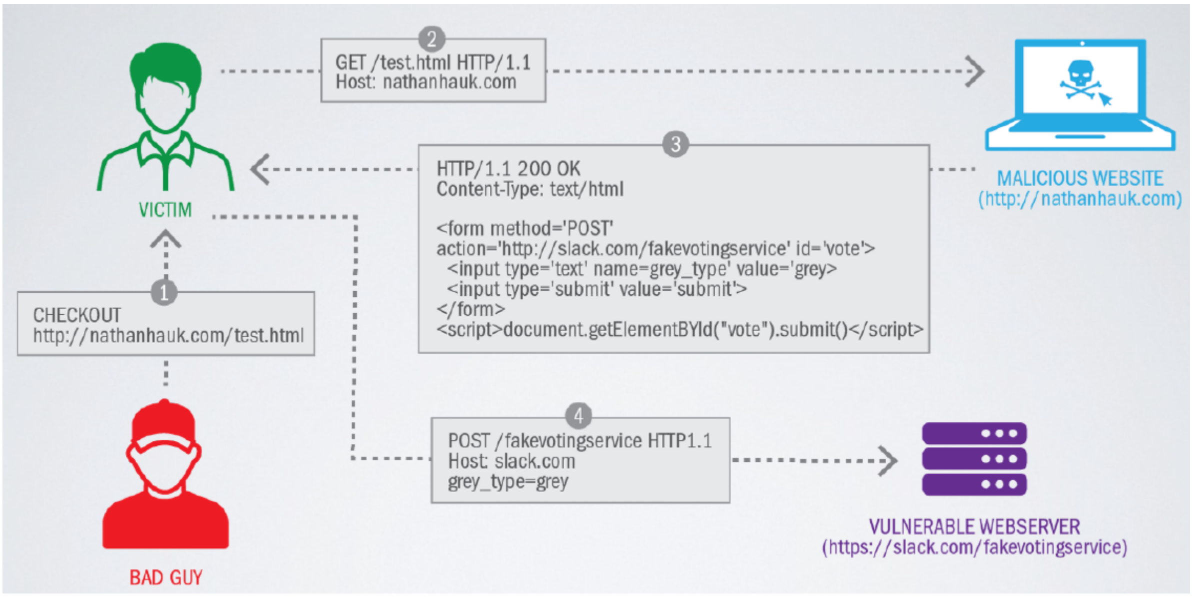
## Cross Site Request Forgery

It is based on the idea of exploiting the trust that a web site has for a user using his session.


VICTIM
1
CHECKOUT
http://nathanhauk.com/test.html
BAD GUY
2
GET /test.html HTTP/1.1
Host: nathanhauk.com
MALICIOUS WEBSITE
(http://nathanhauk.com)
3
HTTP/1.1 200 OK
Content-Type: text/html
<form method='POST' action='http://slack.com/fakevotingservice' id='vote'>
<input type='text' name=grey_type' value='grey>
<input type='submit' value='submit'>
</form>
<script>document.getElementBYId("vote").submit()</script>
4
POST /fakevotingservice HTTP1.1
Host: slack.com
grey_type=grey
VULNERABLE WEBSERVER
(https://slack.com/fakevotingservice)

It works as follows:

- The attacker observes how users interact with the target app, specially how POST interactions are crafted by the server.
- The attacker builds a fake request that mimics a valid user request.
- The attacker sends to the user a fake link that contains a malicious website which embeds the mimic POST request.
- The user will visit the malicious web site that silently submit data to the target application using user's session.
- Now the victim server cannot recognize that the request is fake and it processes it as a normal one.

The problem is that the application blindly trusts every request that comes from the authenticated users. A good practice is that a server must process a submitted POST request only if previously the server served it. In order to mitigate this attack, the server can use a sort of random token on every submission (called anti forgery token). If the token submitted is equal to the one generated previously by the server then it can securely serve the request.

## Content Security Policy

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including XSS. HTTP response header that provides tighter control over which scripts and content will be loaded (and run) on your site. It mitigates XSS and other injection attacks.

CSP makes it possible for server administrators to reduce or eliminate the vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts.

A CSP can be set using an HTTP response header:

```
Content-Security-Policy: policy
```

Alternatively through a metadata in the HTML page:

```
<meta http-equiv="Content-Security-Policy" content="…">
```

A policy is composed by a set of directives:

```
Content-Security-Policy:
    default-src ‘self’;
    img-src *;
    media-src *.media1.com *.media2.com;
    script-src userscripts.example.com
```

The default policy asserts that content is only permitted from the document's origin. An exception is made for images that can be loaded from anywhere. Medias, instead, are allowed only from "media1.com" and "media2.com". Finally, executable scripts are allowed only from "userscripts.example.com".

This approach strongly reduce the possibilities of an attacker to include in a target web page external content.

An interesting feature of CSP is the use of nonce and hashes, a sort of integrity check:

- CSP directive can specify a nonce and the same value must be used in the tag that loads a script. If the values do not match, then the script will not execute. To not be guessable, it must be random for each generated page.
- CSP directive can specify the hash of the contents of trusted script. If the hashes of the actual script and the allowed ones don't match, then the script is not executed. The hash value must be updated if the script changes.

## Access Control Attacks

The Access Controls is a system that enables an authority to control access to areas and resources in a given physical facility or computer-based information system. These systems are critical defense mechanism because they have to decide if an user must be allowed to enter (or use a resource) or not. If something is wrong in its implementation, an attacker can corrupt it and then totally corrupt the rest of the systems/services.

Two different types:

- Vertical Access Controls: allow different types of users to access different parts of the application's functionality. The division is made on functionalities, i.e. each group of users have some permissions of using some restricted functionalities of the application. They are used to enforce the concept separation of duties.
- Horizontal Access Controls: allow users to access a certain subset of a wider range of resources of the same type. E.g. a mail application allows you to read only your email but no one else's.

Obv, Access Controls are broken or corrupted if someone is able to access an unauthorized functionality/resource. There are two different main attacks: Vertical privilege escalation, where an user can perform functions that their assigned role does not permit them to do, or Horizontal privilege escalation, where an user can view or modify resources to which he is not entitled.

Weaknesses:

- Completely unprotected functionality: sometime, some sensitive functionality is left unprotected. Its URL can be guessed and if someone asked for that "hidden" service, its URL

remains in the logs. We cannot base our access control avoiding to tell someone not authorized the URL where the service can be reached.
- Identifier-based functions: when a function of an application is used to retrieve some resource, usually an identifier is passed to it that is associated with the resource itself. So the attacker must know not only the URL but also the identifier of the document he wishes to view. The problem is that resource identifiers are often saved locally by people.
- Multistage functions: involves capturing different items of data from the user at each stage. This data is strictly checked when first submitted and then is usually passed on to each subsequent stage, using hidden fields in an HTML form. Somehow programmers thought that it is sufficient to check only at the first "intended" page the access authorization, thinking that following pages will be called only following a pre defined path. An attacker could exploit this design flaw of the application overcoming so the access control.
- Static files: in some cases, requests for protected resources are made directly to the static resources themselves. E.g. after buying a book, the web site redirect you to the static book available on the web server. There is no check about authorization! An attacker can access it writing down the URL of the static resource in the search bar of a browser. If the naming scheme is discovered by an attacker, whatever static resource is so accessible.

These are best practices to how implement effective access controls in web applications:

- Explicitly document access control requirements for each unit of application functionality. Both users authorized to access the service and the resources that he can access too. Without designing the permissions is quite impossible.
- Rely on user's session to perform an access control decision.
- Use a central application component to check access controls. This allows the application to be scalable and easily to maintain wrt permissions.
- Each request must be processes via the central component in order to decide if it is a valid and legit request.
- Explicitly calls that component in control logic for each page by forcing developers to do that.
- For particularly sensitive functionality, we can use restrictions based on IP address in order to allow only users accessing such functionality from a determined location.
- For a static file we can: either passing the name of the static resource to a server functionality that firstly check for permissions or directly access the static file but including authorization headers in order to be sure that the user has the right permissions to access it.
- Every time the server receives some resource's identifier, it must validate them, because client's data can be tamper while their in transit.
- For critical application functions implements a per-transaction re-authentication and dual authorization to provide additional assurance that the function is not being used by an unauthorized party.
- Log every event for detection intrusion analysis.
- Use layered access control, i.e. each application has its own access control implementation. Don't trust any system-wide security model only.

In order to attack access controls, it may very struggle if an attacker doesn't make progress in a short window of time. But in order to success, he needs to be patient and keep trying on each functionality that each application offers. If a bug is found, then an escalation of permissions could be obtained.

## Protect data in transit

Using only HTTP is very risky, specifically for MITM attacks. Internet is a big problem since it is a public network. The standard solution is to use end-to-end secure channel based on TLS.

HTTPS, indeed, is HTTP on TLS. It guarantees confidentiality, message integrity and authentication if needed. Nowadays, it is used for any web based interaction.

Anti-patterns:

- HTTPS must be used for each critical path, such as for login pages or payment transactions. But unprotected functionalities may leak session id with session sidejacking.
- Mix of HTTPS and HTTP: e.g. loads images with HTTP inside a HTTPS web page. Solution is to use HTTPS everywhere.
- If someone tries to use HTTP, the server can redirect it on the HTTPS version of the web site, performing an upgrade of the protocol.

SSL STRIP ATTACK: the idea is that a HTTPS connection is downgraded to HTTP with MITM attack. The attacker setups a device to intercept connections from his victim (i.e. a proxy). The first HTTP request is intercepted and passed to the server that answer with 301 Redirect. Then the attacker does not forward the response to the client but rather setups the HTTPS connection with the server (exchanging keys). After that, the attacker forwards unprotected data of the server (decrypted correctly) to the user using an unsecure connection through HTTP. Now the attacker can easily see the content of messages exchanged by the two parties.

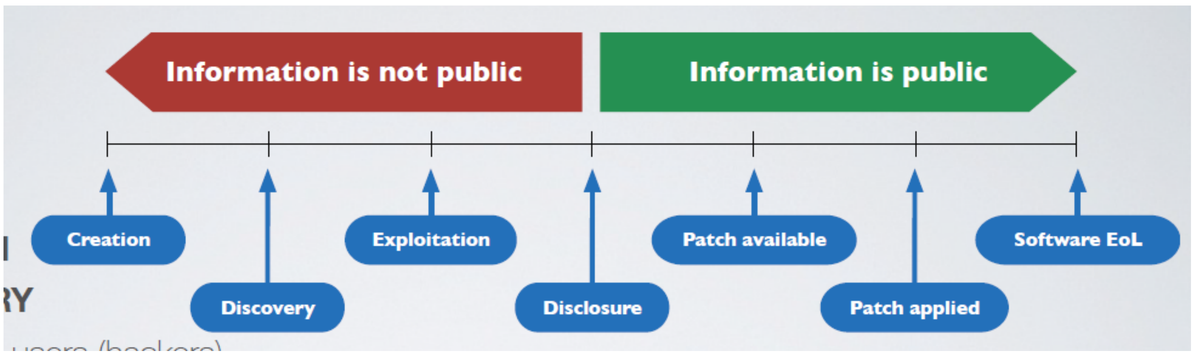
The solution is to enable HTTP Strict Transport Security (HSTS). A web security policy mechanism that helps to protect web sites against SSL stripping attacks and cookie hijacking. It allows web servers to declare that browsers should interact with them only using HTTPS and never with HTTP. Conformant user browsers will automatically redirect any insecure HTTP request to HTTPS for the target web site. The most secure way to do it is to add it on a preloaded list of HTST policies of the browser. This list is loaded from a web site that is updated with the new HTTPS servers of the web, such that there is no need of installed HSTS policy in the first interaction.

## Cost of vulnerabilities

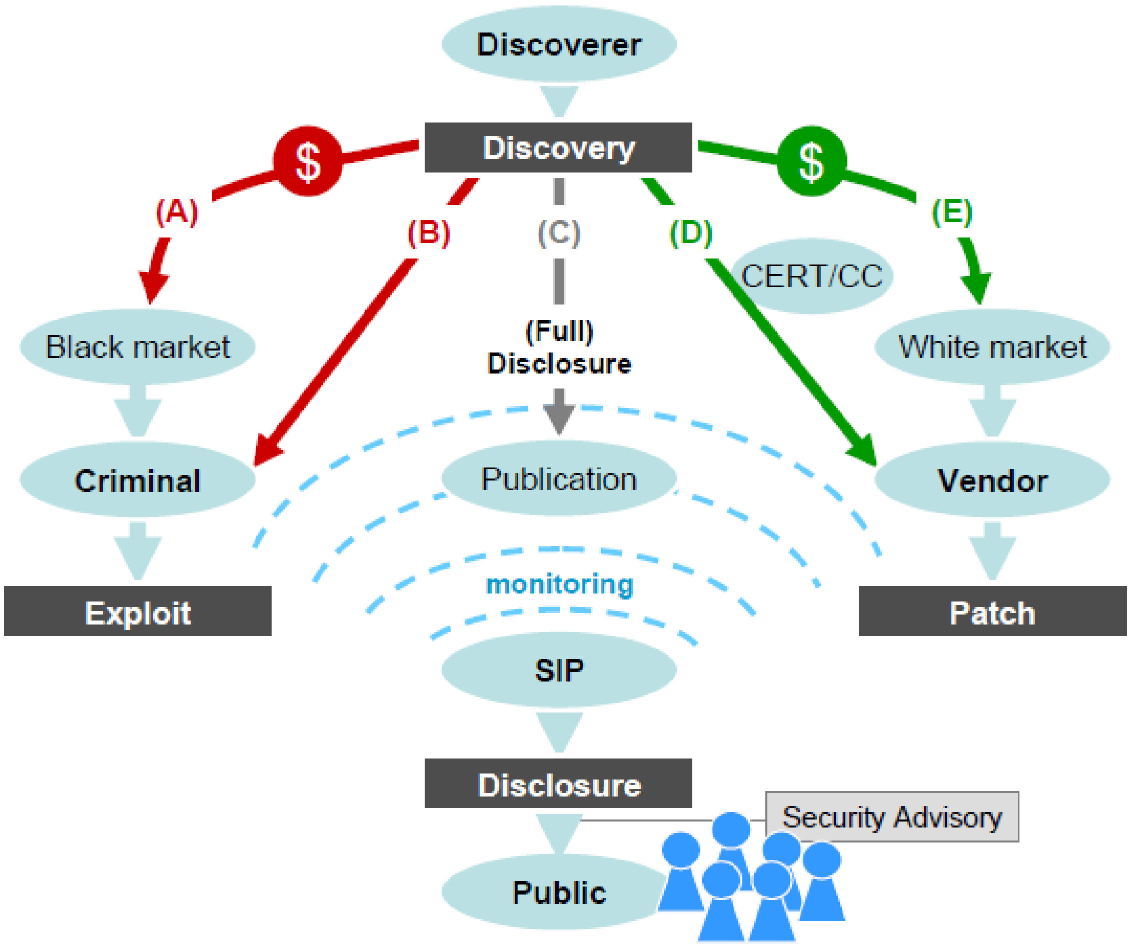
A vulnerability is the intersection of three elements:

- A system has a flaw.
- The attacker finds and accesses to the flaw.
- The attacker is capable to exploit the flaw.

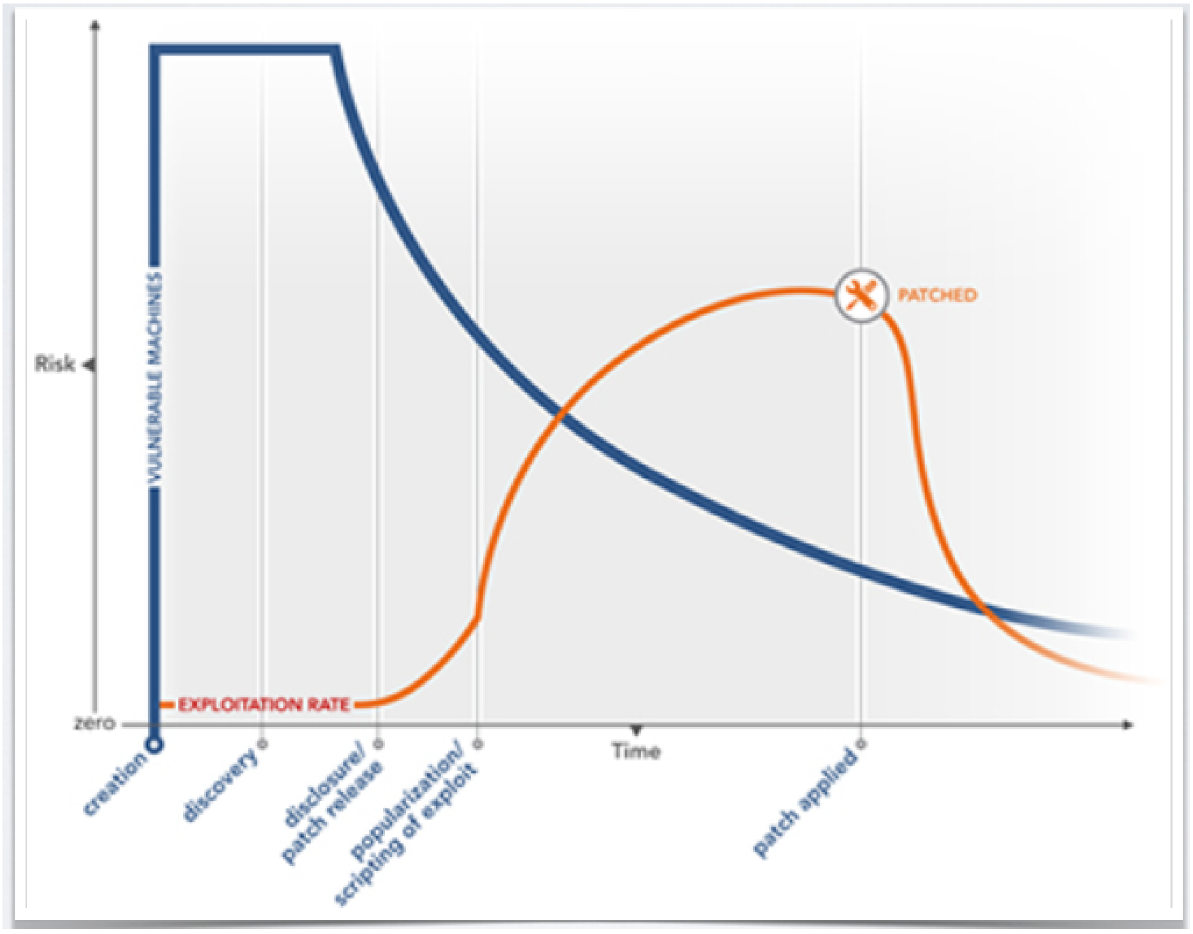
There are a lot of possible causes like bugs, design defects, misconfigurations or aging of libraries used. Also system complexity could be a cause. Fully secure software is very unlikely. The statics say that there are 20 flaws for each 100 lines of code. 95% of flaws can be prevented just keeping updated systems with patches.



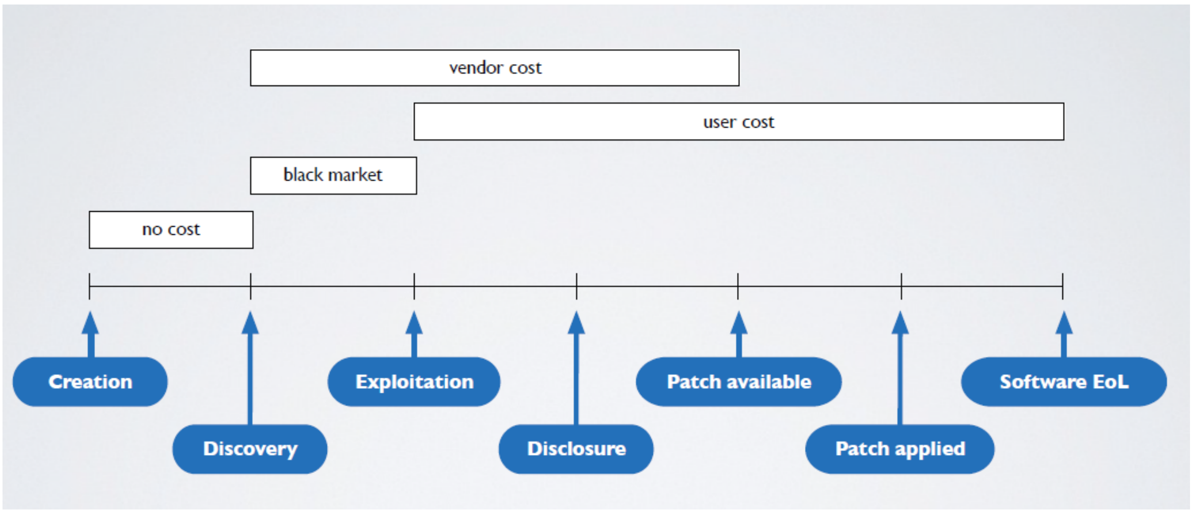
The lifecycle of a vulnerability is characterized by a lot of phases. The discovery phase is a fundamental turning point (described later) because from here on problems arise. If the vulnerability is discovered by a malicious user then exploitation phase comes up with an huge users cost (in term of attacks). If the user is not malicious, the exploitation could be also avoided but costs remain high if the patch is hard to produce. The important point to stress is that until the Disclosure phase, all the information about the vulnerability is not public, i.e. only few persons know about that. After disclosure the information is public, also normal users are aware of the vulnerability. Now on, the software vendor must first of all teach users to assume best practices in order to mitigate the vulnerability and contemporary start to produce a patch, possibly without other bugs, to cancel the vulnerability from the software. In the plot above the vulnerability life ends when the software reaches its End of Life (EoL).



The above picture tells how the discovery process works: a found bug can be sold to the black market in order to be exploited for further attacks, or it can be published with a paper in order to let the community be aware of that (or the producer), notice that before publishing the vulnerability if we contact the producer we must keep secret it until the patch is ready, or, finally, it can be directly sold to the so called white market, i.e. to a company that produces a patch related to that bug and sells more software telling users that ONLY its software is secure wrt to the most recent bug (image bug bounty).



As we can notice from this plot, at the beginning of the vulnerability discovery all the instances of the software are exploitable. Then, before releasing the patch, only making users aware of its existence, the blue curve is decreasing slowly. Then the after publications about the vulnerability the exploitation rate increases dramatically. Only after applying the patch, the exploitation rate decreases.



Before discovery, obv, no costs are present. After that, basing on the kind of discovery, start cost about black market and vendor. After exploitation, user costs arise, since their instances of the software start to be attacked.