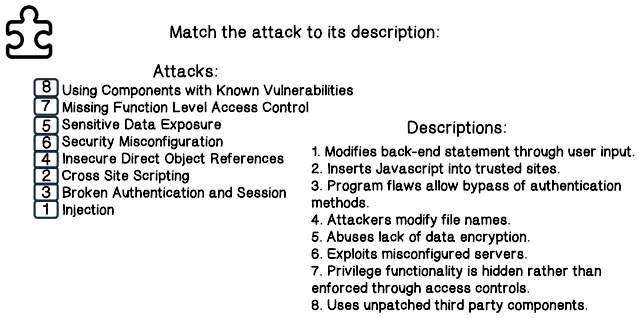
Recommended Readings for this lesson are:

Exposing Private Information by Timing Web Applications

You can find the short summary of the paper at the end of the document

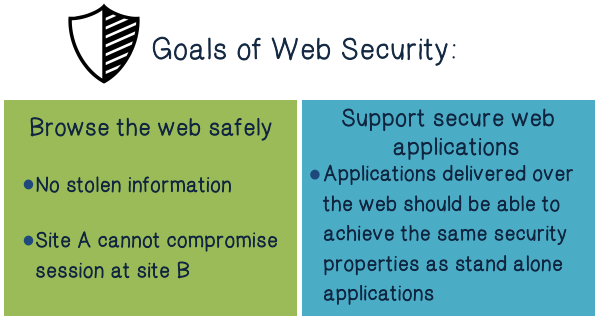
This is a large lesson because the topic, Advanced Web Security, is an expansive subject. By the end of this lesson, you should be familiar with the web security model, defenses against attacks on web applications, HTTPS and its goals and pitfalls, and content security policies and web workers.



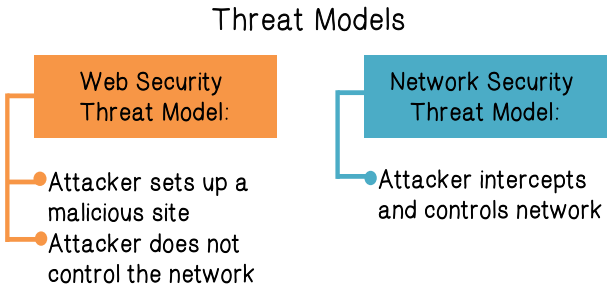
Before we discuss web security, let us remind ourselves as to why we need web security. In this quiz, match the attacks to their descriptions. So the attacks are: using components with known vulnerabilities, missing function level access control, sensitive data exposure, security misconfiguration, insecure direct object references, cross-site scripting, broken authentication and session, and injection. And the descriptions are: modifies back-end statement through user input, inserts Javascripts into trusted sites, program flaws allow bypass of authentication methods, attackers modify file names, abuses the lack of data encryption, exploits misconfigured servers, privilege functionality is hidden rather than enforced through access controls, and uses unpatched third party components.

And the answers are: for the first attack using components with known vulnerabilities, the description is uses unpatched third party components because unpatched third party components have known vulnerabilities. Second, missing function level access control. For this attack, the description is privilege functionality is hidden rather than enforced through access controls because the attack here says it is missing function level access control. The third attack is sensitive data exposure. For that to work, the description is abuses the lack of data encryption. The next attack is security misconfiguration and the description is exploits misconfigured servers. The next attack is insecure direct object references, and the description is that the attacker can modify file names because file names are direct object references. The next attack is cross-site scripting and the description is inserts Javascript into trusted sites. The next one is broken authentication and

session, and the description is program flaws allow bypass of authentication methods because the attack here exploits broken authentication and session. The last one is injection and the description is modifies back-end statement through user input. In other words, the attack action is injected through user input.



Now, let us discuss the goals of web security. Obviously, we need to be able to browse the web safely. This means that, when browsing a website, sensitive data on the user's computer cannot be stolen and uploaded to the web. And that if the web browser has multiple open sessions with multiple sites, for example, one session to a bank website and another session is to a social network site, the sessions do not interfere with each other. Intuitively, if the social network site is compromised, it should not affect the user session with the bank site. In addition, we need to ensure that the web applications can have the same security protection as the other applications that run on our computers.



Now let us discuss the web security Threat Models. We use threat models to understand what the web attackers are likely to do. And we are going to compare the web security threat model and the network security threat model.

On the web, an attacker can typically setup a malicious website and the attacker waits for users to visit the malicious website so that the attack can be launched through the malicious websites to compromise the user's computers. A web attacker typically does not control the network.

Now let us look at the network security threat model. A network attacker can be much more active. Typically, we would assume that a network attacker can intercept and control the network. For example, the attacker can intercept and drop the traffic. Or he can intercept and perform traffic analysis to crack open the encryption key to read the data that is being transmitted, or he can inject malicious traffic into the network.

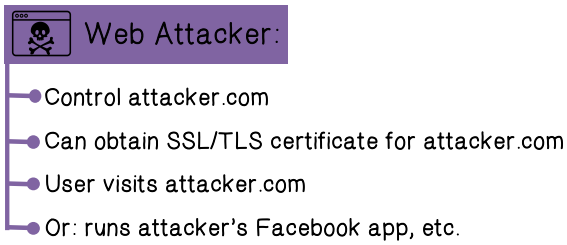
Rank these in order, 1 for the most common, 10 for the least common:

- 5 Security Misconfiguration
- 4 Insecure Direct Object References
- 7 Missing Function Level Access Control
- 6 Sensitive Data Exposure
- 9 Using Components with Known Vulnerabilities
- 3 Cross Site Scripting
- 10 Unvalidated Redirects and Forwards
- 2 Broken Authentication and Session
- 1 Injection
- 8 Cross Site request Forgery

Now let us do a quiz related to web attacks. According to the OWASP in 2013, the following are the top 10 attacks on web security. I would like you to rank them in order, 1 for the most common and 10 for the least common. These attacks are, security misconfiguration, insecure direct object references, missing function level access control, sensitive data exposure, using components with known vulnerabilities, cross site scripting, unvalidated redirects and forwards, broken authentication and session, injection, and cross site request forgery.

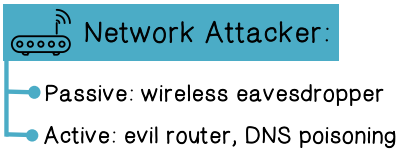
According to OWASP 2013, injection is the most common. Unvalidated redirects and forwards is the least common. And the order of the rest is here.

Web Threat Models



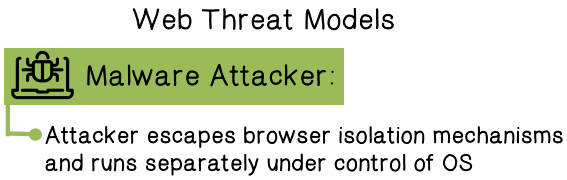
Let us go over the various types of attackers in more details. A web attacker could typically control a suspicious site, say, attacker.com. He can even obtain certificate for his website so that the website can interact with users' browsers through HTTPS. And then the attacker can wait for the user to visit attacker.com. For example, this can be done through phishing and other kinds of redirect. Or, the attacker can set up some sort of malicious, or fake, web app and wait for the user to download these apps and run these apps. The point here is that, typically, a web attacker is somewhat passive. He sets up some attack infrastructure and waits for the users to actually either visit those sites or use those malicious apps.

Web Threat Models

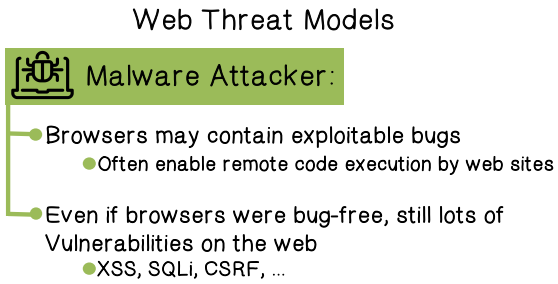


A network attacker is more powerful. He can perform both passive and active attacks. For example, a passive attack means that the attacker simply intercepts and analyze traffic to learn about the communication. For example, the attacker can perform wireless eavesdropping to crack the encryption key for your Wi-Fi network. Examples of active attacks include inserting a malicious router in the network so that traffic can route through the router and be subject to the attacker's attack. That includes both passive attacks like eavesdropping, or active attacks such as traffic injection. Another example is DNS poisoning,

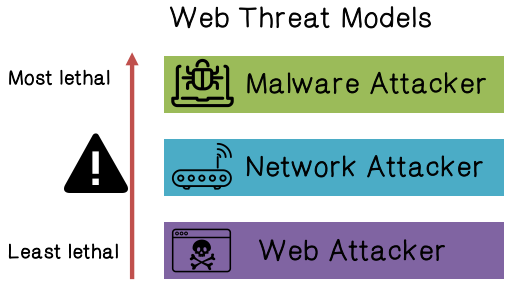
where the attacker changed the DNS entry so that a legitimate site such as cnn.com, now has an IP address of a server that is controlled by the attacker. That is, legitimate traffic such as to cnn.com will not be redirected, so that legitimate traffic such as those to cnn.com will now be redirected to the attacker's machine.



The most general and powerful attack is through malware. By injecting a piece of malware on the user's computer, the attacker essentially escapes the browser's isolation mechanism. And now, the attacker has a program that runs directly under the control of the operating system. That is, the malware runs as any other applications on your computer.

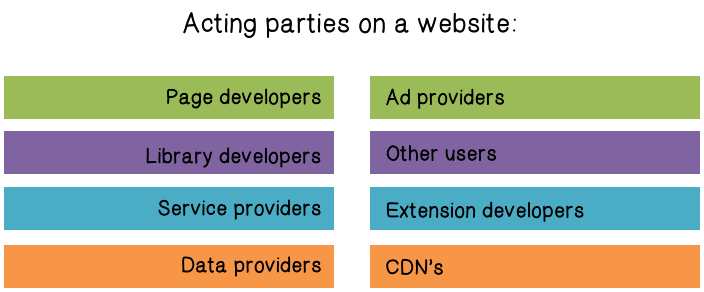


You may ask, why is that possible? Isn't the browser supposed to isolate the rest of the computer from the web? The problem is that browser is a very complex piece of software, and as such, browsers may contain exploitable bugs, and these bugs often enable remote execution of malicious code. For example, when a browser visits a site that is controlled by the attacker, the attacker can send a webpage that contains malicious input. And the result is that a bug is being exploited and a piece of malicious software, or malware, is now installed on a computer. Now, even if the browsers are bug free, there are still lots of vulnerabilities on the web, in particular on the web-server side. That would enable cross-site scripting, SQL injection, and cross-site request forgery. For example, SQL injection would allow the attacker to bypass the control of the web server, and directly inject attackers' code into the backend of the SQL database. The point is that malware attackers can actually bypass the basic control of web, including browser, to actually attack the users' computers or the web service.

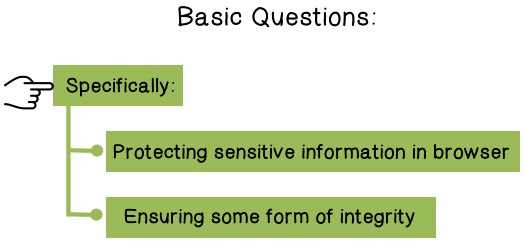


So far we have discussed three main types of attackers. The malware attacker, the network attacker, and the web attacker. It is obvious that a web attacker is the least lethal because he is mostly passive. A network attacker is more powerful because he can perform both passive and active attacks. And a malware attacker is the most lethal and powerful because it can inject code into a user's computer or a server to perform any actions desired by the attacker.

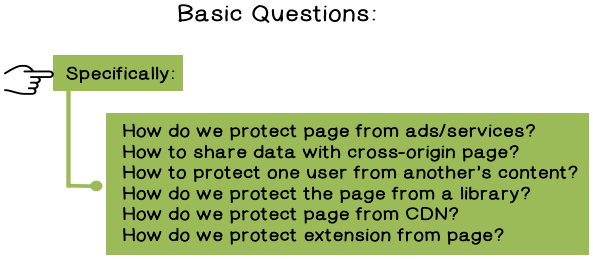
Before we go into the details of web security, let us understand how the modern web works. For a typical website, it contains both static and active contents. The active contents, or the code, can be from many sources and they can be combined in many ways. Then the security challenges are that we have many different types of data and codes for many different sources. And they run and interact with each other. For example, on a typical web page we have code or data related to the page itself, the third-party API's, for example to Twitter, third-party libraries to how you navigate, and scripts that run advertising contents.



And the data and codes on a website can be from many different sources, by many different developers. For example, a website can have many parties contributing to its data and code. These include page developers, library developers, service providers, data providers, ad providers, and other users, and extension developers such as the web app developers, and the CDN's, the content distribution networks. Obviously, these parties can be from different vendors and companies.

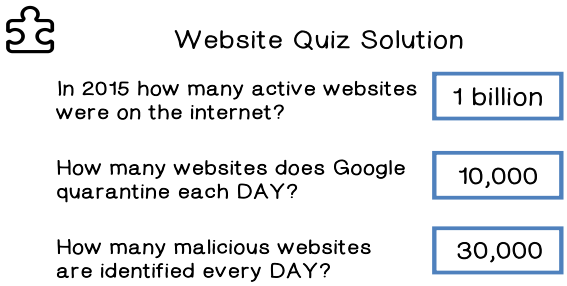


So the basic security questions are with data and codes from so many different sources, how do we ensure data and integrity when we browse the web?



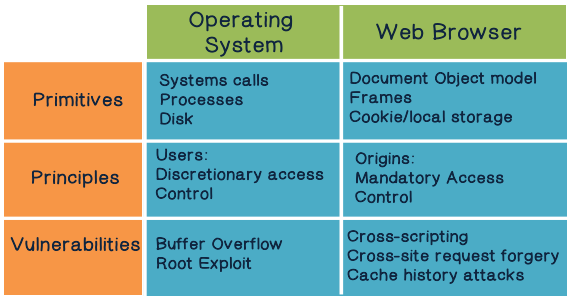
For example, we need to figure out how to protect page from ads and services because they are from different sources. On the other hand, maybe there is a legitimate reason to share data when they are from different sources. That is, how do

we share data with cross-origin page and how do we protect one user from another user's content? How do we protect the page from a third-party library? How do we protect a page from the content distribution network? And how do we protect browser extensions from page?



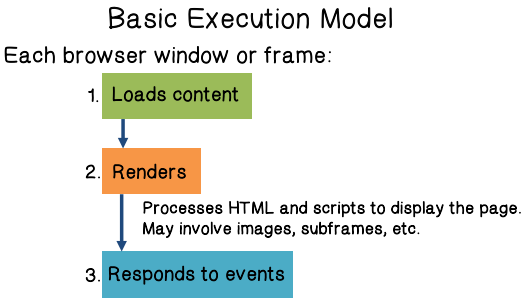
Let us take a moment to understand the enormity of the web security problem. Take your best shot at answering these questions. First, in 2015, how many active websites were on the internet? Second, how many websites does Google quarantine each day? Third, how many malicious websites are identified every day?

The answers are: in 2015, there were 1 billion active websites. And each day, Google quarantines 10,000 websites. And 30,000 malicious websites are identified every day. As you can see, web security is not a small issue. Understanding and stopping malicious actions is paramount to network security.

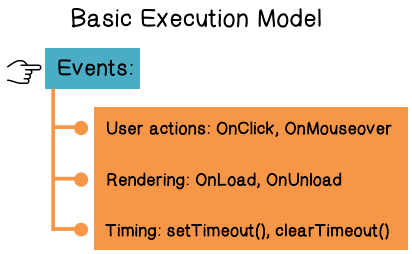


	Operating System	Web Browser
Primitives	Systems calls Processes Disk	Document Object model Frames Cookie/local storage
Principles	Users: Discretionary access Control	Origins: Mandatory Access Control
Vulnerabilities	Buffer Overflow Root Exploit	Cross-scripting Cross-site request forgery Cache history attacks

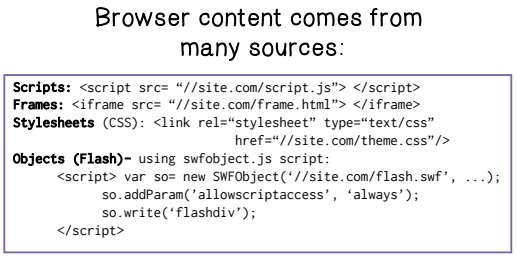
Now, let us discuss browser security model. Let us take a step back and compare operating system with web browser. An operating system supports multiple applications to run on a computer at the same time and allows them to share the resources on a computer. Similarly, a web browser can render multiple webpages to different sites. And each page can contain data and code from multiple sources. So it is instructive to compare the operating system and web browser security models. For operating system, the primitives are system calls, processes, and disk storage. For web browser, the primitives are Document Object Model or DOM, frames, cookies and local storage. The principles on the operating system are users, and associated with users is the discretionary access control policy. For web browser, the principles are origins and mandatory access control is used. Vulnerabilities in operating system can lead to buffer overflow, root exploit and so on. Whereas on web browser, such vulnerabilities can lead to cross-site scripting, cross-site request forgery, cache history attacks, and so on.



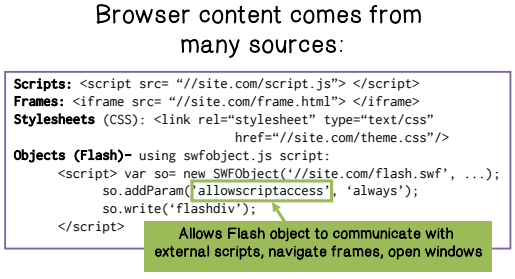
Now let us take a look at the execution model of web browsers. Given a webpage, the browser goes through these steps. First, load the contents. Second, renders the contents. That is, the browser processes the HTML pages and runs each JavaScript to display the contents of the page. The page may include images and frames and so on. And then the browser response to events.



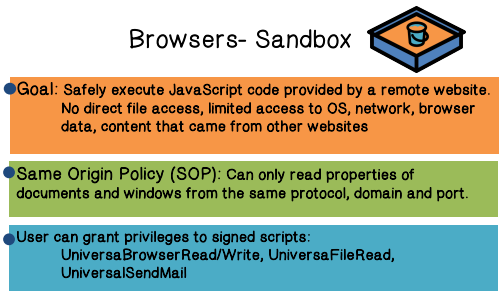
What are the events handled by a web browser? The main events are user actions, such as clicking, moving the mouse, rendering, like loading a page, and timing such as timeout.



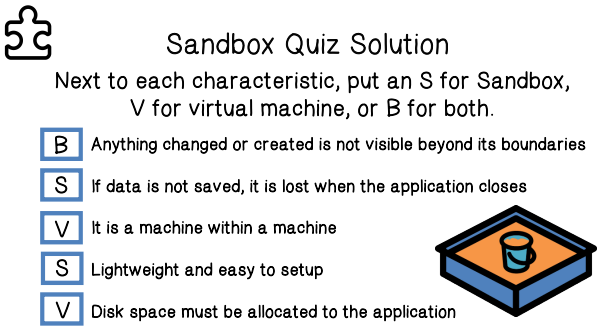
The contents being rendered can be from many sources. For example, you could have scripts, frames loading HTML pages, Flash objects, etc.



By specifying allowscriptaccess, the Flash object can communicate with external data, such as external scripts and navigate external frames and opening windows, etc. The point is that there are many contents from many sources, and they can interact with each other. Obviously, this makes it challenging for enforcing security policies.

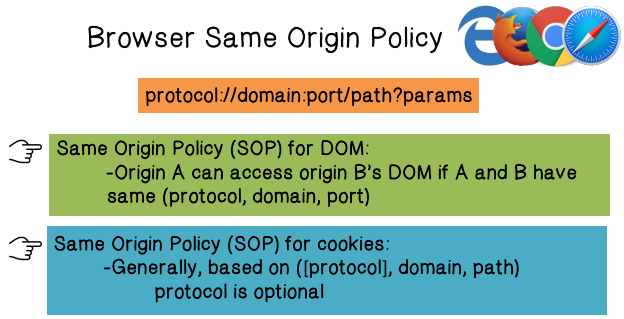


The basic idea of browser security is to sandbox web contents. More specifically, we want to safely execute JavaScript code because it can be from a remote website. This means that a JavaScript code cannot access the file system directly. It can only have limited access to the operating system, the network and browser data, as well as contents from other websites. The main policy is the so-called Same Origin Policy. That means active code, such as JavaScript, can only read properties of documents and windows from the same origin defined as the same protocol, domain, and port. Now, exceptions to this policy can be allowed. That means for scripts that are assigned by legitimate developers that a user can trust, such as scripts signed by Microsoft, Google, Apple, etc., the user can grant privileges such as UniversalBrowserRead/Write, UniversalFileRead, and so on.

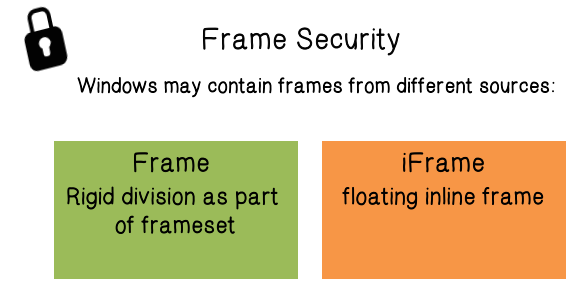


Sandboxes and virtual machines are often confused with one another. Let us use this quiz to try and set the record straight about the two. Next to each characteristic, put an S for Sandbox, V for virtual machine, and B for both. First, anything changed or created is not visible beyond its boundaries. Second, if data is not saved, it is lost when the application closes. Third, it is a machine within a machine. Fourth, lightweight and easy to setup. Fifth, disk space must be allocated to the application.

First, anything changed or created is not visible beyond its boundaries. This can apply to both sandboxes and virtual machines. Sandboxes will isolate applications so that other applications cannot see it. To see changes in virtual machines you must be in the virtual machine. Second, if data is not saved, it is lost when the application closes. This is an advantage of sandbox. And you can call it a security strength of the sandbox because any malware downloaded will not be saved. Third, virtual machines have their own copies of complete operating systems. There can be multiple operating systems on a single hardware platform. Four, sandbox is lightweight and easy to set up. Fifth, for virtual machines, disc space must be allocated to the application.



Origin is defined by protocol, domain, and port. So, the same origin means the same protocol, domain, and port. For document objects or DOM in a browser, the same origin policy says that origin A can access origin B's DOM if A and B have the same origin. Meaning that they have the same protocol, domain, and port. For cookies, we say two cookies have the same origin if they have the same domain and path. The protocol is optional. We are going to discuss in more detail the same origin policy for cookies later.



Frame and iFrame are like many browser windows. A frame is typically rigid or fixed on a page, whereas iFrame can be floating.

Frame Security

iFrame example:

```
<iFrame src='hello.html" width="450"height="100">
</iFrame>
```

Here's an example of iFrame. It essentially says that here is the width and height of the frame window and it will display this page.

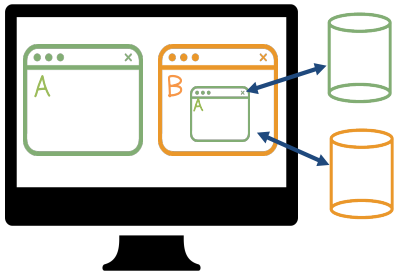
Frame Security

Why use frames?

- Delegate screen area to content from another source
- Browser provides isolation based on frames
- Parent may work even if frame is broken

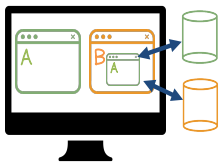
So why do we discuss frames in a context of web security? Or in more general, why do we even use frames? As the previous simple example shows, we can display a webpage within a frame, or a small browser window. So, from this example, it is obvious that frames provide a natural isolation of separation of different web contents. For example, we can delegate screen area to content from another source. And a browser provides isolation based on frames. And, even if a frame is broken, the parent window can still work.

Frame Security



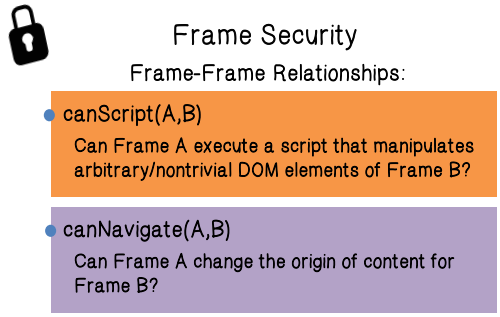
Again, to display web contents from two different sides, A and B, we can have two different browser windows, such as what we see here, A and B. On the other hand, we can achieve the same result by having just one browser window, let us say B here on the right-hand side. And within it, we have a frame that displays contents from A. The point is that we should be able to achieve the same kind of isolation whether we use two different browser windows or use a frame within a window.

Frame Security

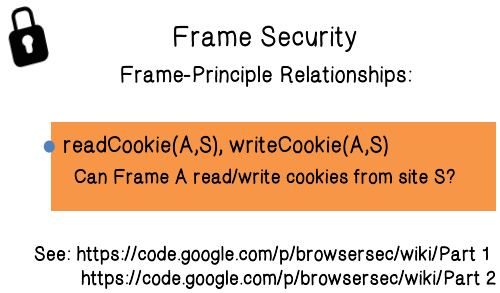


- Each frame of a page has an origin
  Origin= protocol://host:port
- Frame can access its own origin
  Network access, Read/write
  DOM, Storage (cookies)
- Frame cannot access data associated with a different origin

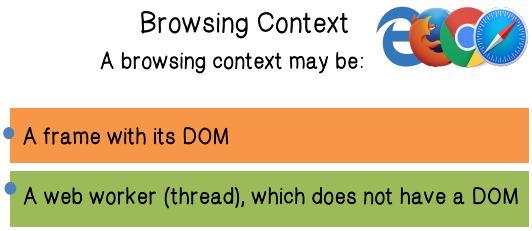
Again, we apply the same origin policy to achieve frame security. Specifically, each frame of a page has an origin, that is defined as protocol, host, and port. A frame can access only the data from its own origin. That is, a frame cannot access data associated with a different origin. Therefore, for example, even though we have a frame within a browser window and they display contents from different sites, for example, A and B. The same-origin policy guarantees that these two sessions, the frame and the browser window, they do not interfere with each other.



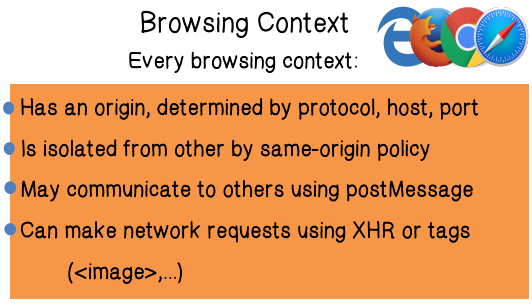
So, there was the default same origin policy. In addition, frame-to-frame access control policy can also be specified. For example, we can say canScript(A,B). That means Frame A can execute a script that manipulates DOM elements of Frame B. We can use canNavigate to specify that Frame A can change the origin of content for Frame B.



Likewise, we can specify policy for frame to access principle. For example, we can use readCookie, writeCookie, to specify that Frame A can read/write cookies from a site. You can read more about the web browser security models by following these links.

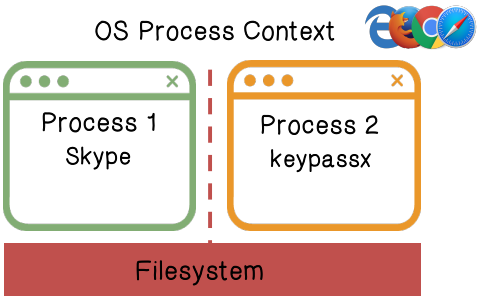


So far we have described the classic web browser security models. To understand the more modern mechanisms, let us define browsing context. A browsing context may be a frame with its DOM, that is, a frame with web contents, or web worker, which does not have a DOM. A web worker as defined by the World Wide Web Consortium or W3C and the Web Hypertext Application Technology Working Group is a Javascript executed from HTML page that runs in the background independently of other user interface scripts that may also have been executed from the same HTML page. In short, a web worker is a Javascript that runs in the background and it is independent of the user interface elements.

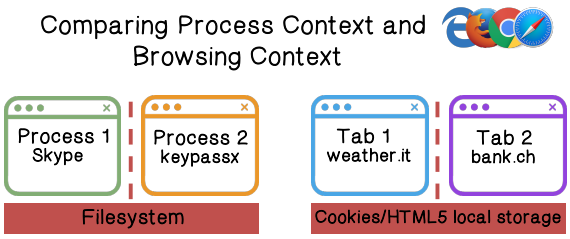


Now, every browsing context has an origin. Again, an origin is determined by protocol, host, and port. And as such, our browsing context is isolated from another context by the same-origin policy. Different browsing contexts may communicate using postMessage. and they can

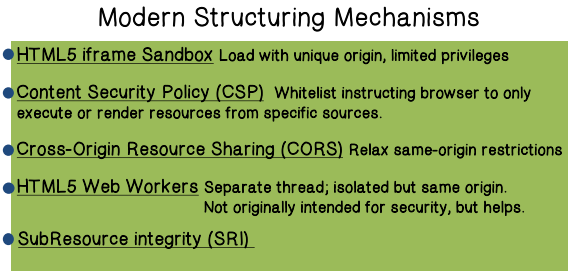
make network requests through XHR or tags. XHR stands for XML HTTP Request. It is an API available to Javascript. Typically, XHR is used to send HTTP or HTTPS requests to a web server. And the server responds data back into the script. That is, a Javascript use XHR to request contents from a web server.



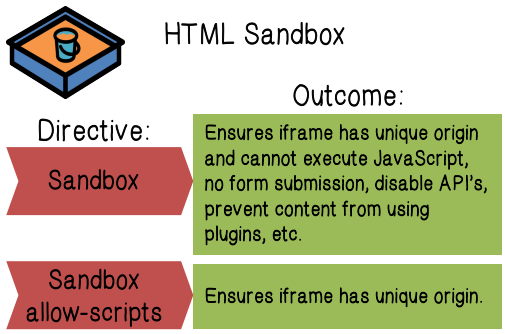
There are similarities between browsing context and process context. An opening system uses separation and isolation to allow multiple execution context and provide local storage and communication services.



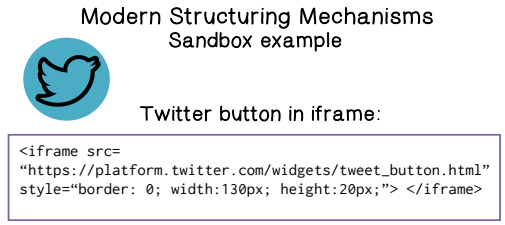
Similarly, while a web browser provides common local storage it uses isolation and separation to provide security protection to the browsing contexts.



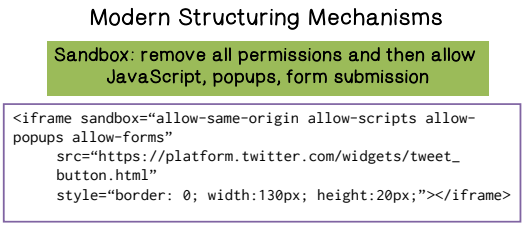
The modern browser mechanisms that can be used for security protection include HTML5 iframe Sandbox, content security policy, cross origin resource sharing, HTML Web Workers, and SubResource integrity. And we are going to discuss these mechanisms now.



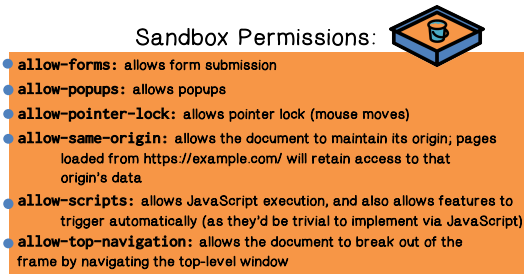
As in operating systems, sandbox is very useful for browser security. The idea is to restrict frame actions. When we use a directive Sandbox for frame essentially we are insuring that the iframe has unique origin, cannot submit forms, and APIs are disabled, and it can prevent contents from plugins. On the other hand when we create iframe if we use the Sandbox allow-scripts directive, then we only ensure that iframe has unique origin, but we allow the rest of the actions.



For example, here is a Twitter button in iframe. In this example, there is no Sandbox related directive. So, this you can call it the classic iframe.

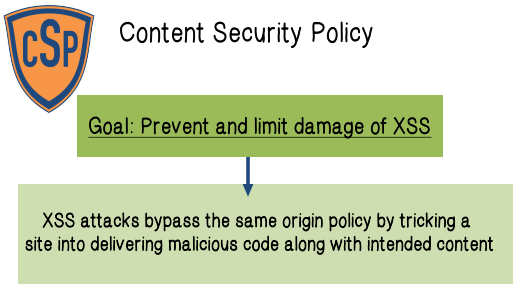


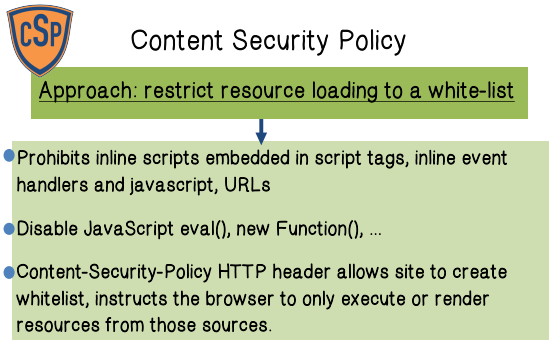
Now we can use a Sandbox directive here. We specify the Sandbox directive. But then we also specify that we will allow Javascripts and allow form submissions and so on. This simple example shows that we can use the Sandbox directive associated with the iframe in order to specify the security policy that is appropriate.



Here are the list of Sandbox permissions that you can specify for iframe.

Now let us discuss content security policy, or, CSP. The goal of content security policy is to prevent or at least limit the damage of cross-site scripting. Recall that we discussed cross-site scripting attacks in CS 6035: Introduction to Information Security. Essentially, a cross-site scripting attack bypasses the same origin policy by tricking a site into delivering some malicious code along with the intended content. For example, a website is setup to echo the user input as a web page back to a browser, such as echoing the user's name. But if the user input contains malicious code, then the website will be sending malicious code to a web browser.

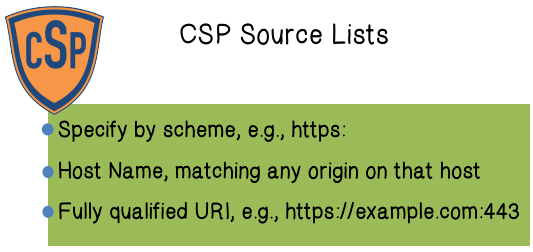




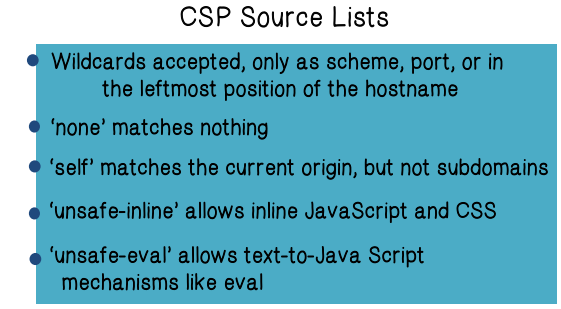
With CSP, the main idea is that a browser can be instructed to load resources only from a whitelist. CSP prohibits inline scripts embedded in script tags, inline event handlers, JavaScript, and URLs, etc., and also disables JavaScript eval, new function and so on. That means all the resources that a browser will load can be statically checked. And again, the resources are loaded only from a whitelist.

Directive	Outcome
script-src	limits the origins for loading scripts
connect-src	limits the origins to which you can connect (via XHR, WebSockets, and EventSource)
font-src	specifies the origins that can serve web fonts
frame-src	lists origins can be embedded as frames
img-src	lists origins from which images can be loaded
media-src	restricts the origins for audio and video
object-src	allows control over Flash, other plugins
style-src	is script-src counterpart for style sheets
default-src	define the defaults for any directive not specified

Since there are many different types of web contents with CSP we can specify the whitelist for each type of web contents.



The sources of web contents can be specified and matched. For example, they can be specified by schemes such as HTTPS or HTTP, host name, then we match any origin on that host or fully qualified URI such as https://example.com:443.



You can also specify how to match the sources listed on a whitelist, such as, wildcards accepted, none, or self, and so on. You can even create exceptions or allow inline JavaScripts or allow eval functions.

CSP Quiz Solution

Which of the following statements are true?

- ☐ If you have third party forum software that has inline script, CSP cannot be used
- ☑ CSP will allow third party widgets (e.g. Google +1 button) to be embedded on your site.
- ☐ For a really secure site, start with allowing everything, then restrict once you know which sources will be used on your site.

Now let us do a quiz on CSP. Which of the following statements are true? First, if you have third party forum software that has inline script, CSP cannot be used. Second, CSP will allow third party widgets, such as Google +1 button, to be embedded on your site. Third, for a really secure site, start with allowing everything, then restrict once you know which sources will be used on your site.

The second statement is true because you can certainly list Google as a trusted source and list it in the whitelist. For the first statement, if you use third party software that has inline script, you can still embed it on your site. You can use script source and style source to allow inline script. For the third statement, for really secure site, it is best to restrict everything. Then once you know which sources will be used, add them to the whitelist.

Web Worker

Run in an isolated thread, loaded from a separate file:

```
var worker – new Worker('task.js');
worker.postMessage(); // Start the worker.
```

Now let us discuss Web Worker. Web Workers were ultimately not intended for security, but they help improve security because they allow JavaScript to run in isolated threads. Here is an example of how you create a Web Worker. Again, it is loaded from JavaScript.

Web Worker

Same origin as frame that creates it, but no DOM

Communicate using postMessage:

main thread

```
var worker – new Worker('doWork.js');
worker.addEventListener('message', function(e) {
      console.log('Worker said: ', e.data);
}, false);
worker.postMessage('Hello World'); //Send data to worker
```

doWork

```
self.addEventListener('message', function(e) {
      self.postMessage(e.data); // Return message it is sent
}, false);
```

A Web Worker has the same origin as the frame that creates it, but the Web Worker has no DOM. It can communicate using postMessage. So here is a simple example. The main thread, meaning the main iframe thread, creates a worker. It then starts the worker thread by sending a message using postMessage. And here the worker actually performs the work.

SubResource Integrity

Many pages pull scripts and styles from a wide variety of service and content delivery networks.

Now let us discuss SubResource Integrity. Integrity is a very important security goal. In the context of web browsing, many pages load scripts and styles from a wide variety of service and content delivery networks. Given that contents can be from many different sources and content delivery networks, how do we ensure the integrity of the contents that we are loading?

SubResource Integrity

How can we protect against:

- Downloading content from a hostile server (via DNS poisoning, or other such means)
- Modified file on the Content Delivery Network (CDN)

For example, how do we protect against loading contents from a malicious server? For example, the browser gets to the malicious server because of DNS poisoning and how do we ensure that contents that we load from a Content Delivery Network have not been modified, for example, on purpose by the CDN?

SubResource Integrity

Idea:
page author specifies has a (sub) resource they are loading; browser checks integrity.

E.G., integrity for scripts:

```
<link rel="stylesheet" href="https://site53.cdn.net/style.
css" integrity="sha256-SDfwewFAE...wefjijfE">
```

The main idea is that the author of the content specifies and makes available the hash of the contents. And so when the browser loads the contents, it use the hash value to check integrity. For example, the author of this stylesheet will specify the hash of the file.

SubResource Integrity

Idea:
page author specifies has a (sub) resource they are loading; browser checks integrity.

E.G., integrity for elements:

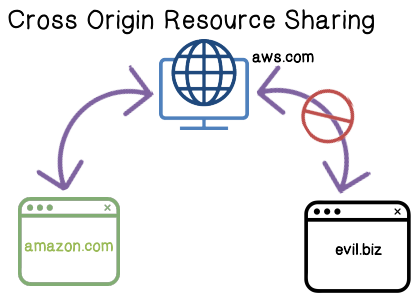
```
<script src= "https://code.jquery.com/jquery-1.10.2.min.
js" integrity= "sha256-
C6CB9UYIS9UJewinPHWTHVqh/E1uhG5Tw+Y5qFQmYg=">
```

Similarly, for JavaScript, the author can also specify its hash. So basically, to use SubResource Integrity, our website author who wishes to include a resource from a third party can specify a cryptographic hash of that resource in addition to the location of the resource. Then when a browser fetches the resource, it can compare the hash provided by the website author with the hash computed from the resource. If the hashes do not match, the resource is discarded.

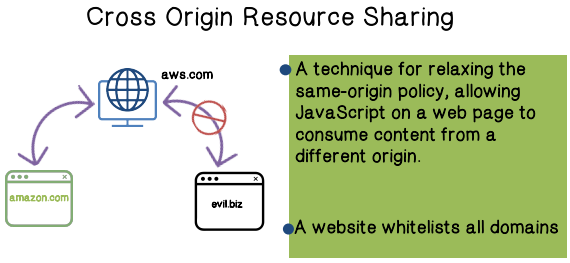
SubResource Integrity:

Case 1 (default)	Case 2
Browser reports violation and does not render/execute resource	CSP directive with integrity-policy directive set to report Browser reports violation, but may render/execute resource

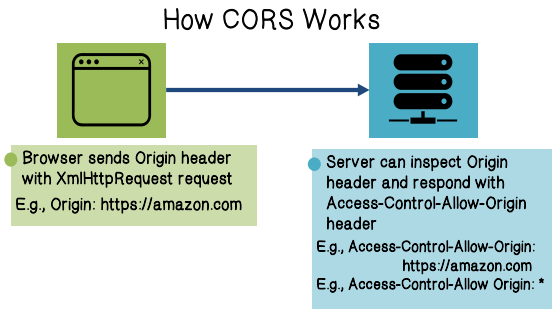
So, what happens when the integrity check fails? By default, the browser can report the violation and simply does not render, or execute the resource. Or if the directive simply says, report, that means the browser will report the violation but can still render or execute the resource.



Now, let us discuss cross origin resource sharing. We have been discussing the same origin policy, which means that cross origin reading and writing is typically not allowed. Now, what happens when a website has multiple domains? For example, Amazon, the company has both the amazon.com and aws.com websites. These two domains belong to the same company, so we expect that they should be able to share some resources. Now of course, we want the same origin policy, so that another analytic website cannot easily access resource from Amazon.



Cross Origin Resource Sharing is a technique that we can use to relax the same-origin policy, so that JavaScript on a web page such as on amazon.com now can consume content from a different origin, say, aws.com. It basically uses a whitelist. For example, amazon.com can list the domains that it allows.



Here is how Cross Origin Resource Sharing works. The browser sends the options request to the origin HTTP header. The value of this header is the domain that served the parent page. For example, when a page from amazon.com attempts to access a user's data in aws.com, the following request header will be sent to aws.com. That is, it specifies origin https://amazon.com. The server can inspect the Origin header and respond whether the access is allowed or not. For example, the server can send back an error page, if the server does not allow the cross-origin request or it can specify which origin is allowed to access. For example, in this case, the origin https://amazon.com is allowed. Or, the server can say that all domains are allowed.

CORS Quiz Solution

Select all the statements that are true:

- [x] CORS allows cross-domain communication from the browser
- [x] CORS requires coordination between the server and client
- [ ] CORS is not widely supported by browsers
- [ ] The CORS header can be used to secure resources on a website

Now, let us do a quiz. Select all statements that are true. First, cross-origin resource sharing allows cross-domain communication from the browser. Second, it requires coordination between the server and client. Third, it is not widely supported by browsers. Fourth, the header can be used to secure resources on a website.

The first two are true. The first statement is false because it is not widely supported by many browsers. The fourth is also false because the cross-origin resource sharing header cannot be used as a substitute for sound security.

SOP Quiz Solution

Recall that a same-origin policy requires requests to access data must be from the same origin.

What is the definition of an origin?

A combination of URI (UniformResource Identifier) scheme, hostname, and port number.

As a quick review let us do a quiz. Recall that a same-origin policy requires that requests to access data must be from the same origin. But what is the definition of an origin?

An origin is the combination of a URI, which stands for UniformResource Identifier scheme, such as HTTP or HTTPS, and hostname, and port number.

SOP Quiz Solution

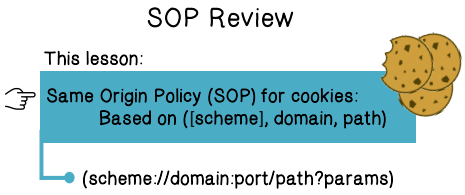
```
https://example.org/absolute/URI/with/absolute/path/to/resource.txt
//example.org/scheme-relative/URI/with/absolute/path/to/resource.txt
/relative/URI/with/absolute/path/to/resource.txt
relative/path/to/resource.txt
../../../resource.txt
./resource.txt#frag01
resource.txt
#frag01
```

Here are some examples of URI references.

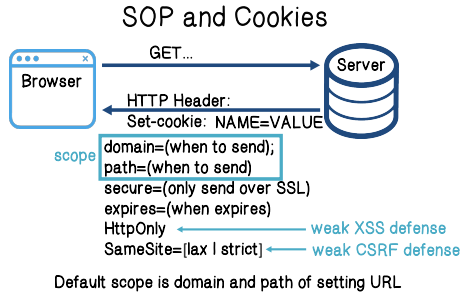
SOP Review

Same Origin Policy (SOP) for DOM:
-Origin A can access origin B's DOM if A and B have same (protocol, domain, port)

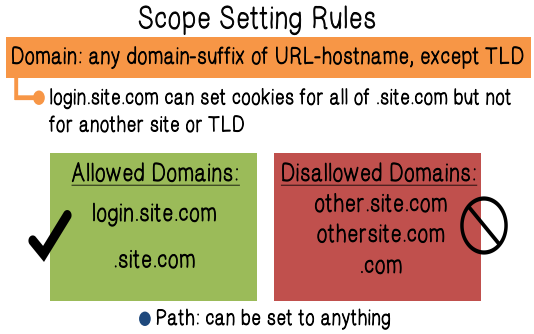
Let us continue with a review of Same Origin Policy. We have discussed the Same Origin Policy for DOM, which stands for Document Object Model. The Same Origin Policy for DOM says that origin A can access origin B's DOM if A and B have the same protocol, domain and port.



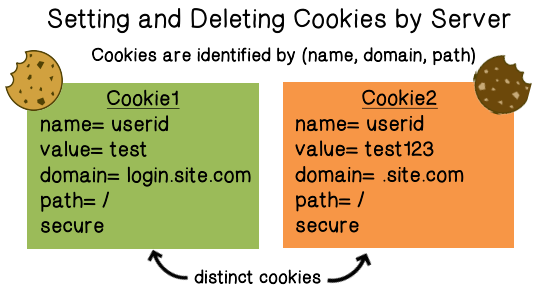
In this lesson, we are going to discuss the Same Origin Policy for cookies. Here, origin is determined by the combination of scheme, domain, and path, and scheme can be optional.



Recall that when a browser connects to a site, the server sets the cookie for the web browsing session. There are a number of attributes that a server can set for a cookie. For example, SameSite means that do not send cookie on a cross-site post request, and strict means that never send cookie on cross-site request. Therefore, they provide some sort of cross-site request forgery defense. With HttpOnly, it tells the browser that this particular cookie should only be assessed by the server. Any attempt to access the cookie from script is strictly forbidden. This can provide defense against cross-site scripting attacks. And the scope of the cookie is determined by the combination of domain and path.

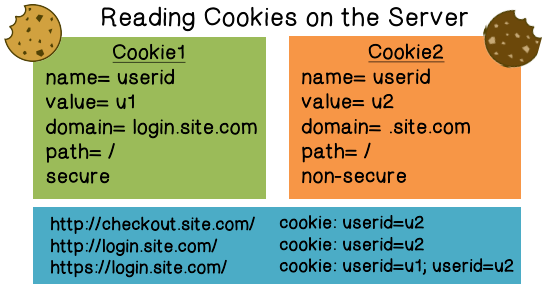


A domain is any domain-suffix of a URL-hostname, except the top level domain. For example, the web server login.site.com can set cookies for all of .site.com because .site.com is a suffix, but not another site or the TLD, which is .com. Using this rule the cookies set by login.site.com have these allowed domains, login.site.com and site.com. And these domains are not allowed because they are other domains or the TLD, .com. And path can be set to anything within that domain.

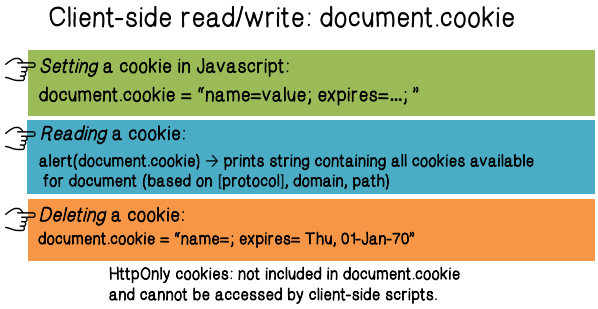


How are the cookies identified? They are identified by name, domain, and path. Here we have two cookies. Both cookies are stored in browser's cookie jar. And both are in scope of login.site.com, but they're distinct.

What are the policies for a server to read cookies? In other words, the reading same origin policy. The browser sends all cookies in URL scope, which is determined by domain and path. And the goal is that server should only see cookies in its own scope.



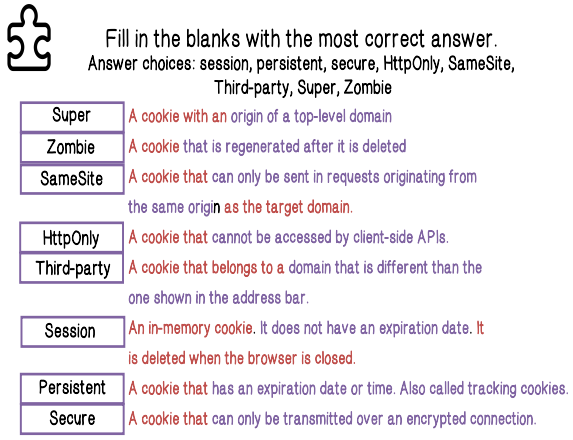
Here is an example. We have two cookies, both set by login.site.com. The different servers see different cookies depending on their scopes. For example the server http://checkout.site.com only sees Cookie2 because it is within the scope of site.com. Another example, http://login.site.com, again, only sees Cookie2. And the reason is Cookie1 requires secure, which means that the connection has to be HTTPS. The third example here, https://login.site.com, it can use both Cookie1 and Cookie2.



What are the rules for client-side read/write of cookies? A JavaScript can set cookie values. It can also read out the attributes of a cookie. It can even delete a cookie. The exception is that if the cookie is set as HttpOnly, that means it cannot be accessed by client-side scripts. Which means client-side scripts cannot read or write this HttpOnly cookie.

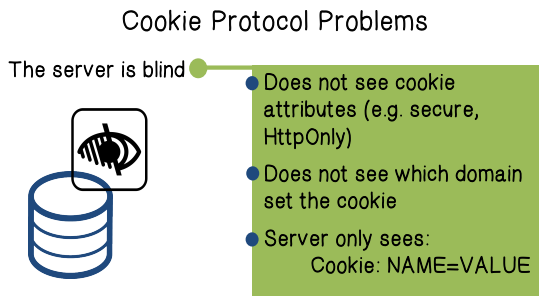
Now let us do a quiz on the same origin policy. Given this website, for the requests that are submitted from the following URLs, which of these URLs will result in a successful request, and which will be rejected as not being from the same origin? Determine the outcome, success or failure, for each URL.

The first three are allowed because they have the same protocol, host and port. The fourth has a different port, port 81, so it is not in the same origin. The fifth has a different host and the sixth has a different protocol.

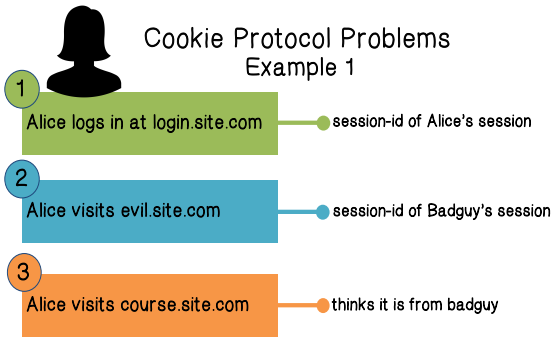


For the following cookies, determine whether they are session cookie, persistent cookie, secure cookie, HttpOnly cookie, SameSite cookie, Third-party cookie, Super cookie, or Zombie cookie.

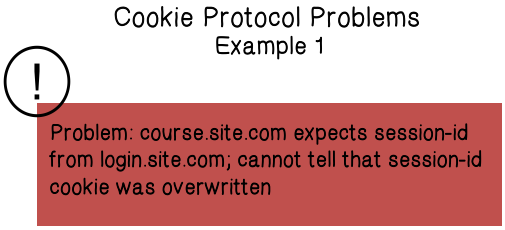
In particular, a cookie that can only be sent in requests originating from the same origin as the target domain is a SameSite cookie. Again, this can be used to defend against cross-site request forgery. And a cookie that cannot be accessed by client-side APIs is the HTTPOnly cookie. It can be used to defend against cross-site scripting attacks.



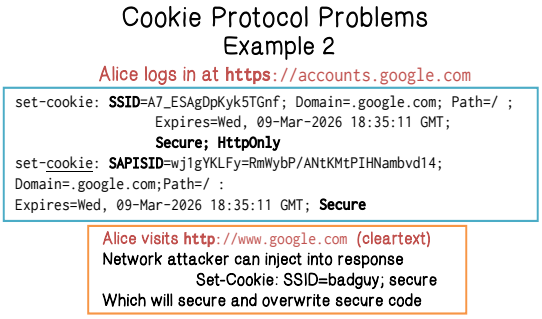
Now let us discuss some security problems with cookies. First of all, the server is blind and what do we mean by that? It does not see all the cookie attributes. For example, whether the cookie attributes include secure, which means HTTPS only, or has the attribute HttpOnly. When a server receives a cookie, it does not see which domain sent the cookie. Actually, all the server sees is some selected attributes sent by the browser.



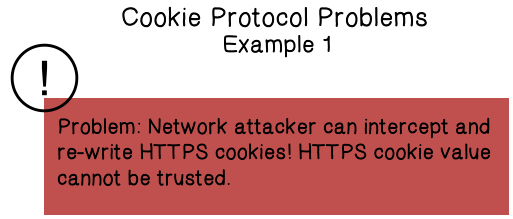
This problem can be exploited by attackers. For example, say Alice wants to submit her homework. She logs in login.site.com and login.site.com sets the session-id cookie for site.com. And then, Alice decides to take a break and unknowingly visits a malicious site, for example, because of a phishing attack. And evil.site.com can override the .site.com session-id cookie with a session-id of user Badguy. Then Alice returns to the homework site ready to turn in her homework, course.site.com thinks that it is talking to the Badguy because the session-id has been overwritten.



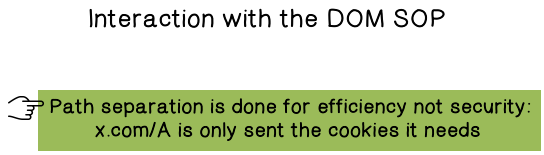
The problem is that course.site.com expects session-id cookie that was set by login.site.com. It cannot tell that the session-id cookie was overwritten.



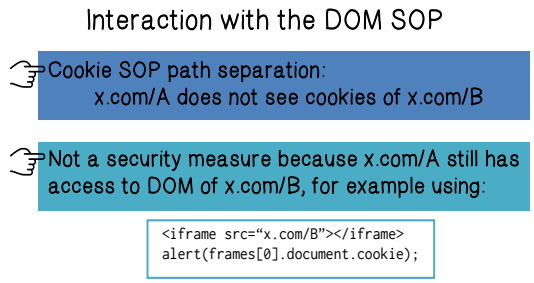
Here is another example of cookie security problems. Suppose Alice logs in https://accounts.google.com, meaning that she logs into her Google account. And accounts.google.com will set the cookie. In particular, it also says that this cookie is Secure, meaning that it should be used for HTTPS. Now, suppose that due to some phishing attack, Alice visits the cleartext site, http://www.google.com, and because this is a cleartext protocol, a network attacker can intercept the traffic and override the cookie attributes. And the result is that this overwritten cookie can be used for a HTTPS session.



As we can see, a network attacker can intercept and rewrite HTTPS cookies, which means that even with a HTTPS cookie, its values cannot be trusted.

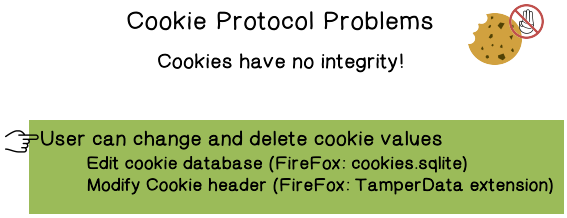


We have not talked about the path of a cookie. The path separation is done only for efficiency, not for security. For example, x.com/A would tell that if a server only needs to access this path, that only this cookie's needed.

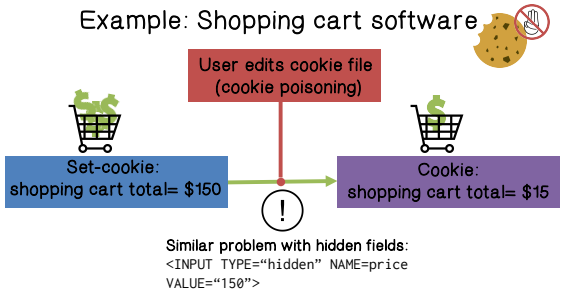


Recall that the scope of a cookie is determined by domain and path. Which means that x.com/A does not see cookies of x.com/B because they are different paths. That is, they are in different scopes. However, this is not a strong security measure because x.com/A still has access to the DOM, meaning the document object model of x.com/B, because they are the same origin as far as DOM is concerned. For example, x.com/A can use the following to print out or read the cookie of x.com/B.

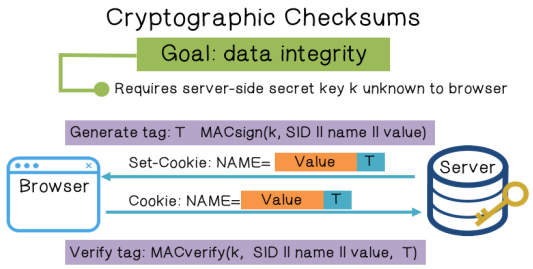
Another security problem of cookies is that cookies have no integrity.



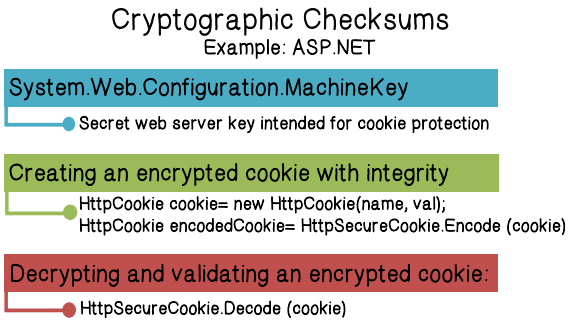
For example, a user can change or even delete cookie values. For example, there are tools that a user can use to change or delete cookie values.



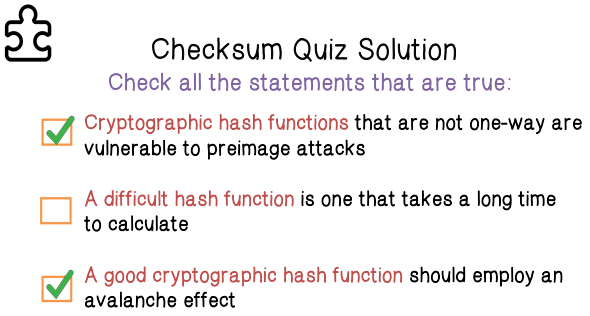
For example, a user can change the shopping cart cookie and change the total dollar amount from $150 to $15. Similarly, if the website had used a hidden field in the webpage to record the value, a user can still edit the source of the page and change the value.



Obviously, we can use cryptography to provide data integrity protection. The main idea is that when a server sets a cookie attribute, it will attach a integrity check value for the attribute, and it can later on check whether that attribute has been modified. So, to do this, the server uses a secret key that is unknown to the browser, and for each attribute value that is set, it computes an integrity check. The courier tag T essentially is a message authentication code, using the secret key k, and computed over the session ID, the name, and value of the attribute. And when it sets the cookie, we attach the message authentication code to each attribute value. When a browser, later on, presents the cookie to a server, the server can then check the integrity of that cookie attribute value. The server essentially uses the secret key and compute over the session ID, nam,e and value of the cookie attribute, and then verify that the result is the same as T. Again, because T is computed using the secret key, the browser cannot compute it. So, that is, only the server can compute T, and the server can use T to verify that the attribute value of the cookie is not changed.



Here is an example of how this can be done in the real world. So, a server key can be generated and the integrity of a cookie can be protected using this key. Similarly, integrity can be tracked. Here are the example APIs that you can use to provide cookie integrity protection.



Now let us do a quick review quiz on cryptographic checksum. Check all the statements that are true. First, cryptographic hash functions that are not one-way are vulnerable to preimage attacks. Second, a difficult hash function is one that takes a long time to calculate. Third, a good cryptographic hash function should employ an avalanching effect.

The first and third statements are true. The second statement is false. A difficult hash function should be very hard for the attackers to analyze. But we want all the hash function to be efficient to calculate

Exposing Private Information by Timing Web Applications

We show that the time web sites take to respond to HTTP requests can leak private information, using two different types of attacks. The first, direct timing, directly measures response times from a web site to expose private information such as validity of an username at a secured site or the number of private photos in a publicly viewable gallery. The second, cross-site timing, enables a malicious web site to obtain information from the user's perspective at another site. For example, a malicious site can learn if the user is currently logged in at a victim site and, in some cases, the number of objects in the user's shopping cart. Our experiments suggest that these timing vulnerabilities are wide-spread. We explain in detail how and why these attacks work, and discuss methods for writing web application code that resists these attacks.

This paper discusses a pervasive bug in web application software. The fact that timing data at many web sites leaks private information suggests that this side channel is often ignored by web developers. We presented a number of direct and indirect measurement techniques that can effectively exploit real-world leaks of private information, including a new cross-site timing method that can reveal private user state. While a difficult problem to solve, one approach to fixing these vulnerabilities is carefully controlling the time taken to respond to any request, either through careful server-side coding or a web server module that automatically regulates the time at which responses are sent.