Web cache poisoning

In this section, we'll talk about what web cache poisoning is and what behaviors can lead to web cache poisoning vulnerabilities. We'll also look at some ways of exploiting these vulnerabilities and suggest ways you can reduce your exposure to them.

What is web cache poisoning?

Web cache poisoning is an advanced technique whereby an attacker exploits the behavior of a web server and cache so that a harmful HTTP response is served to other users.

Fundamentally, web cache poisoning involves two phases. First, the attacker must work out how to elicit a response from the back-end server that inadvertently contains some kind of dangerous payload. Once successful, they need to make sure that their response is cached and subsequently served to the intended victims.

A poisoned web cache can potentially be a devastating means of distributing numerous different attacks, exploiting vulnerabilities such as XSS, JavaScript injection, open redirection, and so on.

Labs

If you're already familiar with the basic concepts behind web cache poisoning and just want to practice exploiting them on some realistic, deliberately vulnerable targets, you can access all of the labs in this topic from the link below.

View all web cache poisoning labs »

Web cache poisoning research

This technique was first popularized by our 2018 research paper, "Practical Web Cache Poisoning", and developed further in 2020 with a second research paper, "Web Cache Entanglement: Novel Pathways to Poisoning". If you're interested in a detailed description of how we discovered and exploited these vulnerabilities in the wild, the full write-ups are available on our research page.

Research

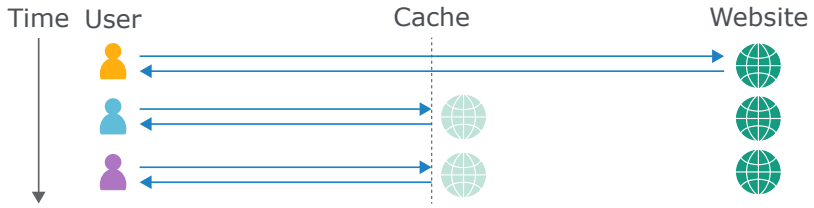
Practical Web Cache Poisoning »
Web Cache Entanglement: Novel Pathways to Poisoning »

How does a web cache work?

To understand how web cache poisoning vulnerabilities arise, it is important to have a basic understanding of how web caches work.

If a server had to send a new response to every single HTTP request separately, this would likely overload the server, resulting in latency issues and a poor user experience, especially during busy periods. Caching is primarily a means of reducing such issues.

The cache sits between the server and the user, where it saves (caches) the responses to particular requests, usually for a fixed amount of time. If another user then sends an equivalent request, the cache simply serves a copy of the cached response directly to the user, without any interaction from the back-end. This greatly eases the load on the server by reducing the number of duplicate requests it has to handle.



Cache keys

When the cache receives an HTTP request, it first has to determine whether there is a cached response that it can serve directly, or whether it has to forward the request for handling by the back-end server. Caches identify equivalent requests by comparing a predefined subset of the request's components, known collectively as the "cache key". Typically, this would contain the request line and `Host` header. Components of the request that are not included in the cache key are said to be "unkeyed".

If the cache key of an incoming request matches the key of a previous request, then the cache considers them to be equivalent. As a result, it will serve a copy of the cached response that was generated for the original request. This applies to all subsequent requests with the matching cache key, until the cached response expires.

Crucially, the other components of the request are ignored altogether by the cache. We'll explore the impact of this behavior in more detail later.

What is the impact of a web cache poisoning attack?

The impact of web cache poisoning is heavily dependent on two key factors:

- **What exactly the attacker can successfully get cached**
  As the poisoned cache is more a means of distribution than a standalone attack, the impact of web cache poisoning

is inextricably linked to how harmful the injected payload is. As with most kinds of attack, web cache poisoning can also be used in combination with other attacks to escalate the potential impact even further.

- **The amount of traffic on the affected page**
  The poisoned response will only be served to users who visit the affected page while the cache is poisoned. As a result, the impact can range from non-existent to massive depending on whether the page is popular or not. If an attacker managed to poison a cached response on the home page of a major website, for example, the attack could affect thousands of users without any subsequent interaction from the attacker.

Note that the duration of a cache entry doesn't necessarily affect the impact of web cache poisoning. An attack can usually be scripted in such a way that it re-poisons the cache indefinitely.

Constructing a web cache poisoning attack

Generally speaking, constructing a basic web cache poisoning attack involves the following steps:

1. Identify and evaluate unkeyed inputs
2. Elicit a harmful response from the back-end server
3. Get the response cached

Identify and evaluate unkeyed inputs

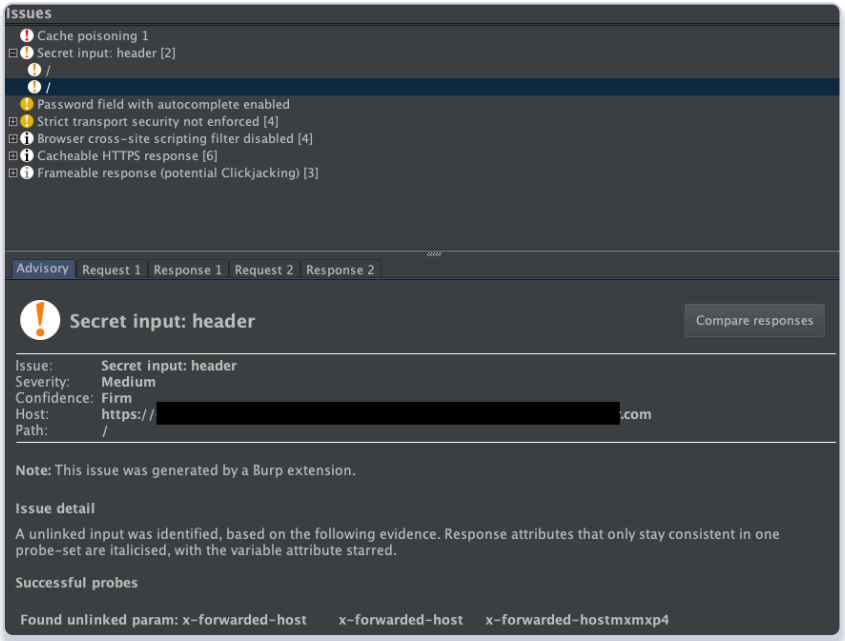
Any web cache poisoning attack relies on manipulation of unkeyed inputs, such as headers. Web caches ignore unkeyed inputs when deciding whether to serve a cached response to the user. This behavior means that you can use them to inject your payload and elicit a "poisoned" response which, if cached, will be served to all users whose requests have the matching cache key. Therefore, the first step when constructing a web cache poisoning attack is identifying unkeyed inputs that are supported by the server.

You can identify unkeyed inputs manually by adding random inputs to requests and observing whether or not they have an effect on the response. This can be obvious, such as reflecting the input in the response directly, or triggering an entirely different response. However, sometimes the effects are more subtle and require a bit of detective work to figure out. You can use tools such as Burp Comparer to compare the response with and without the injected input, but this still involves a significant amount of manual effort.

Param Miner

Fortunately, you can automate the process of identifying unkeyed inputs by adding the Param Miner extension to Burp from the BApp store. To use Param Miner, you simply right-click on a request that you want to investigate and click "Guess headers". Param Miner then runs in the background, sending requests containing different inputs from its extensive, built-in list of headers. If a request containing one of its injected inputs has an effect on the response, Param Miner logs this in Burp, either in the "Issues" pane if you are using Burp Suite Professional, or in the "Output" tab of the extension ("Extensions" > "Installed" > "Param Miner" > "Output") if you are using Burp Suite Community Edition.

For example, in the following screenshot, Param Miner found an unkeyed header `X-Forwarded-Host` on the home page of the website:



Caution: When testing for unkeyed inputs on a live website, there is a risk of inadvertently causing the cache to serve your generated responses to real users. Therefore, it is important to make sure that your requests all have a unique cache key so that they will only be served to you. To do this, you can manually add a cache buster (such as a unique parameter) to the request line each time you make a request. Alternatively, if you are using Param Miner, there are options for automatically adding a cache buster to every request.

Elicit a harmful response from the back-end server

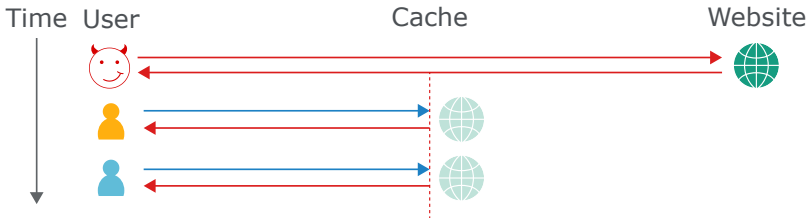
Once you have identified an unkeyed input, the next step is to evaluate exactly how the website processes it. Understanding this is essential to successfully eliciting a harmful response. If an input is reflected in the response from the server without being properly sanitized, or is used to dynamically generate other data, then this is a potential entry point for web cache poisoning.

Get the response cached

Manipulating inputs to elicit a harmful response is half the battle, but it doesn't achieve much unless you can cause the response to be cached, which can sometimes be tricky.

Whether or not a response gets cached can depend on all kinds of factors, such as the file extension, content type, route, status code, and response headers. You will probably need to devote some time to simply playing around with

requests on different pages and studying how the cache behaves. Once you work out how to get a response cached that contains your malicious input, you are ready to deliver the exploit to potential victims.



Exploiting web cache poisoning vulnerabilities

This basic process can be used to discover and exploit a variety of different web cache poisoning vulnerabilities.

In some cases, web cache poisoning vulnerabilities arise due to general flaws in the design of caches. Other times, the way in which a cache is implemented by a specific website can introduce unexpected quirks that can be exploited.

In the following sections, we'll outline some of the most common examples of both of these scenarios. We've also provided a number of interactive labs so that you can see some of these vulnerabilities in action and practice exploiting them.

Read more

Exploiting cache design flaws »
Exploiting cache implementation flaws »

How to prevent web cache poisoning vulnerabilities

The definitive way to prevent web cache poisoning would clearly be to disable caching altogether. While for many websites this might not be a realistic option, in other cases, it might be feasible. For example, if you only use caching because it was switched on by default when you adopted a CDN, it might be worth evaluating whether the default caching options really do reflect your needs.

Even if you do need to use caching, restricting it to purely static responses is also effective, provided you are sufficiently wary about what you class as "static". For instance, make sure that an attacker can't trick the back-end server into retrieving their malicious version of a static resource instead of the genuine one.

This is also related to a wider point about web security. Most websites now incorporate a variety of third-party technologies into both their development processes and day-to-day operations. No matter how robust your own internal security posture may be, as soon as you incorporate third-party technology into your environment, you are relying on its developers also being as security-conscious as you are. On the basis that you are only as secure as your weakest point, it is vital to make sure that you fully understand the security implications of any third-party technology before you integrate it.

Specifically in the context of web cache poisoning, this not only means deciding whether to leave caching switched on by default, but also looking at which headers are supported by your CDN, for example. Several of the web cache poisoning vulnerabilities discussed above are exposed because an attacker is able to manipulate a series of obscure request headers, many of which are entirely unnecessary for the website's functionality. Again, you may be exposing yourself to these kinds of attacks without realizing, purely because you have implemented some technology that supports these unkeyed inputs by default. If a header isn't needed for the site to work, then it should be disabled.

You should also take the following precautions when implementing caching:

- If you are considering excluding something from the cache key for performance reasons, rewrite the request instead.
- Don't accept fat `GET` requests. Be aware that some third-party technologies may permit this by default.
- Patch client-side vulnerabilities even if they seem unexploitable. Some of these vulnerabilities might actually be exploitable due to unpredictable quirks in your cache's behavior. It could be a matter of time before someone finds a quirk, whether it be cache-based or otherwise, that makes this vulnerability exploitable.