USENIX'23 Artifact Appendix: A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs

Gertjan Franken
imec-DistriNet, KU Leuven

Tom Van Goethem
imec-DistriNet, KU Leuven

Lieven Desmet
imec-DistriNet, KU Leuven

Wouter Joosen
imec-DistriNet, KU Leuven

A Artifact Appendix

A.1 Abstract

BugHog is a comprehensive framework designed to identify the complete lifecycles of bugs, from their introduction to mitigation, and potential regression. For each bug's proof of concept (PoC) integrated in the BugHog experiment web server, the framework can perform automated and dynamic experiments using Chromium and Firefox revision binaries.

Each experiment is performed within a dedicated Docker container, ensuring the installation of all necessary dependencies, in which BugHog downloads the appropriate browser revision binary, and instructs the browser binary to navigate to the locally hosted PoC web page. Through observation of HTTP traffic, the framework determines whether the bug is successfully reproduced. Based on experiment results, BugHog can automatically bisect the browser's revision history to identify the exact revision or narrowed revision range in which the bug was introduced or fixed.

The framework offers a graphical user interface, accessible through a locally hosted web page. The experiment results are visualized using a Gantt chart, facilitating easy interpretation and analysis.

In our study, BUGHOG was employed to identify the lifecycles of 75 bugs related to the Content Security Policy, across its complete development history in Chromium and Firefox.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing our artifact on evaluators' machines poses no risk.

A.2.2 How to access

The stable version of BugHog as part of the USENIX artifact evaluation process is available at https://github.com/DistriNet/BugHog/tree/usenix23-artifact-stable. The most recent version, which will also be maintained and updated in the future, can be found at https://github.com/DistriNet/BugHog. To clone the repository, use the following command:

```
git clone https://github.com/DistriNet/BugHog.git
```

Or use the GitHub web interface.

A.2.3 Hardware dependencies

To run the framework, the following minimum hardware specifications are required:

- 2 CPU cores (more cores may be necessary for concurrent experiments).
- 8 GB of RAM.
- 5 GB of disk space.
- Internet connection.

Ideally, the number of CPU cores should match or exceed the number of concurrent experiments the user allows BugHog to perform. Insufficient disk space may cause the framework to crash since this could prevent it from temporarily storing downloaded binaries.

A.2.4 Software dependencies

The BUGHOG framework only relies on Docker, making it compatible with any operating system that supports Docker (e.g. Windows, macOS, Linux). All necessary dependencies are included in the Docker images, eliminating the need for additional software installations.

A.2.5 Benchmarks

No specific benchmarks are associated with this artifact.

A.3 Set-up

The installation instructions that follow can be found in our GitHub repository's `README.md` as well.

A.3.1 Installation

Follow these steps to install BUGHOG:

1. Clone the repository, and navigate to the root directory:

```
git clone https://github.com/DistriNet/BugHog.git
cd BugHog
```

   The project can also be downloaded through GitHub's web interface at https://github.com/DistriNet/BugHog instead.

2. Obtain the required BUGHOG Docker images:

   - Option A: Pulling (fastest)

     Use the following command to pull the necessary pre-built Docker images:

```
docker compose pull core worker web
```

   - Option B: Building

     If you intend to modify the source code, use the following commands to build the required Docker images. Rerun this script if you make any changes to the source code:

```
docker compose up node_install_deps
docker compose up node_build
docker compose build core worker web
```

     For reference, building the images takes approximately 4 minutes on a machine with 8 CPU cores and 8 GB of RAM.

3. (**Optional**) Use your own MongoDB instance.

   If you prefer using your own MongoDB instance, provide the connection parameters in a `.env` file at the project's root:

```
bci_mongo_host=[ip_address_of_host]
bci_mongo_database=[database_name]
bci_mongo_username=[database_user]
bci_mongo_password=[database_password]
```

   If not provided, BUGHOG will spin up a MongoDB instance in a Docker container. The data is persisted between runs within the `database` folder, allowing you to safely stop and start BUGHOG without losing any data.

A.3.2 Basic Test

To start the framework, execute the following command:

```
docker compose up core web
```

Depending on the installation option chosen earlier, a pulled or locally built image will be used. You can switch between the two options by executing the appropriate installation step before starting the framework.

To access the web interface, open your web browser and navigate to http://localhost:5000. BUGHOG is ready for use when the following message is logged in either the terminal window or the web interface:

```
[INFO] bci.master: BugHog is ready!
```

Perform a simple test by following these steps:

1. Select the `CSP` project and `Chromium` browser from the dropdown menus in the upper left corner of the web interface.
2. Choose the `c1064676` experiment from the `Experiments` pane. [1]
3. Set the `Evaluation range` with a lower version of 20 and an upper version of 110.
4. Input 1 for the `Number of parallel containers`. [2]
5. Click the green `Start evaluation` button.

As the evaluation progresses, the framework will provide updates in the terminal window or `Log` pane at the bottom of the web interface. The information and Gantt chart in the `Results` pane will be updated automatically as well, if `c1064676` is selected in the `Select an experiment` dropdown menu. Please note that the Gantt chart requires a minimum of two completed experiments before it can be generated. Each dot in the Gantt chart represents whether the bug can be reproduced in the corresponding revision binary. By hovering over a dot, the associated revision number and browser version can be observed. To prevent the Gantt chart from refreshing automatically, the `Auto-refresh Gantt chart` checkbox can be unchecked. This might be necessary when zooming in on a specific part of the chart, since the zoom level will be reset when the chart is refreshed. A refresh can be triggered manually by clicking the `Refresh` button.

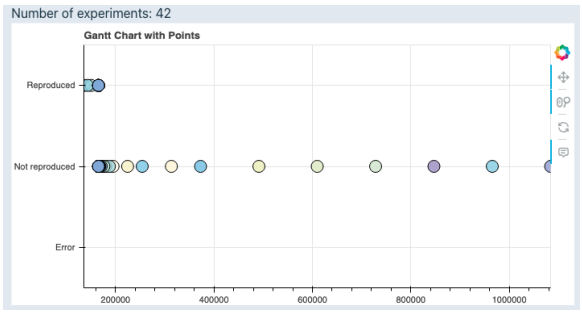
You have the option to stop the evaluation at any time by clicking either the yellow `Stop gracefully` button, which allows the ongoing experiments to finish before stopping, or the red `Stop forcefully` button, which immediately attempts to halt all experiments. When all experiments have ended, either by user intervention or because the last available revision has been evaluated, the following line will be logged:

```
[INFO] bci.master: BugHog has finished the evaluation!
```

[1] Experiments are named after the bug report ID. Experiments with a `c` as prefix are Chromium reported bugs, while experiments with an `f` as prefix are Firefox reported bugs.

[2] Feel free to increase this number if you have more available CPU cores.

(a) Chromium.



(b) Firefox.

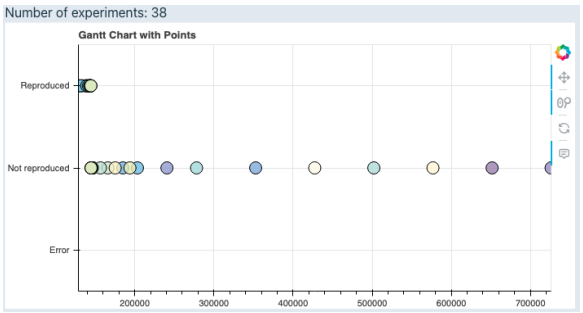


Figure 1: Resulting Gantt chart of experiment (E1).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BUGHOG is capable of running all browser revision binaries that support CSP enforcement[3] required for our study, without the need for manual dependency management. This claim is supported by experiment (E1) conducted on both browsers.

(C2): We successfully identified the complete lifecycles of 75 bugs in Chromium and Firefox using BUGHOG. Considering the impracticality of evaluators individually pinpointing the lifecycles of all bugs, we will provide our comprehensive MongoDB lifecycle dataset.[4] Evaluators can utilize BUGHOG to pinpoint the lifecycle of any bug in our dataset and compare them with our findings. Furthermore, as an example, we describe experiment (E2) where we trace the lifecycle of one of the CSP bugs.

A.4.2 Experiments

The experiments were conducted on a machine equipped with 8 CPU cores and 8 GB of RAM, where BUGHOG was configured to use 8 parallel containers.

(E1): *[Support for the complete CSP browser development history] [20 human-minutes + 15 compute-minutes]*:

Description: BUGHOG is capable of running all necessary browser revision binaries that support CSP without the need for manual dependency management. To demonstrate this capability, we conduct an experiment to identify the introduction of CSP in Chromium and Firefox. For this experiment, we utilized a PoC that employs a CSP policy blocking all resource loading through the `default-src` directive, which is supported since CSP's introduction. In revisions without CSP support, requests are allowed to reach our server, indicating successful reproduction. The PoC can be found in the `experiments/pages/Support/CSP` directory.

Execution: Select the following evaluation parameters:

- `Project`: Support
- `Browser`: Chromium for one evaluation, Firefox for the other
- `Experiment`: CSP
- `Evaluation range`: 20 to 110
- `Search strategy`: Binary search
- `Reproduction id`: csp
- `Number of parallel containers`: any number from 2 to 8 (higher numbers will result in faster evaluation)

Click the green `Start evaluation` button to begin the evaluation.

Results: By refreshing the `Results` pane in the web interface, you will eventually observe that `BugHog` successfully identifies a specific revision range in which the introduction of CSP occurred. The Gantt chart for both browsers will also exhibit a distinct pattern indicating the utilization of binary search. Figures 1a and 1b show the resulting Gantt charts for Chromium and Firefox, respectively.
By solely using downloaded publicly available revision binaries, we can infer that the introduction of CSP in Chromium took place at revision 165317.[5] This revision corresponds to a WebKit roll, where WebKit's revision range [133029 - 133116] was integrated into Chromium. Through manual analysis of revision metadata, we determined that revision 133131[6] is the exact revision responsible for introducing CSP.

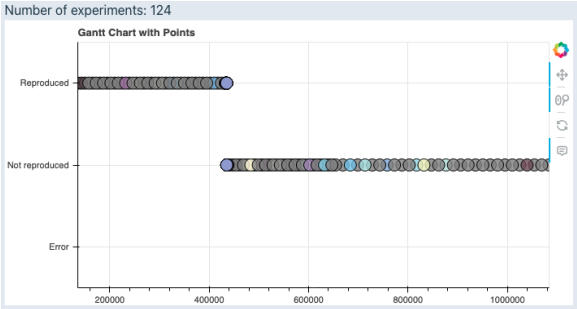
[3] CSP support starts from revision 165317 and 144546 for Chromium and Firefox, respectively.

[4] A dump of this dataset is available at: https://github.com/DistriNet/lifecycle-data

[5] https://crrev.com/165317

[6] https://trac.webkit.org/changeset/133131/webkit

(a) Chromium.



(b) Firefox.

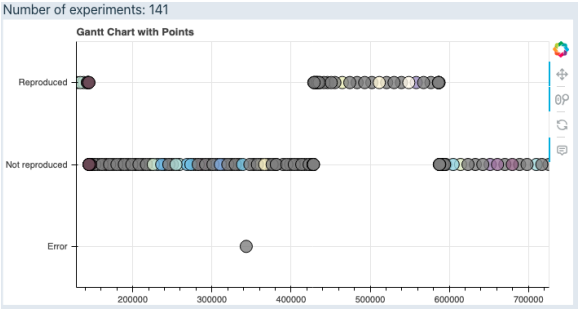


Figure 2: Resulting Gantt chart of experiment (E2).

For Firefox, the framework narrows down the revision range to [144529 - 144643]. Here, through a combination of self-built binaries and manual analysis, we inferred that revision 144546[7] indeed introduced CSP.

(E2): *[Full lifecycle analysis] [20 human-minutes + 40 compute-minutes]*:

BUGHOG provides a comprehensive lifecycle analysis of bugs through the `Composite search` method. This approach involves two stages:

- In the first stage, N evenly spread out revisions are evaluated over the whole indicated range, with N determined by the value of `Sequence limit`.
- In the second stage, the analysis focuses on identifying smaller revision ranges where reproducibility shifts are observed.

In this experiment, we will conduct a lifecycle analysis of bug `f1441468`, which was reported for Firefox.[8] Although we did not find this bug reported for Chromium, `BugHog` has revealed that the bug is reproducible in Chromium as well in our cross-browser analysis. Therefore, we will perform this evaluation on both browsers again in this experiment.
Unlike the previous experiment, our goal here is not just to determine when the bug was introduced. We aim to obtain a complete view of the bug's lifecycle. To achieve this, we will employ the `Composite search` strategy. To ensure that we evaluate approximately one binary per release version, we will set the `Sequence limit` to 100.

Execution: Select the following evaluation parameters:

- `Project`: CSP
- `Browser`: Chromium for one evaluation, Firefox for the other
- `Experiment`: f1441468
- `Evaluation range`: 20 to 110
- `Search strategy`: Composite search
- `Sequence limit`: 100
- `Reproduction id`: f1441468
- `Number of parallel containers`: any number from 2 to 8 (higher numbers will result in faster evaluation)

Click the green `Start evaluation` button to begin the evaluation.

Results: Figures 2a and 2b show the resulting Gantt charts for Chromium and Firefox, respectively.
For Chromium, BUGHOG shows that this bug is foundational, as the bug is reproducible since the introduction of CSP. BUGHOG also identifies a narrow revision range, specifically [435165 - 435177], where an effective fix was applied. By manually analyzing revision metadata, we can determine that the bug was fixed in revision 435165.[9]
In the case of Firefox, the results indicate that the bug is non-foundational since it could not be reproduced at the time of CSP introduction. Instead, the bug was introduced within revision range [428395 - 428677], and subsequently fixed within revision range [587121 - 587215]. Through a combination of self-built binaries and manual analysis of revision metadata, we can identify the introducing revision as 428568[10] and the fixing revision as 587202.[11]
Both results can be cross-referenced against the lifecycle data dump by opening the `json` file in any text editor with string search functionality, and searching for the bug ID without the single-letter prefix (i.e. 1441468). The object associated with this ID contains the bug's life-

[7] https://hg.mozilla.org/releases/mozilla-release/rev/6b181afc9fadbd4bb9d04648aa24a34bd9731e82
[8] https://bugzilla.mozilla.org/show_bug.cgi?id=1441468
[9] https://crrev.com/435165
[10] https://hg.mozilla.org/releases/mozilla-release/rev/428568
[11] https://hg.mozilla.org/releases/mozilla-release/rev/587202

cycles for both browsers, in which the aforementioned revisions are listed.

A.5 Notes on Reusability

As mentioned in our paper, BUGHOG is not limited to evaluating CSP bugs but can also be used to analyze other types of bugs, including those impacting other (security) policies and functionalities. By integrating the bug's PoC into the framework, BUGHOG can uncover its complete lifecycle. This integration is performed by adding the necessary web page files to the `experiments/pages` folder and by including the URL queue of pages to be visited during the experiment in the `experiments/url_queues` folder. Since the integration format may evolve in the future, we provide more detailed instructions on how to integrate new bug PoCs in the `README.md` file of our GitHub repository.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.