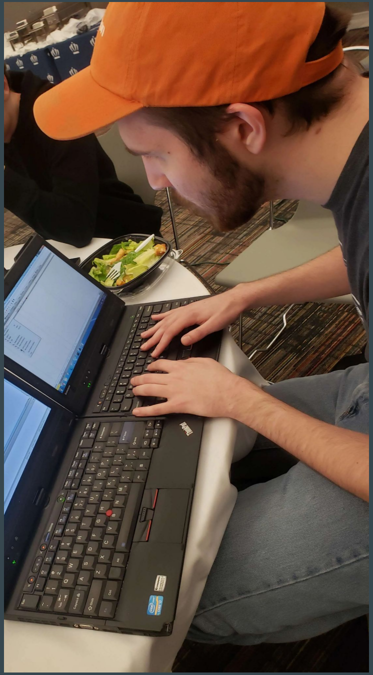
From Component to Compromised: XSS via React createElement

...

Nick Copi

whoami

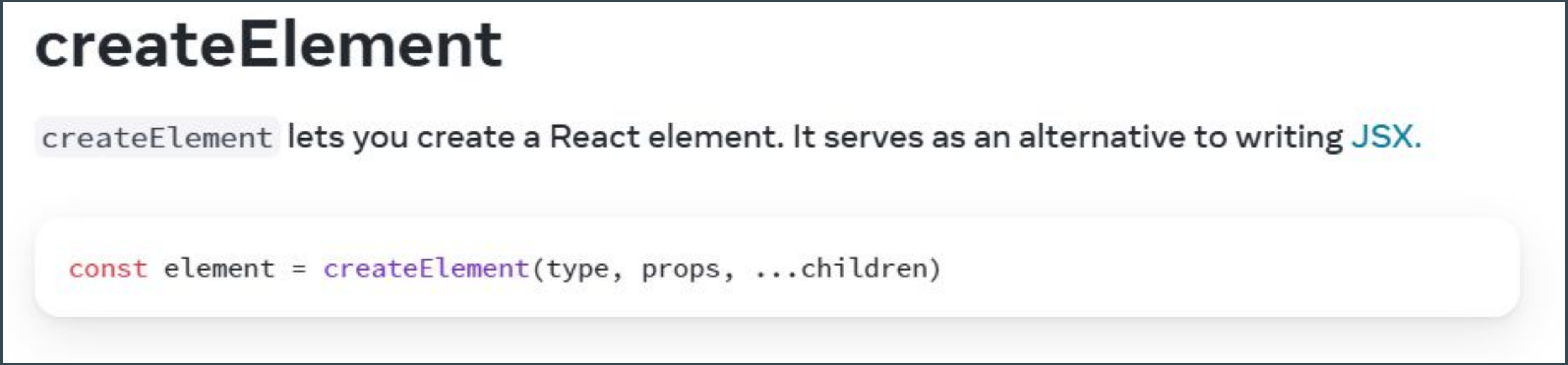
- AppSec Engineer at a Fortune 200 company by day, Independent Security Researcher by night
- Richmond Virginia born and raised God loving American patriot
- CTBB FTHG Member
- Thinkpad Dual Wielder
- https://nickcopi.site/
- @7urb01 on X
- ...More hours spent debugging JavaScript in the last year than in direct sunlight



What is React JS?

- React is a JavaScript library for building user interfaces, developed by Meta (Facebook).
- It allows developers to create reusable UI components using a declarative programming model.
- Commonly used with JSX (JavaScript XML), which compiles to React.createElement() calls.
- Widely adopted for building modern web apps, including single-page applications (SPAs).
- Behind the scenes, JSX is just sugar over React.createElement() - and that’s where our story begins...

What is createElement in React?

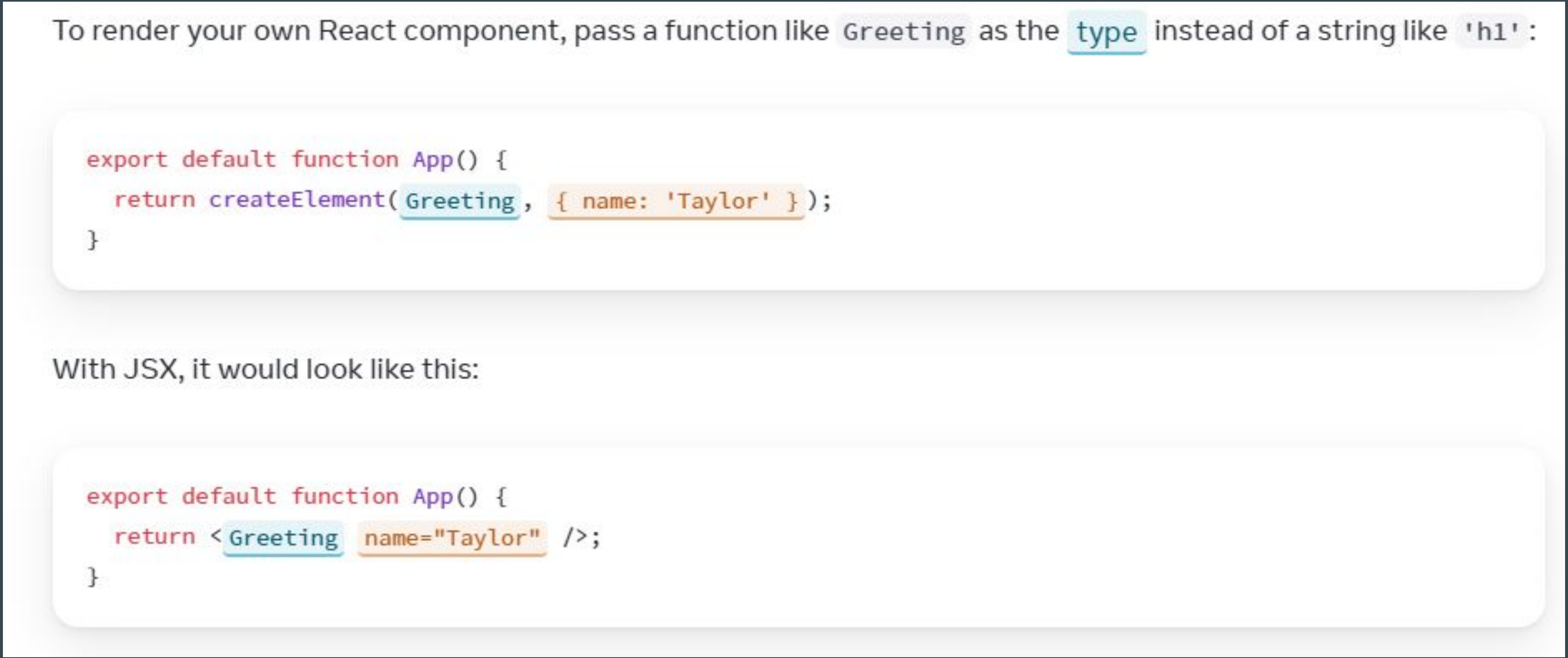


createElement

`createElement` lets you create a React element. It serves as an alternative to writing JSX.

```
const element = createElement(type, props, ...children)
```

How does JSX get compiled into React createElement calls?



To render your own React component, pass a function like `Greeting` as the `type` instead of a string like `'h1'`:

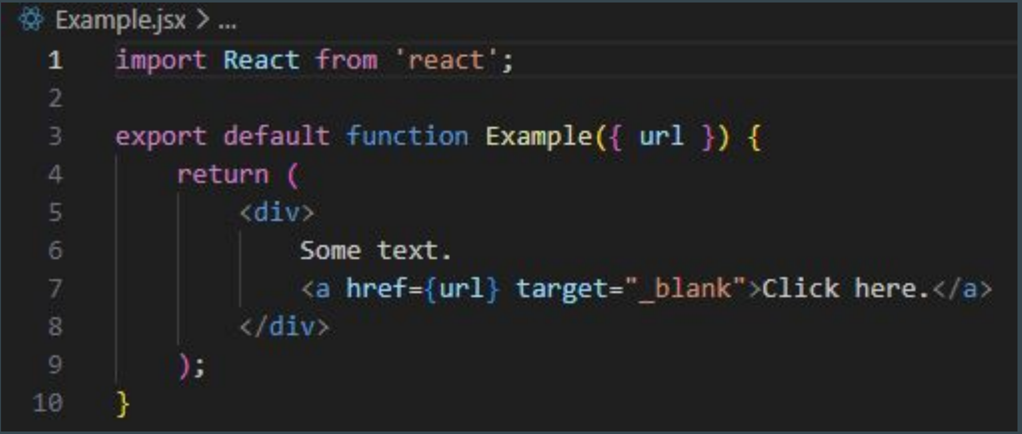
```
export default function App() {
  return createElement(Greeting, { name: 'Taylor' });
}
```

With JSX, it would look like this:

```
export default function App() {
  return <Greeting name="Taylor" />;
}
```

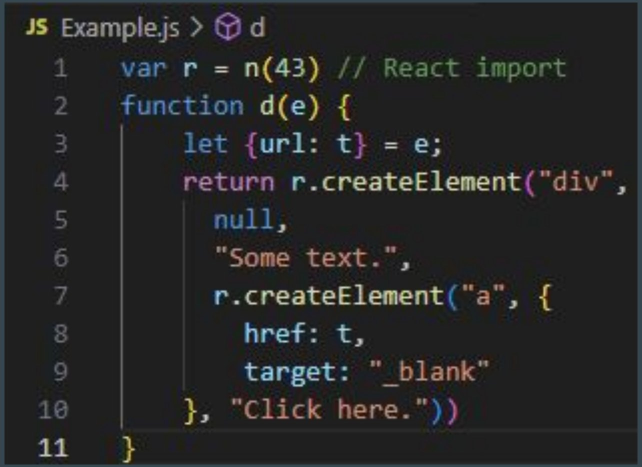
How does JSX get compiled into React createElement calls?

- Implementations vary massively between the latest versions of React and older versions that are still largely in use in the wild, but the usage of the React createElement function as a powerful sink still holds true.



Example.jsx > ...

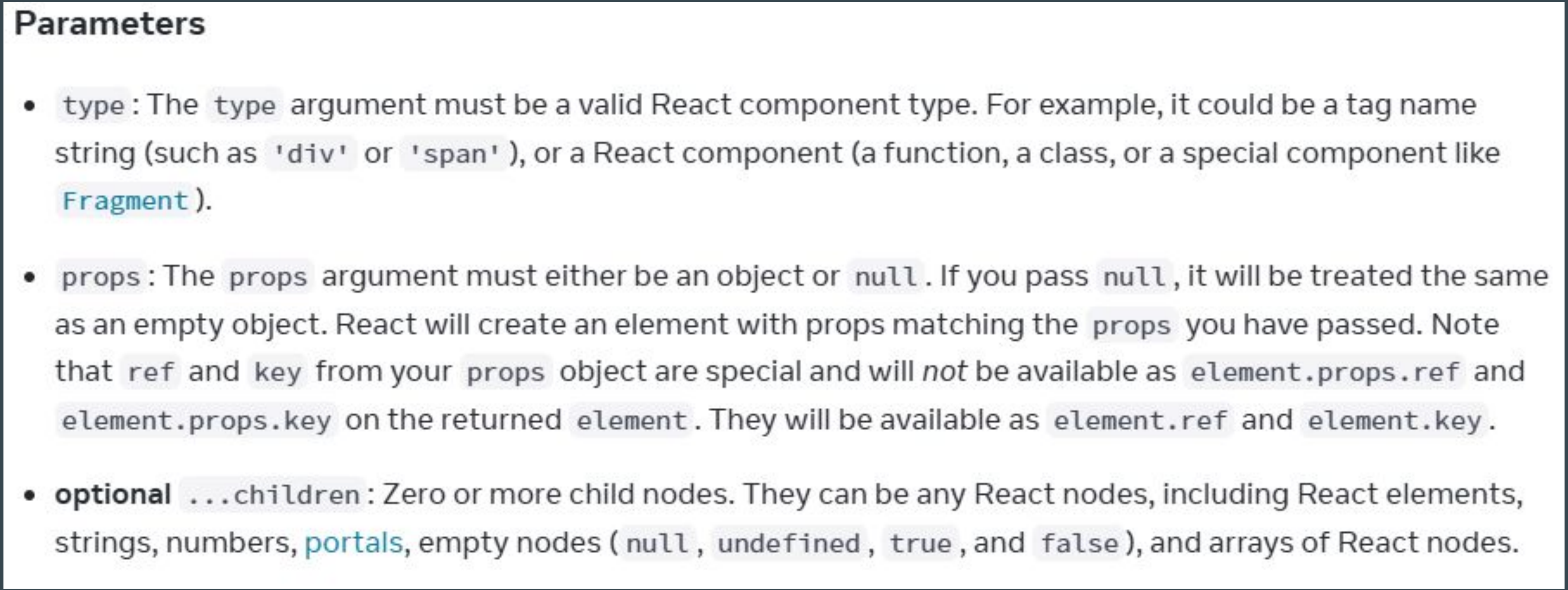
```jsx
import React from 'react';

export default function Example({ url }) {
    return (
        <div>
            Some text.
            <a href={url} target="_blank">Click here.</a>
        </div>
    );
}
```



Example.js > d

```js
var r = n(43) // React import
function d(e) {
    let {url: t} = e;
    return r.createElement("div",
      null,
      "Some text.",
      r.createElement("a", {
        href: t,
        target: "_blank"
    }, "Click here."))
}
```

Breaking down createElement's function signature



Parameters

- `type`: The `type` argument must be a valid React component type. For example, it could be a tag name string (such as `'div'` or `'span'`), or a React component (a function, a class, or a special component like `Fragment`).
- `props`: The `props` argument must either be an object or `null`. If you pass `null`, it will be treated the same as an empty object. React will create an element with props matching the `props` you have passed. Note that `ref` and `key` from your `props` object are special and will *not* be available as `element.props.ref` and `element.props.key` on the returned `element`. They will be available as `element.ref` and `element.key`.
- **optional** `...children`: Zero or more child nodes. They can be any React nodes, including React elements, strings, numbers, portals, empty nodes (`null`, `undefined`, `true`, and `false`), and arrays of React nodes.

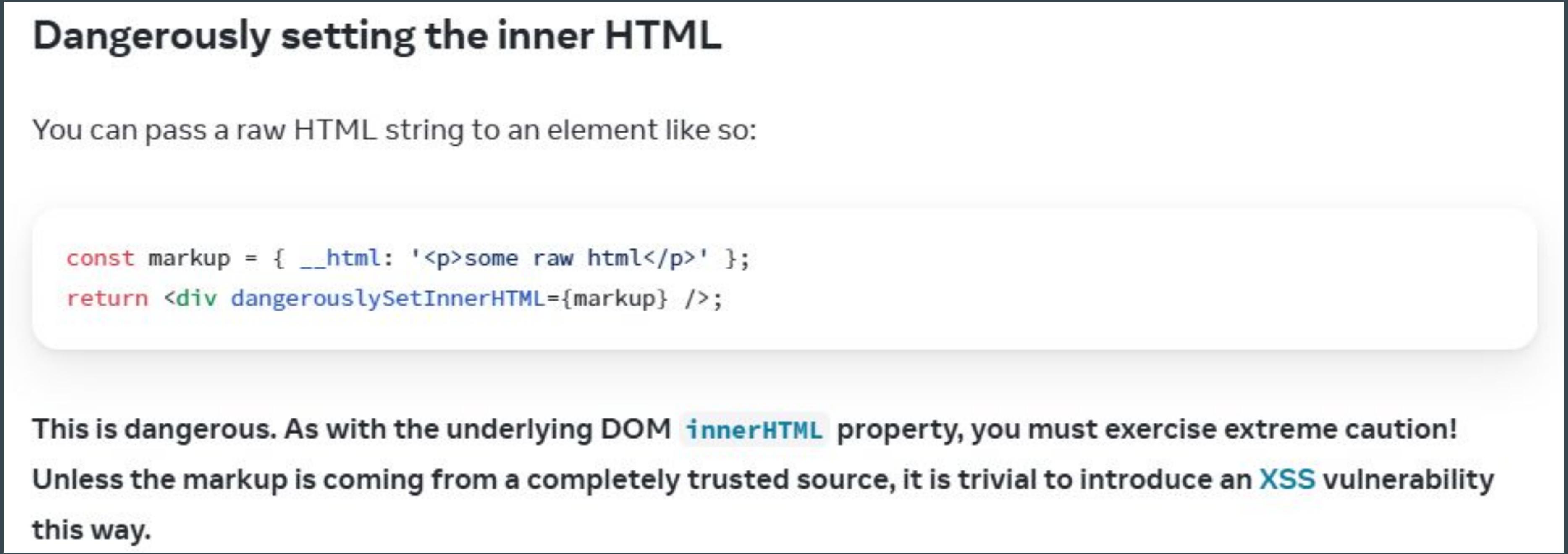
Breaking down createElement's function signature - type

- The first argument passed to createElement is the type to be created, which interestingly can be a number of values with different behaviors depending on the type of the type value
    - Strings - creates an HTML element of that literal string type (i.e. "div" -> <div></div>)
    - Functions/Classes - treats these as a React component definition and calls the appropriate code to construct and render an instance of these
- In scenarios where elements are dynamically created and type can be influenced by an attacker provided value, passing a string here instead of an expected React component can lead to unintended consequences with potential impact if more createElement arguments have some level of attacker control.

Breaking down createElement's function signature - props

- The second argument, props, is one of the better known injection points for attackers.
- An object or null is expected, and key/value pairs on this object will be assigned to the created element as props if the type is a React component, or HTML element attributes if the type is a string, with some restrictions.
- Certain special values exist, like the well known dangerouslySetInnerHTML field
- Control over certain fields of the props argument, the entire props argument, or an object spread to the props argument can be a very powerful tool for achieving XSS

Breaking down createElement's function signature - props



Dangerously setting the inner HTML

You can pass a raw HTML string to an element like so:

```
const markup = { __html: '<p>some raw html</p>' };
return <div dangerouslySetInnerHTML={markup} />;
```

This is dangerous. As with the underlying DOM `innerHTML` property, you must exercise extreme caution! Unless the markup is coming from a completely trusted source, it is trivial to introduce an XSS vulnerability this way.

Breaking down createElement's function signature - children

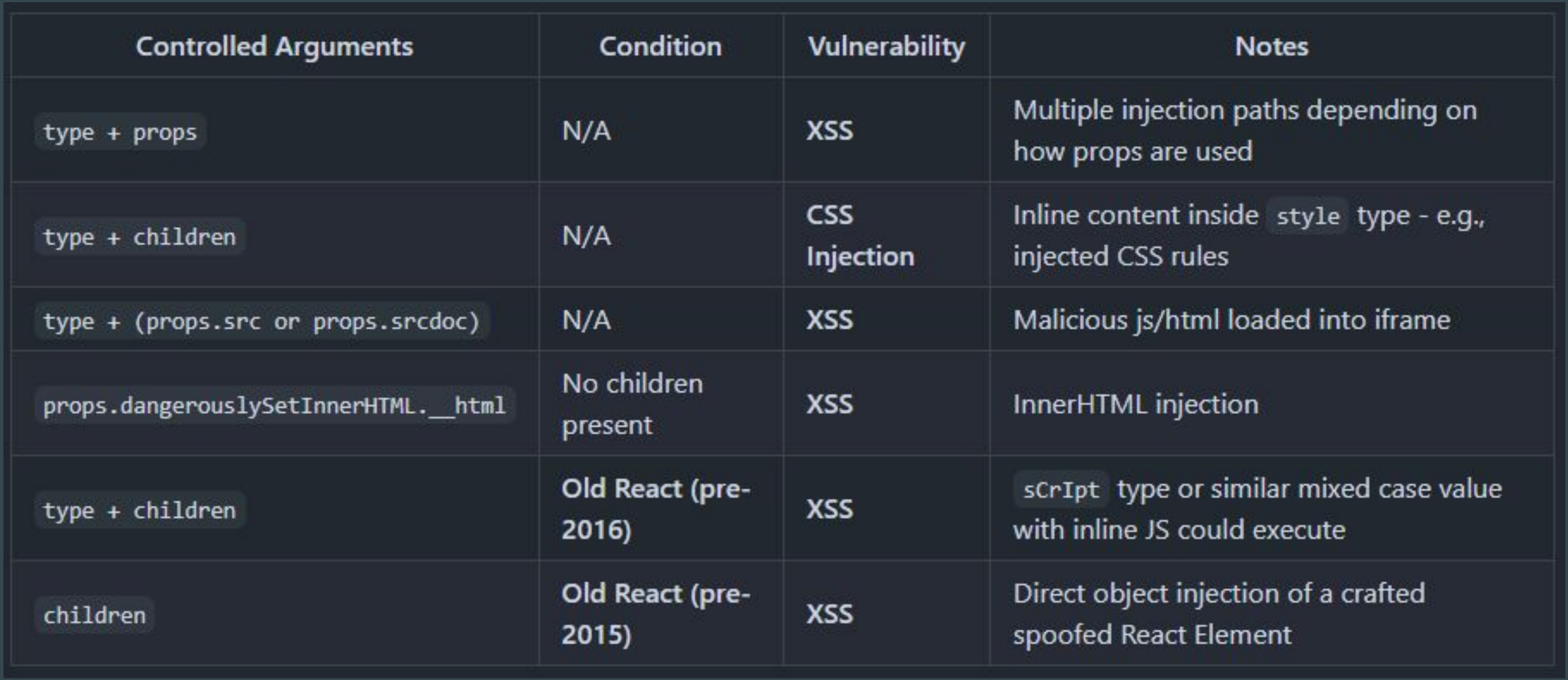
- The children argument(s) of createElement takes "React nodes"
- This can be a string literal that will be rendered as a text node
- This can be a React element object
  - In modern React, this requires certain fields be set to certain Symbol values, preventing the ability to inject valid arbitrary React elements from deserialized JSON
  - In much older React (Changed in 2015) validating these instead checks the _isReactElement: true field, allowing for arbitrary JSON to be deserialized into a valid React element, making this a much more powerful sink in ancient React versions.

What does any of this have to do with XSS?

- Quite often, components will be built in such a way that allow for attacker controlled sources to make their way into the createElement sink via one or more of its arguments
- This can be used to influence how HTML generation is performed
- These kinds of findings require a deeper understanding of what is going on with the application and are less likely to be "picked clean" on hardened bug bounty targets by traditional XSS payload sprayers

Exploitation Cheat Sheet

- Assuming attacker controlled deserialized JSON being passed into this function:



Controlled Arguments	Condition	Vulnerability	Notes
`type + props`	N/A	**XSS**	Multiple injection paths depending on how props are used
`type + children`	N/A	**CSS Injection**	Inline content inside `style` type - e.g., injected CSS rules
`type + (props.src or props.srcdoc)`	N/A	**XSS**	Malicious js/html loaded into iframe
`props.dangerouslySetInnerHTML.__html`	No children present	**XSS**	InnerHTML injection
`type + children`	**Old React (pre-2016)**	**XSS**	`sCrIpt` type or similar mixed case value with inline JS could execute
`children`	**Old React (pre-2015)**	**XSS**	Direct object injection of a crafted spoofed React Element

Lab challenges

- Pop XSS on each one of the challenges
- These challenges are inspired by vulnerabilities in real bug bounty targets
- Please talk to one another and work with the people near you. There's no reason to come all the way out to DefCon and then not talk to people.
- https://defcon.turb0.one
- Some of these challenges include source maps, some deliberately don't.
- The final challenge has a "hardcore" mode with a fun CSP to try to bypass :)
- Pop open your browser's devtools and start hacking!

Go Hack The Labs

•••

https://defcon.turb0.one

Live Demo Walkthroughs

...