DOM Jungle
Can We Trust The UI?

Gal Weizman

Client side security (Web)

- Many types of attacks
  - Dom Clobbering
  - XSLeaks
  - CSP Bypass
  - Prototype Pollution
  - CSS Injection
  - XSS
  - Supply Chain attacks
  - More…

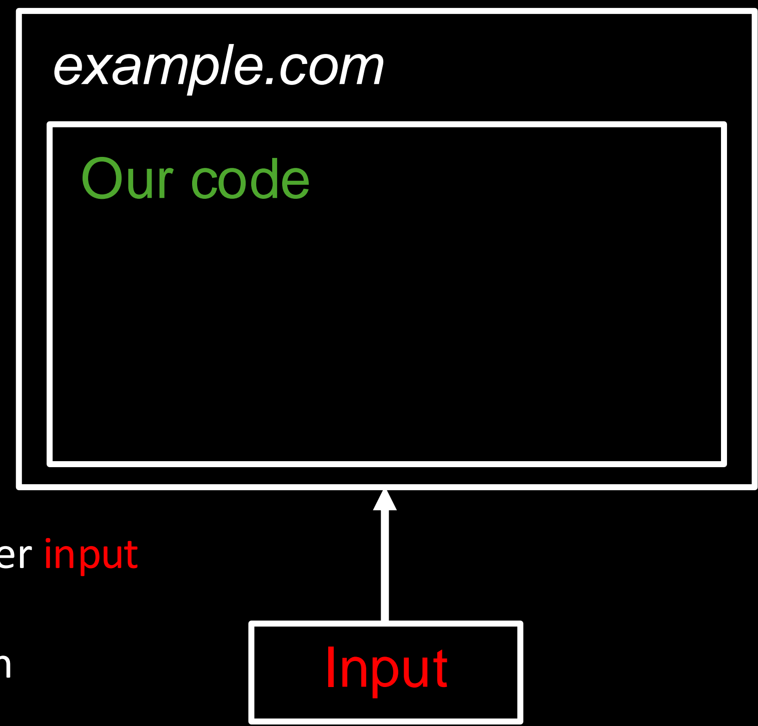
Code Execution

Breach of Origin - Implications

Evolution of XSS (the problem)

- Usability
  - User input
  - Jack
- Usability *x* Security
  - (code execution)
- Introduction of XSS
  -
    Jack <script> alert(“XSS”) </script>

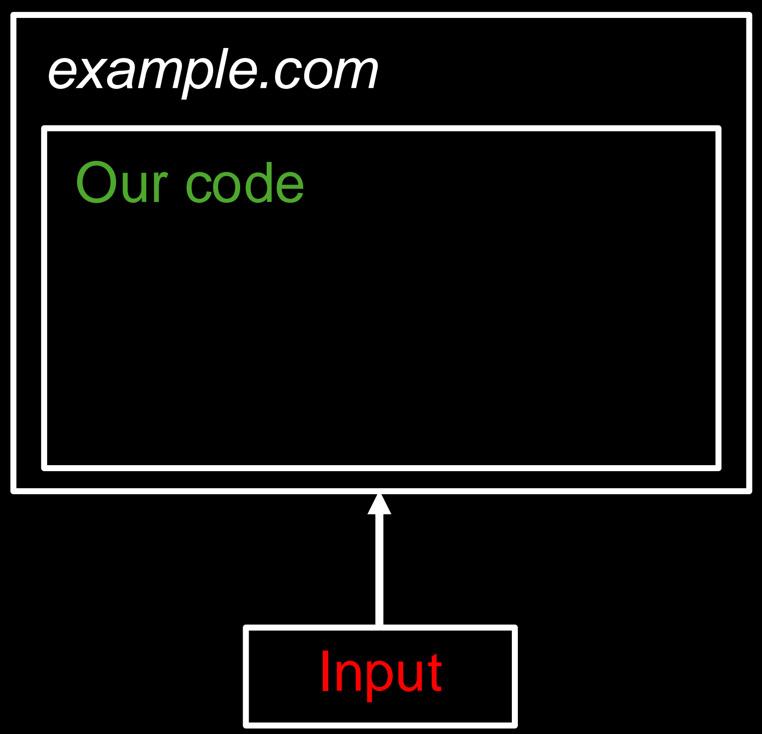
- An “outside” attack
  - We trust our app’s code - we don’t trust user input
- XSS
  - = Code that should not execute in our origin
- Web security had to catch up



Evolution of XSS (the solution)

- The Web fixes XSS!
- Sanitization
  - DOMPurify, Trusted Types, etc
- + CSP
  - script-src, strict-dynamic
  - unsafe-eval, unsafe-inline
- = No XSS

- Web security caught up
  - Web builders didn’t
  - Their problem ¯_(ツ)_/¯



2023's landings

With that introduction, we can now talk about some of what we've accomplished in the last year of security rollouts at Google! This is a small subset of the security rollouts we've done in the past year, focusing just on web security features that landed:

- **166** services are newly enforcing Strict CSP
- **176** services are newly enforcing Trusted Types
- **347** services are newly enforcing Resource Isolation Policy
- **1079** JavaScript builds have new static guarantees that all transitive dependencies satisfy our safe coding conformance checks
- **438** legacy DOM XSS sink assignments were removed
- **160** services are newly enforcing SameSite cookies

https://bughunters.google.com/blog/5896512897417216/a-recipe-for-scaling-security

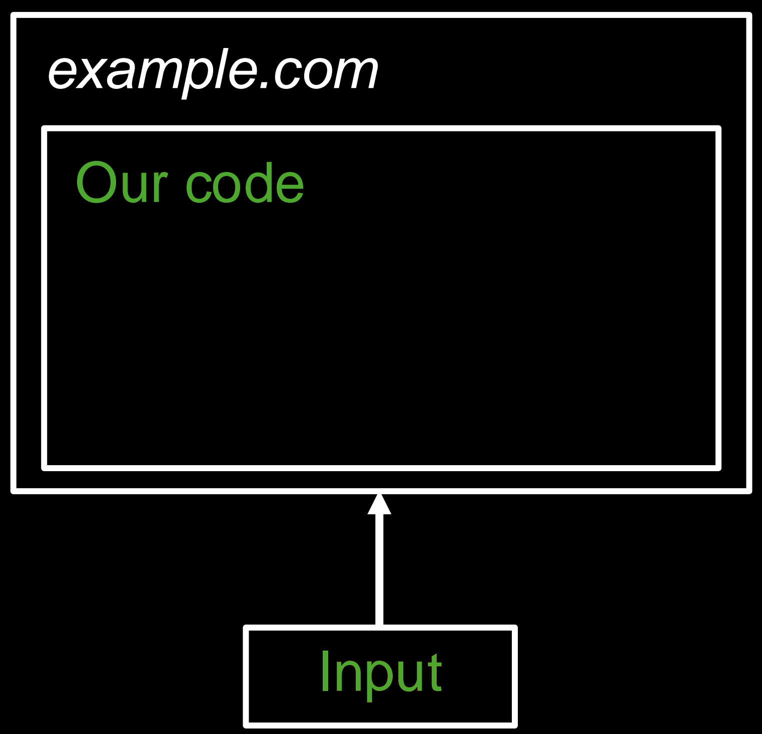
Some more napkin math comes up with an estimate that these 6 rollouts alone would have taken Google at least 20 engineering years of work if done in the traditional manner. By having the security team take on these rollouts centrally, we were able to land all these security features with a tiny fraction of that effort. This increases velocity for both the security team (leading to more secure products!) and for product teams (enabling product teams to ship better products!) and is a huge win for the user. All of this demonstrates what we see as **the full *safe coding* approach, where we uplift security at scale, for everyone**. To understand the full impact of this, we can also look at our overall security feature coverage, where we can see amazing statistics such as 96% of our most sensitive services enforcing CSP, and 80% of our most sensitive services enforcing Trusted Types. Across hundreds of web apps built on top of our secure frameworks that comprehensively enable these and other security mechanisms (such as safe server-side HTML templating), we have seen zero XSS vulnerabilities since the features were enforced.

If you're interested in a more detailed description of what this sort of project looks like end-to-end, check out this blog post on fixing debug log leakage with Safe Coding. This highlights many of the complexities inherent to doing large-scale rollouts, but is also a great story of a massively scalable security improvement across Google. Also, watch out for future blog posts in this series where we'll talk more about these rollouts and the tools and techniques that power them!

Evolution of XSS (the solution)

- The Web fixes XSS!
- Sanitization
  - DOMPurify, Trusted Types, etc
- + CSP
  - script-src, strict-dynamic
  - unsafe-eval, unsafe-inline
- = No XSS

- Web security caught up
  - Web builders didn’t
  - Their problem ¯_(ツ)_/¯

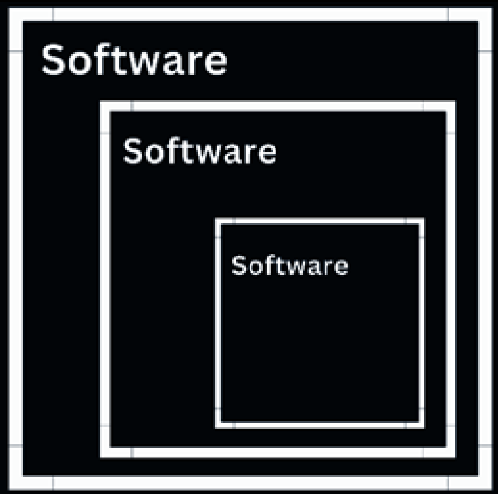


Evolution of Supply Chain Attacks

- A whole different story…
- Composability

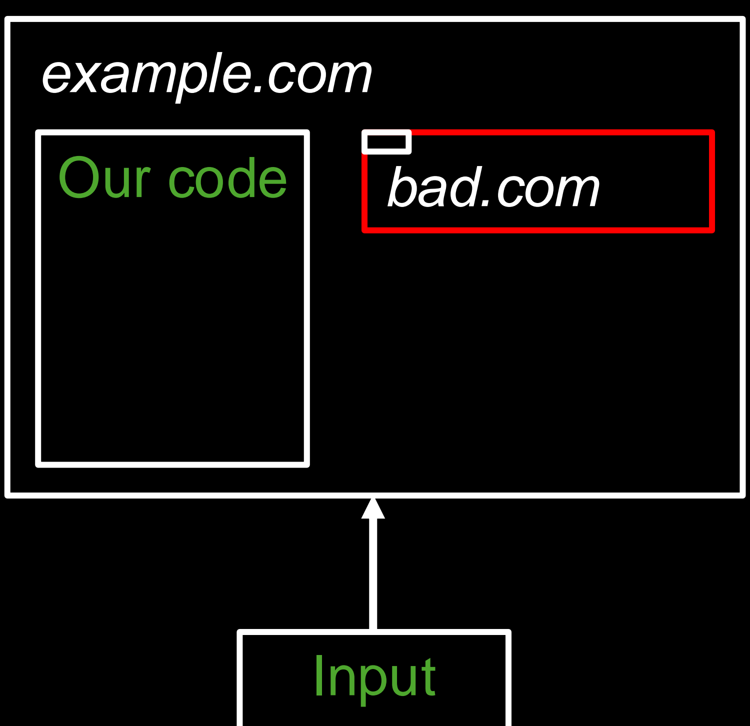
Composability (software)

- Composability
  - Software composability
    - “the ability for software components to be easily combined and integrated to create new applications or systems.”
      – (by ChatGPT)
  - Important principle
    - Development of software
    - Creation of services



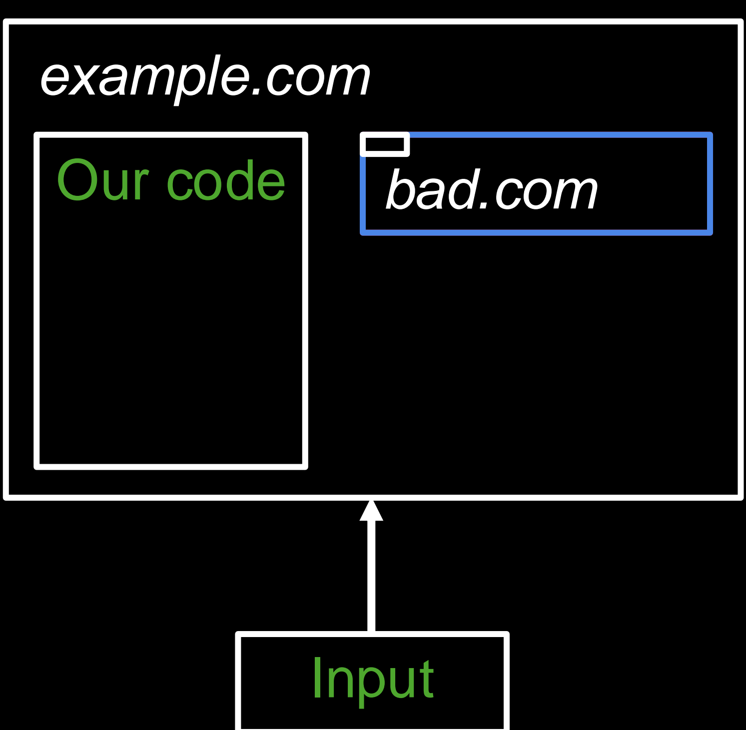
Composability (past)

- Web
  - A composability friendly ecosystem!
- Past
  - Runtime
    - <script/> or <iframe/>
  - On the client side - in the browser!



Composability (security)

- Composability *x* Security
- The Same Origin Policy
  - Isolate origins from one another
  - *bad.com* can't access *example.com*
  - *example.com* can't access *bad.com*
- What does "*access*" mean?

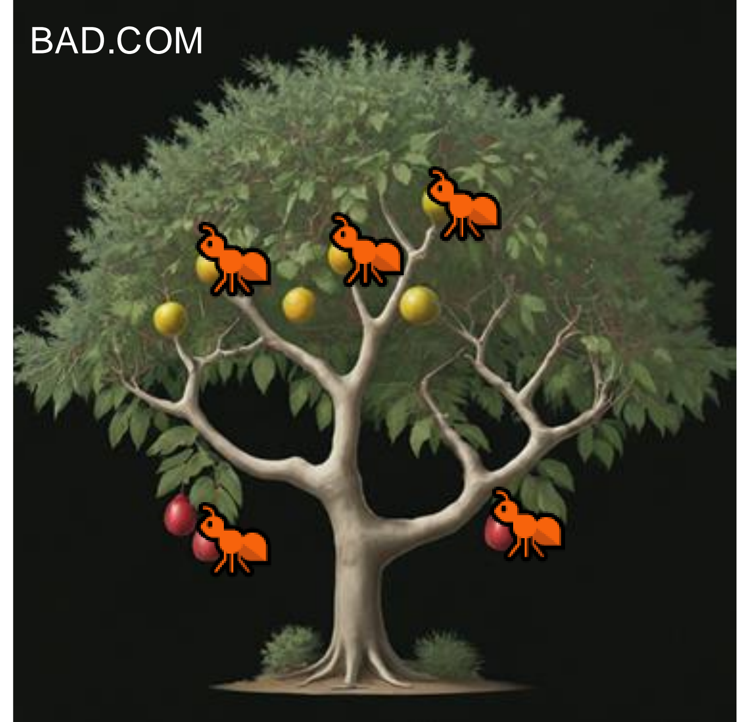


Composability (the same origin policy)

```
<html>
  <head>
    <title> EXAMPLE.COM </title>
    <script src="/code.js"></script>
  </head>
  <body>
    <h1> Welcome Guest </h1>
    <iframe src="https://BAD.COM"></iframe>
  </body>
</html>
```

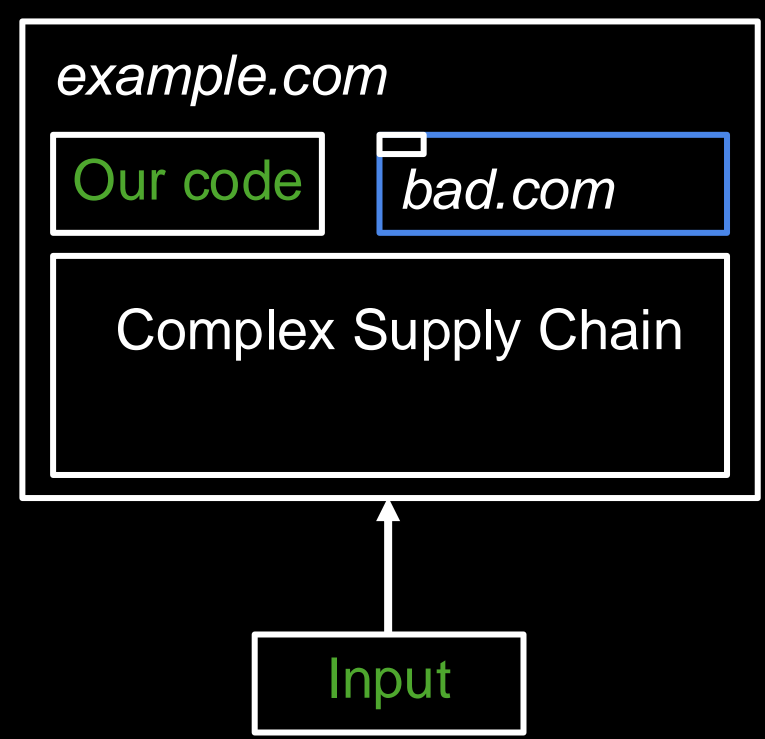
```
<html>
  <head>
    <title> BAD.COM </title>
    <script src="/code.js"></script>
  </head>
  <body>
    <h1> Welcome Guest </h1>
  </body>
</html>
```

Composability (the same origin policy)



Composability (present)

- Web
  - A composability friendly ecosystem!
- Past
  - Runtime
    - </script> or </iframe>
- Present
  - Build time
    - JavaScript composability
    - Dramatically improved
    - NPM ecosystem
      - Dependencies / packages
    - JavaScript - language of the Web
      - = Web composability improved!



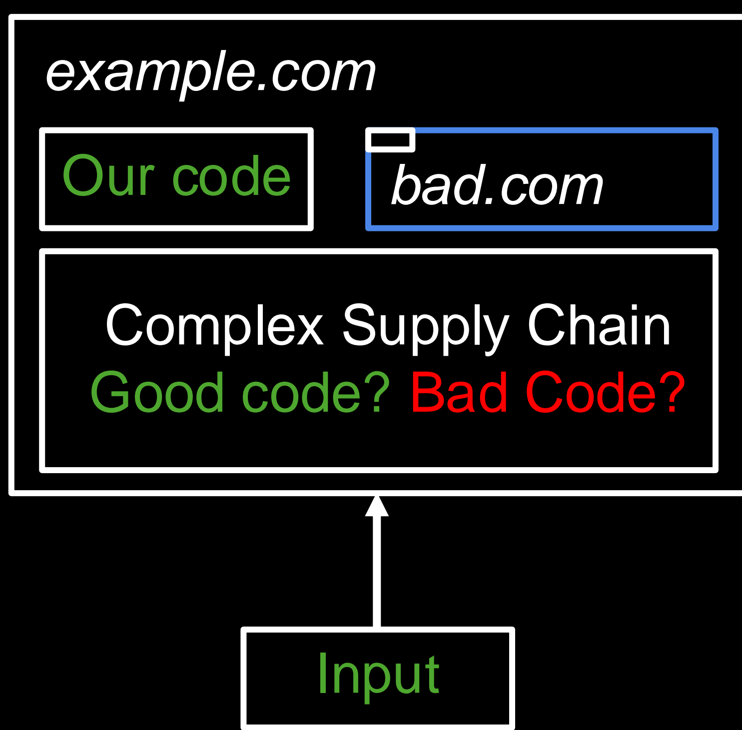
Evolution of Supply Chain Attacks (the problem)

- Composability *x* Security

1. Supply Chain Attacks differ from XSS attacks dramatically
   a. An "inside" attack (constructed mostly at build-time)
   b. We can't trust our app anymore
   c. Supply Chain Attack
      i. = Code that should(?) execute in our origin

1. Most code under our origin is maintained by someone else

1. The Web isn't ready for this
   a. Increase in resolution
   b. ~~SOP: Origin vs origin~~
   c. Dep vs dep / script vs script
      i. within one origin



Evolution of Supply Chain Attacks (the implications)

- The Same Origin Policy
  - Isolate origins from one another
  - *bad.com* can't access *example.com*
  - *example.com* can't access *bad.com*
- BUT, this also means:
  - DON'T Isolate entities within a single origin
  - *bad.com* can access *bad.com*
  - *example.com* can access *example.com*

example.com

Our code

bad.com

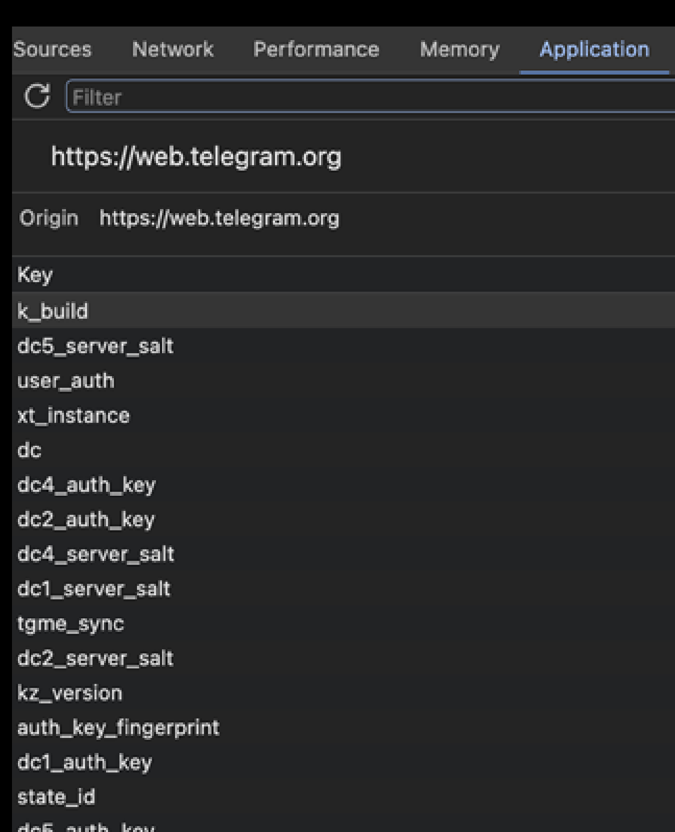
Complex Supply Chain
Good code? Bad Code?

No isolation within an origin + Complex Supply Chain =

Input

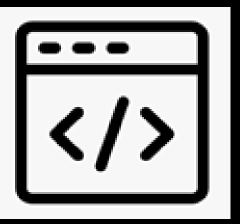
Evolution of Supply Chain Attacks (the implications)

- Storage
- Easiest to demonstrate:
  - Visit https://web.telegram.org/
  - Log in
  - Copy your localStorage
  - Open an incognito tab
  - Paste your localStorage there
  - Refresh - you should be logged in
- If you can do it manually - JavaScript can too!



DOM JUNGLE!

- DOM 🌴🌿🐒🌳
  - Well isolated from cross origin DOMs
  - One hell of a JUNGLE within a single origin!
- It all comes down to encapsulation
  - *Encapsulation is a way to restrict the direct access to some components of an object*
- JavaScript is very friendly to encapsulation
  - It has Scopes!



JavaScript Scopes!

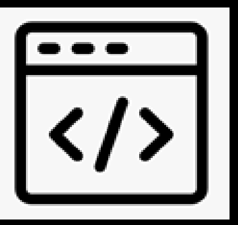
- Scope variables
  - Easily
  - Safely

```
> function createSecretLogger(secret) {
      return function logSecret() {
          console.log('This is the secet: ', secret);
      }
  }

  const logger = createSecretLogger('SCOPED_SECRET');
<· undefined
> logger
<· ƒ logSecret() {
          console.log('This is the secet: ', secret);
      }
> logger();
  This is the secet:  SCOPED_SECRET
```

DOM JUNGLE!

- DOM is terrible for safe encapsulation
- Jump between any DOM nodes
  - As long as they are attached!
- Including across iframes
  - As long as they share an origin!



DOM JUNGLE!

DOM JUNGLE!

Insert Email:

secret_address@x.com

Elements Console Sources Network Performance Memory Application

```html
<!DOCTYPE html>
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<div data-dashlane-rid="71cfa75ed3974a28" data-form-type="other">
      <p>Insert Email:</p>
      <input id="email" data-dashlane-rid="f85cab285d57bb38" data-form-type="email"> == $0
    </div>
  </body>
</html>
```

html body div input#email

Console Search What's new Issues Network conditions

top ▼ sec

```javascript
> // attacker:

  const field = document.getElementById('email');
  field.addEventListener('blur', () => {
      const stolenEmail = field.value;
      console.log('I stole your email:', stolenEmail);
  });
<· undefined
  I stole your email: secret_address@x.com
```

Evolution of Supply Chain Attacks (the implications)

Real life example

- MetaMask 🦊
  - Self custodial crypto wallet
  - Private key stored locally
  - Compromised private key == Compromised assets
  - Obsessed with client side security!
  - #1 concern - composability security risks
- LavaMoat 🌋
  - The LavaMoat JavaScript security toolbox
    - LavaMoat, Snow, scuttling, LavaDome, more
  - Enforce isolation between entities living within one single origin
- LavaDome 🌋
  - Subtree encapsulation within one single DOM
    - Against “read” attacks

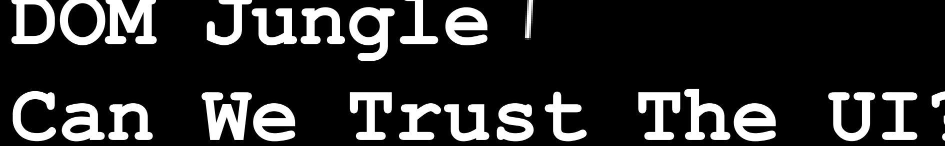
LavaDome

DEMO

Key Takeaways

- These days, Web apps are mostly code maintained by others
    - Great in terms of composability
    - Terrible in terms of security
- The Web does not take this gap into account
    - Focuses on isolation of origins
    - Not on isolation of entities under a single origin
- This puts all assets of an origin under risk
    - Storage, Network, DOM (and more)
- What if we had some policy like SOP for DOM subtrees?
    - We could defend against attackers living within our origin from:
        - Accessing sensitive info (read)
        - Performing phishing attacks (write)
- But also - we could unlock new levels of composability for the Web!

What new technologies could we invent if we could create safely encapsulated DOM components?



Gal Weizman