1 CSCG 2025 - vidplow

Category: Web

Difficulty: Medium

Author: Popax21

Flag: `dach2025{wh0_n33ds_n0de_j5_anyw4y_edabc46f6280250fc7e646dee1a3937c}`

Description:

> We recently stumbled upon an exposed SVN server of a large multimedia corporation, containing some of their backend application and internal tooling code. However, the access keys seem to not be the ones used in production - the real ones should fetch us quite a high price though, if we manage to get our hands on them that is. Just one problem - the tech stack seems to be really obscure, and no one on our team seems to have any clue what the heck is going on. Can you take a look, and maybe find some vulnerabilities in this thing?

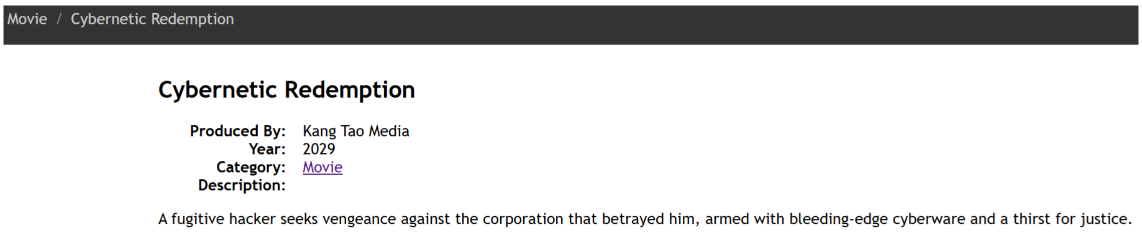
2 Introduction

The challenge is about the old web framework Helma, which has not been maintained for more than 7 years. The challenge's objective is to retrieve the access key of the configuration file.

In the following sections, we walk through the entire process, from the initial analysis to the final exploitation.

3 Reconnaissance

Starting with the landing page, we have a selection of different movie categories containing movies and subcategories.


Movie / Cybernetic Redemption

Cybernetic Redemption

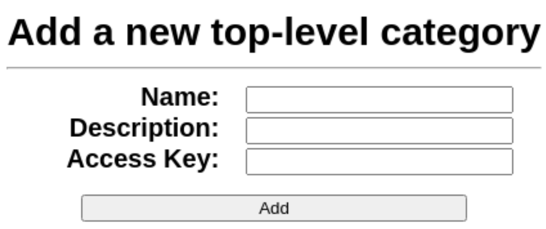
Produced By: Kang Tao Media
Year: 2029
Category: Movie
Description:

A fugitive hacker seeks vengeance against the corporation that betrayed him, armed with bleeding-edge cyberware and a thirst for justice.

The server files seem very odd with file extensions like `.hac` , `.hsp` or `.skin` . After some research, it's clear that the old web framework named Helma, written in *Java*, is being used. The flag is hidden in the `app.properties` configuration file in the `accessKey` property. With this key, we can modify, add and delete any categories and movies from the website, as we will see in a moment. Fortunately, there is some documentation about this obscure framework.

Helma is organized around two core concepts: HopObjects and Actions. `HopObjects` are Helma's server-side, *JavaScript*-implemented object types that can be persisted in a database. They allow developers to model application data as native *JavaScript* objects with properties, methods and inheritance, without having to write separate database schemas. `Actions`, on the other hand, are the controllers in *Helma's* MVC-style architecture. Each `Action` is a *JavaScript* function that is bound to a specific URL endpoint and is responsible for processing incoming HTTP requests, manipulating `HopObjects`. `Actions` can also render responses in *HTML*, *JSON* and other formats.

The main `HopObject` is the `root` object, defining the entry point of the web page with the `Root/main.hac` file, like in this challenge. There is an `Action` defined with the path `/add_category`, triggering the code inside `Root/add_category.hac`. Checking it out, we are greeted by an HTML form requiring, among other things, the `accessKey`:


Add a new top-level category

Name:	
Description:	
Access Key:	

The `Root/type.properties` describes that accessing the `Root` object's collection goes via the `name` properties of the underlying `HopObjects`. Added categories via `add_category` are added to the collection of `Root`. The same goes for subcategories and videos, which can be added to categories. This effectively means by clicking through the hyperlinks in the website we can traverse the collections of the `HopObjects` starting with `Root`, going recursively over `Category`, if subcategories were added, and finally reaching `Video` and the `Property` objects.

4 Vulnerability description

The challenge files overwrite the usual `HopObjects` behaviour. They will always call the `getProperty` function of a `HopObject` to retrieve its properties:

```
function getChildElement(name) {
    if (this.getProperty && this.getProperty(name)) {
        var prop = new Property();
        prop.obj = this;
        prop.name = name;
        return prop;
    }
    return this.get(name);
}
```

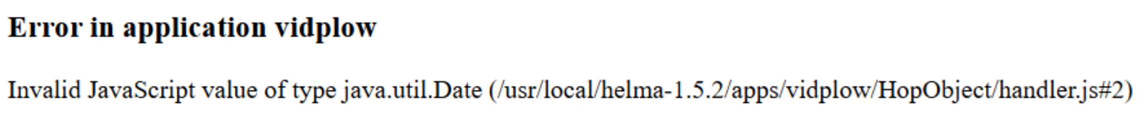
The `getProperty` function for `Video` objects works slightly differently than for the others:

```
function getProperty(name) {
    if (name == "category") {
        return this._parent;
    } else {
        return this[name];
    }
}
```

We control the `name` parameter for this function via the URL path. Due to `this[name]` , we can retrieve any property of the object `this` like `name` and `producer` . But more importantly, we can also get internal properties like `_id` or `__parent__` . The `accessKey` is part of the `Global` object, so effectively we need to traverse the properties of any `Video` object, reaching the `Global` object and its `accessKey` property to retrieve the flag.

5 Setting up a local instance

Setting up a local instance is not quite necessary, but it makes debugging much easier. For this, we need to leak the used *Helma* version, which turns out pretty simple. We can trigger an error, for example, by checking out the `__created__` property, revealing it is *Helma* v1.5.2:



Error in application vidplow

Invalid JavaScript value of type java.util.Date (/usr/local/helma-1.5.2/apps/vidplow/HopObject/handler.js#2)

6 Exploitation

As the code is the best documentation, let's have a look at the source code of HopObjects. Here we find all of the different internal properties. Playing around with `__parent__` will result in following URL to obtain the flag:

```
/Documentary/The+Fall+of+Arasaka+Tower/__parent__/accessKey
```

```
dach2025{wh0_n33ds_n0de_j5_anyw4y_edabc46f6280250fc7e646dee1a3937c}
```

7 Mitigation

All of the information leaks come from handling unvalidated and unsanitized user input in the `getProperty` function of the `Video` object. Moreover, dynamically handling access to properties like this always creates openings for attackers. Using for example a white list would be much more secure in this case only allowing specific properties to be accessed. Finally, avoid using an unmaintained web framework like *Helma* - migrate to a more modern alternative.