the add function

```
# Check x is a small integer
movq rax, x
test al,0x1
jnz Deoptimize
# Check y is a small integer
movq rbx, y
testb rbx,0x1
jnz Deoptimize
```

Check integers

```
# Convert y from Smi to Word32
movq rdx,rbx
shrq rdx, 32
# Convert x from Smi to Word32
movq rcx,rax
shrq rcx, 32
```

Get values of parameters

```
add rdx, rcx
shl rcx, 32
```

Actually add and save value

TurboFan’s Sea of Nodes

- When doing compilation, TurboFan breaks the code up into a “sea of nodes”
- Basically combined CFG and DFG
  - Control Flow Graph
  - Data Flow Graph
- It’s pretty much just a graph
  - There are a bunch of lines, generally you can just look at it to figure out what’s going on
- Turbolizer
  - Visualizes the above
  - Incredibly useful

Demo Time

Optimization Phases

- There are a ton of these
  - The vulnerability I want to exploit is in the Typer
  - Within the last month, chrome actually put out a commit meant to reduce vulnerabilities in the typer
- Typer
  - Associate types with a node
  - For example, PlainNumber is a type
- I’ll probably talk more about these later (not today)

Ranges

- We still need to optimize more!
- Just knowing the type of number (e.g. float vs int) isn’t enough to do all the optimizations we want
- If we know the Range of a number, we can do even more.
- So TurboFan has a Range type that determines the possible range a number can be within.
- Typically if you can generate an incorrect Range you have a vulnerability

Questions?