February 2021

AUTOMATICALLY FIXING SECURITY SMELLS IN CONTENT SECURITY POLICIES (CSP) FOR WEB APPLICATIONS

HP INC

Automatically Fixing Security Smells in Content Security Policies (CSP) for Web Applications

ABSTRACT

Content Security Policy (CSP) is one of the most promising countermeasures against XSS, but it should follow some best practices, otherwise security smells (i.e., recurring coding patterns that are indicative of security weakness) may appear. Among the most important CSP best practices is to not allow the use of `unsafe-inline` and `unsafe-eval` policies. The method disclosed here adds innovative aspects to the current state-of-the-art to provide automatic means of getting rid of `unsafe` policies in CSP.

Existing solutions may automate the definition of CSP to some extent, but won't help with the removal of the `unsafe-inline` and `unsafe-eval` policies, which require changing the source code of dependencies:

- `unsafe-inline` can only be replaced by a nonce-based policy if the nonce is propagated to all the injected scripts;
- There's no alternative policy to the use of `unsafe-eval`, which means that "string as code" (e.g., calls to the `eval()` method) must be avoided at any cost.

We propose a method that, at development time (i.e., before releasing software), iteratively and gradually defines CSP policies according to the violations reported at each iteration. Part of our method mimics prior-art, but it adds an innovative streak in the situations in which it's impossible to define CSP policies according to best practices without changing the source code of dependencies.

The main components and interactions of our proposal are presented in *Figure 1*. Like prior-art, our method starts with a strict CSP, which iteratively and gradually gets more lenient as violations are reported (follow this link for details on how to enable CSP reporting). If the CSP violations eventually stop, the process finishes. The innovative streak of our proposal takes place if changes in web assets (e.g., JavaScript scripts) are required due to nonces that are not propagated or to the use of "string as code". In both cases, if the web assets that violate CSP are external to the application, they have to be downloaded, modified, and stored inside the application (or in any other server that is under the control of the developers of the application), thus becoming internal to the application. Next, we present more details on how our proposal works.

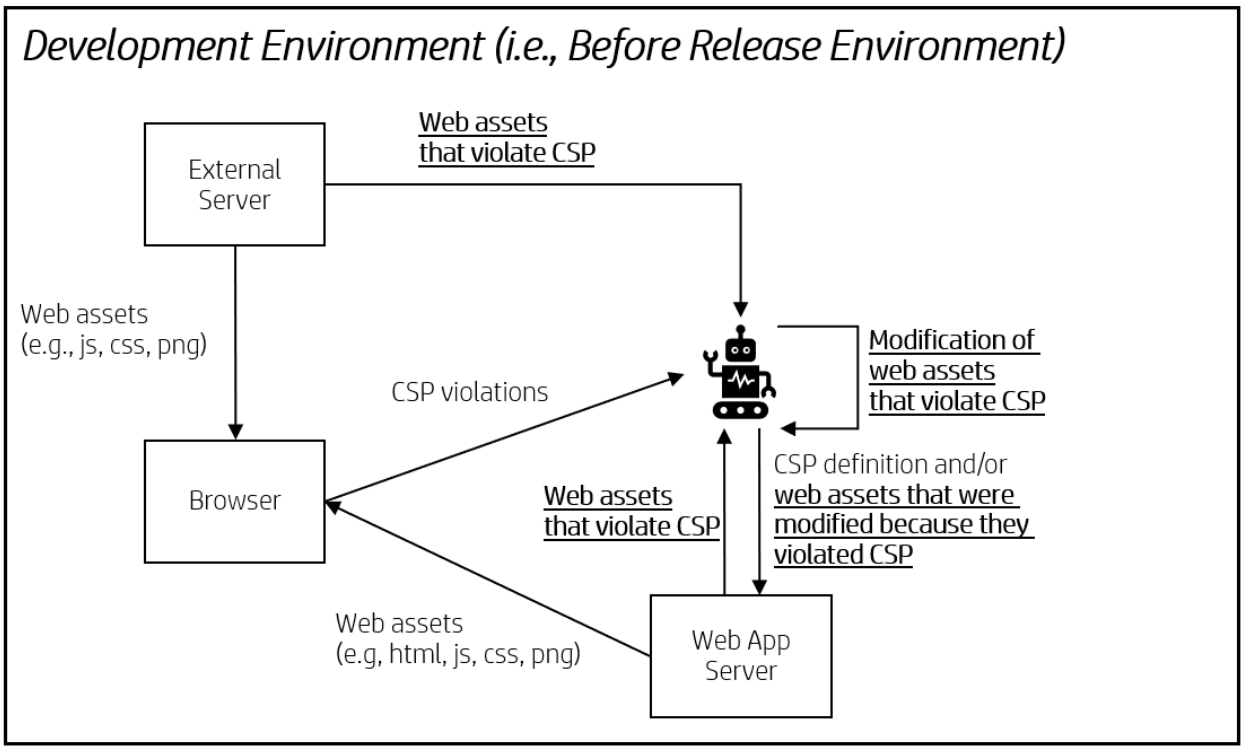

Figure 1: Our proposal's main components and their interactions. The innovative aspects of our proposal are shown **bold and underlined**.

Propagating nonces

Nonce-based policies only work if the nonce is propagated to all the injected scripts. If, for example, under a nonce-based policy, an script `A` injects another script `B`, which injects yet another script `C`, but the nonce is not propagated from `A` to `B` to `C`, CSP forbids the execution of the script that does not get the nonce, so the script that violates CSP must be fixed.

To do that, first, the exact place (i.e., JavaScript API call) of the buggy script injection must be identified, which is usually an API call like `insertBefore(script, element)`, where `script` is inserted before `element` in the DOM. This information is usually made available by popular browser development tools. Then, the script that violates CSP has to be modified to become aware of the nonce before its insertion in the DOM. This can be done by calling the API method `setAttribute(name, value)` of the injected script element, where `name` contains the string *"nonce"* and `value` contains the actual nonce value that must be propagated. *Code Snippets 1* and *2* present an example of a script that doesn't propagate the nonce and its fix, respectively.

```
var existingScript = document.getElementsByTagName('script')[0]
var newScript = document.createElement('script')
existingScript.parentNode.insertBefore(newScript, existingScript)
```

Code Snippet 1: example of a CSP-violating script that doesn't propagate the nonce.

```
// unmodified line
var existingScript = document.getElementsByTagName('script')[0]
// unmodified line
var newScript = document.createElement('script')
// inserted line: gets an element that has a nonce
var elementWithNonce = document.querySelector('[nonce]')
// inserted line: gets the nonce value
var nonce=elementWithNonce.nonce||elementWithNonce.getAttribute('nonce')
// inserted line: propagates the nonce to the newly added script
newScript.setAttribute('nonce', nonce)
// unmodified line
existingScript.parentNode.insertBefore(newScript, existingScript)
```

Code Snippet 2: Code Snippet 1 modified to properly propagate the nonce.

Avoiding “string as code”

In order to avoid “string as code”, it's necessary to replace all `eval()` calls with the actual value of the strings that are executed by them. We do it in 2 steps. *Code Snippets 3*, *4*, and *5* present an example of such a replacement, in which the “string as code”, for the sake of simplicity, is `“1 + 1”`. Basically, the first step saves the value of the “string as code” in a persistent storage and the second step replaces the `eval()` call with the value saved in the persistent storage.

```
var stringAsCode = operationsToBeEvaluated(); // assume “1 + 1”
eval(stringAsCode) // evaluates 1 + 1, which returns 2
```

Code Snippet 3: example of code that executes “string as code” through an `eval()` call.

```
var stringAsCode = operationsToEvaluated(); // assume “1 + 1”
storePersistently(stringAsCode) // saves "1 + 1" to a persistent storage
eval(stringAsCode) // evaluates 1 + 1, which returns 2
```

Code Snippet 4: Step 1 for the removal of `eval()`, which saves the string to be executed in a persistent storage that will be used in step 2.

```
1 + 1 // content of the persistent storage saved in step 1
```

Code Snippet 5: Step 2 for the removal of `eval()`, which substitutes it with the content of the persistent storage saved in step 1.

How to keep up with evolving scripts?

In order to allow the web application to keep up-to-date or, at least, aware of potentially stale script dependencies that keep evolving, our method should be integrated with a CI/CD pipeline. At each execution of the pipeline, our method should be executed from scratch (i.e., using the original external dependencies), and the textual difference between the output of the last execution and the output of the previous one should be calculated. If the result is not empty, then it can be concluded that the external dependency has evolved, which may trigger an event that

keeps developer awareness by notifying the evolution of external dependencies. The pipeline may always use the original dependencies, but before generating a release, it applies our method, thus "freezing" the modified dependencies for released software.

Disclosed by Mauricio Moraes, Jhonny Mertz, and Chris Myers, HP Inc.