Programmation orientée sécurité : quelques éléments de réponse

Olivier Levillain

4INFOS8ProSec @ INSA 2018-2019

Avant-propos

- Les supports de cours seront disponibles sur `http://paperstreet.picty.org/POS/`
- En cas de question, n'hésitez-pas
  - pendant le cours, à m'interrompre,
  - plus tard, à m'envoyer un courrier électronique à `cours-POS@picty.org`

Bonnes pratiques

- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Bonnes pratiques

- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Bonnes pratiques
Nettoyage des entrées utilisateur
Architecture logicielle
Tester le positif et le négatif
Améliorer la qualité du code
Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Le danger des entrées non filtrées

Plusieurs exemples vus dans les cours passées

- les injections SQL
- les injections *shell*
- les injections diverses et variées
- les débordements de tampon
  - en lecture (*Heartbleed*) : fuite d'information
  - en écriture : modification d'une variable locale
  - en écriture : détournement du flot de contrôle

Comment contrer les injections ? (1/2)

- Interdire les caractères *spéciaux*
  - exemples en PHP pour les injection SQL :

Comment contrer les injections ? (1/2)

- Interdire les caractères *spéciaux*
  - exemples en PHP pour les injection SQL :
  - `mysql_escape_string` (obsolète)

Comment contrer les injections ? (1/2)

- Interdire les caractères *spéciaux*
  - exemples en PHP pour les injection SQL :
  - `mysql_escape_string` (obsolète)
  - `addslashes` (ne couvre que les guillemets et les apostrophes)

Comment contrer les injections ? (1/2)

- Interdire les caractères *spéciaux*
  - exemples en PHP pour les injection SQL :
  - `mysql_escape_string` (obsolète)
  - `addslashes` (ne couvre que les guillemets et les apostrophes)
  - `mysqli_real_escape_string` (couvre-t-elle réellement tous les cas, y compris les encodages alternatifs comme `\u0022` ?)

Comment contrer les injections ? (1/2)

- Interdire les caractères *spéciaux*
  - exemples en PHP pour les injection SQL :
  - `mysql_escape_string` (obsolète)
  - `addslashes` (ne couvre que les guillemets et les apostrophes)
  - `mysqli_real_escape_string` (couvre-t-elle réellement tous les cas, y compris les encodages alternatifs comme `\u0022` ?)
  - la liste noire peut marcher, mais il y a mieux

Comment contrer les injections ? (1/2)

- Interdire les caractères *spéciaux*
  - exemples en PHP pour les injection SQL :
  - `mysql_escape_string` (obsolète)
  - `addslashes` (ne couvre que les guillemets et les apostrophes)
  - `mysqli_real_escape_string` (couvre-t-elle réellement tous les cas, y compris les encodages alternatifs comme `\u0022` ?)
  - la liste noire peut marcher, mais il y a mieux

- Utiliser les *prepared statements*, qui permettent de conserver la structure d'une requête
  - pour SQL, on trouve des fonctions `prepare()` et `bind_param()`
  - en *shell*, il faut éviter `system` et `popen` (qui appellent un nouvel interpréteur) et préférer appeler directement `exec*`

Comment contrer les injections ? (2/2)

- Dans certains cas, on peut vérifier l'entrée utilisateur en liste blanche

Comment contrer les injections ? (2/2)

- Dans certains cas, on peut vérifier l'entrée utilisateur en liste blanche
- Exemple avec la lecture d'un code postal français (5 chiffres)

Comment contrer les injections ? (2/2)

- Dans certains cas, on peut vérifier l'entrée utilisateur en liste blanche
- Exemple avec la lecture d'un code postal français (5 chiffres)
- En Perl, on peut même forcer le développeur à *faire quelque chose* avec les entrées avant de les utiliser (*taint checking*)

```
# reading the first argument of the script
# arg1 is tainted
my $arg1 = shift;
if ($arg1 =~ /^(\d{5})$/) {
    # regular expression matching words of exactly
    # 5 decimal digits
    $arg1 = $1;
    ... # it is possible to use $arg1
} else {
    ... # cleaning up the tainted variable failed
    ... # the variable can not be used
}
```

Protection contre les *buffer overflow* (1/3)

Pendant le développement

Protection contre les *buffer overflow* (1/3)

Pendant le développement

- éviter les fonctions `strcpy`, `strcat`
- leur préférer des fonctions moins dangereuses (`strncpy`, `strncat`, `snprintf`...)

Protection contre les *buffer overflow* (1/3)

Pendant le développement

- éviter les fonctions `strcpy`, `strcat`
- leur préférer des fonctions moins dangereuses (`strncpy`, `strncat`, `snprintf`...)

- vérifier la taille des paramètres reçus
- vérifier la cohérence entre les différents paramètres reçus (en particulier les champs taille)

Protection contre les *buffer overflow* (2/3)

Mécanismes au niveau de l'environnement de compilation ou d'exécution

Protection contre les *buffer overflow* (2/3)

Mécanismes au niveau de l'environnement de compilation ou d'exécution

- la pile non exécutable
  - de manière générale, appliquer le `W xor X`
  - mis en œuvre par l'environnement d'exécution
  - a priori compatible avec la majorité des programmes

Protection contre les *buffer overflow* (2/3)

Mécanismes au niveau de l'environnement de compilation ou d'exécution

- la pile non exécutable
  - de manière générale, appliquer le `W xor X`
  - mis en œuvre par l'environnement d'exécution
  - a priori compatible avec la majorité des programmes

- la randomisation mémoire
  - mis en œuvre par l'environnement d'exécution
  - nécessite la recompilation de toutes les bibliothèques pour une réelle efficacité

Protection contre les *buffer overflow* (2/3)

Mécanismes au niveau de l'environnement de compilation ou d'exécution

- la pile non exécutable
  - de manière générale, appliquer le `W xor X`
  - mis en œuvre par l'environnement d'exécution
  - a priori compatible avec la majorité des programmes
- la randomisation mémoire
  - mis en œuvre par l'environnement d'exécution
  - nécessite la recompilation de toutes les bibliothèques pour une réelle efficacité
- les canaris et le réarrangement de la pile
  - protection appliquée à la compilation (`-fstack-protector-all` pour `gcc`)
  - peu de problème de compatibilité
  - coût à l'exécution parfois important

Protection contre les *buffer overflow* (3/3)

Attention, les *buffer overflow* dans la pile ne sont pas la seule corruption mémoire possible :

- débordement de tampon dans le tas
- *double-free*
- *use-after-free*
- ...

Bonnes pratiques
- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Gestion de privilèges (1/2)

Comment concilier dans un programme

- des traitements complexes sur des données non maîtrisées et
- des opérations nécessitant des privilèges ?

Gestion de privilèges (1/2)

Comment concilier dans un programme

- des traitements complexes sur des données non maîtrisées et
- des opérations nécessitant des privilèges ?

Exemple : un serveur `mail`

Gestion de privilèges (2/2)

Une méthode classique est de séparer le programme en plusieurs processus :

Gestion de privilèges (2/2)

Une méthode classique est de séparer le programme en plusieurs processus :

- les processus traitant les données complexes tournent sous une identité non privilégiée (et peuvent de plus être restreints, voir plus loin)

Gestion de privilèges (2/2)

Une méthode classique est de séparer le programme en plusieurs processus :

- les processus traitant les données complexes tournent sous une identité non privilégiée (et peuvent de plus être restreints, voir plus loin)
- un processus privilégié persiste et évite tout traitement complexe

Gestion de privilèges (2/2)

Une méthode classique est de séparer le programme en plusieurs processus :

- les processus traitant les données complexes tournent sous une identité non privilégiée (et peuvent de plus être restreints, voir plus loin)
- un processus privilégié persiste et évite tout traitement complexe
- entre les deux, un canal de communication utilisant un protocole *simple*

Isolation/cloisonnement des processus

Des mécanismes pour restreindre les capacités d'un processus [1]

1. Certains exemples sont spécifiques à Linux

Isolation/cloisonnement des processus

Des mécanismes pour restreindre les capacités d'un processus [1]

- `chroot` pour restreindre la visibilité sur le système de fichiers
- les *mount namespaces* permettent la même chose de manière plus robuste

1. Certains exemples sont spécifiques à Linux

Isolation/cloisonnement des processus

Des mécanismes pour restreindre les capacités d'un processus [1]

- `chroot` pour restreindre la visibilité sur le système de fichiers
- les *mount namespaces* permettent la même chose de manière plus robuste

- les *PID namespace* pour restreindre la visibilité sur les processus

1. Certains exemples sont spécifiques à Linux

Isolation/cloisonnement des processus

Des mécanismes pour restreindre les capacités d'un processus [1]

- `chroot` pour restreindre la visibilité sur le système de fichiers
- les *mount namespaces* permettent la même chose de manière plus robuste
- les *PID namespace* pour restreindre la visibilité sur les processus
- les (*net namespace*) pour limiter les capacités réseau

1. Certains exemples sont spécifiques à Linux

Isolation/cloisonnement des processus

Des mécanismes pour restreindre les capacités d'un processus [1]

- `chroot` pour restreindre la visibilité sur le système de fichiers
- les *mount namespaces* permettent la même chose de manière plus robuste
- les *PID namespace* pour restreindre la visibilité sur les processus
- les (*net namespace*) pour limiter les capacités réseau
- divers mécanismes pour limiter les capacités d'un processus
  - `rlimit` (limitations sur les ressources)
  - `seccomp` (restriction des appels système)
  - SELinux, AppArmor ou RBAC GRsec

1. Certains exemples sont spécifiques à Linux

Exemples de telles architecture

- `postfix` : un serveur mail avec une myriade de processus, tournant chacun avec les droits qui lui sont strictement nécessaires
- OpenSSH met également en œuvre de la séparation de privilèges
- Chrome : le navigateur de Google qui met en œuvre de nombreux mécanismes de cloisonnement

Bonnes pratiques
- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Démarche classique de tests

- Mise en place de tests représentatifs du comportement attendu de l'application
- Parfois, les tests sont écrits avant le code (*test-driven development*)

Démarche classique de tests

- Mise en place de tests représentatifs du comportement attendu de l'application
- Parfois, les tests sont écrits avant le code (*test-driven development*)

- De manière générale, l'idée est que les tests représentent la « spécification fonctionnelle »

Démarche classique de tests

- Mise en place de tests représentatifs du comportement attendu de l'application
- Parfois, les tests sont écrits avant le code (*test-driven development*)

- De manière générale, l'idée est que les tests représentent la « spécification fonctionnelle »

- Une classe de test répandue est le test unitaire
- Une méthodologie pertinente est d'enrichir une base de tests de non-régression : à chaque problème corrigé, on écrit un test détectant ce problème.

Généraliser les tests *négatifs*

- Au-delà du test fonctionnel (ce qui marche)
- Il faut tester ce qui doit échouer

Généraliser les tests *négatifs*

- Au-delà du test fonctionnel (ce qui marche)
- Il faut tester ce qui doit échouer

- L'absence totale de cette démarche s'est vue dans le cas d'erreurs d'implémentations TLS
  - Goto Fail Apple
  - Goto Fail GnuTLS
  - Erreurs de padding à la POODLE sur TLS
  - Erreurs sur la vérification de MAC par certaines implémentations (les motifs d'intégrité ne sont que partiellement vérifiés)

Généraliser les tests *négatifs*

- Au-delà du test fonctionnel (ce qui marche)
- Il faut tester ce qui doit échouer

- L'absence totale de cette démarche s'est vue dans le cas d'erreurs d'implémentations TLS
  - Goto Fail Apple
  - Goto Fail GnuTLS
  - Erreurs de padding à la POODLE sur TLS
  - Erreurs sur la vérification de MAC par certaines implémentations (les motifs d'intégrité ne sont que partiellement vérifiés)

- Le *fuzzing* peut être vu comme une forme de tests de sécurité

Bonnes pratiques
- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Lisibilité du code

De manière générale, améliorer la qualité du code permet une meilleure maintenance dans le temps

- Expliciter l'implicite (éviter les paramètres par défaut)
- Documenter ce qui doit l'être
- Éviter d'utiliser des constructions complexes du langage
- Faire relire son code...
- ... et écouter son relecteur !

Lisibilité du code

De manière générale, améliorer la qualité du code permet une meilleure maintenance dans le temps

- Expliciter l'implicite (éviter les paramètres par défaut)
- Documenter ce qui doit l'être
- Éviter d'utiliser des constructions complexes du langage
- Faire relire son code...
- ... et écouter son relecteur !

Avez-vous déjà relu votre propre code 2 mois ou 2 ans après l'avoir écrit ?

Encapsulation

Bien qu'ils n'apportent pas de garanties robustes, les mécanismes d'encapsulation (modules, espaces de nom, modificateurs d'accès) permettent d'éviter certaines erreurs.

- Une interface bien pensée restreint la visibilité des champs internes
- Ainsi, il est plus difficile de se tromper en utilisant un tel module
- Cela permet de garantir (en fonctionnement normal) certains invariants
- Il reste important de vérifier ces invariants de manière défensive

Typage/annotations

Une façon de contraindre ce que peut faire le programme est d'utiliser un langage typé, ou de jouer sur des annotations (`const` en C par exemple).

- Le typage agit comme une forme de documentation toujours à jour
- Dans certains langages fonctionnels, le *pattern matching* exhaustif permet de garantir une bonne gestion de tous les cas
- Attention à bien en connaître les limites dans certains langages

Utilisation des outils standard

La plupart des compilateurs peuvent réaliser des analyses complémentaires et aider le développeur à rester dans le droit chemin.

- variables/fonctions déclarées mais non utilisées
- conversions sauvages
- variables masquées
- mauvaise utilisation des formats dans `printf`

Il faut utiliser les avertissements et les modes stricts offerts

Bonnes pratiques
- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Un mot sur l'offuscation (1/2)

L'offuscation (ou obscurcissement) de code a pour vocation de masquer le fonctionnement interne d'un programme.

- Le réel intérêt est souvent la protection de la propriété intellectuelle

Un mot sur l'offuscation (1/2)

L'offuscation (ou obscurcissement) de code a pour vocation de masquer le fonctionnement interne d'un programme.

- Le réel intérêt est souvent la protection de la propriété intellectuelle
- L'intérêt pour la sécurité est moins clair : il s'agit surtout de ralentir l'attaquant
- On préférera plutôt des méthodes de *diversification* pour éviter qu'un même code d'exploitation fonctionne de manière identique sur plusieurs machines

Un mot sur l'offuscation (1/2)

L'offuscation (ou obscurcissement) de code a pour vocation de masquer le fonctionnement interne d'un programme.

- Le réel intérêt est souvent la protection de la propriété intellectuelle
- L'intérêt pour la sécurité est moins clair : il s'agit surtout de ralentir l'attaquant
- On préférera plutôt des méthodes de *diversification* pour éviter qu'un même code d'exploitation fonctionne de manière identique sur plusieurs machines

- L'offuscation (et la diversification) ont un coût sur les possibilités de *debug*
- Si l'offuscation est réalisée « à la main », le code risque de contenir plus d'erreur et d'être moins facile à maintenir

Un mot sur l'offuscation (2/2)

L'offuscation de code n'est **pas** en général une bonne chose pour la sécurité du programme.

Si vous devez vraiment en faire usage, utilisez des outils (é)prouvés pour automatiser les transformations !

En particulier, évitez les transformations allant à l'encontre de la lisibilité ou de la sécurité (code auto-modifiant)

Bonnes pratiques
- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Développement et cryptographie

Développer du code cryptographique est très difficile

- il faut comprendre les difficultés liées à l'écriture de code crypto (importance du temps, gestion de l'aléa)
- créer son cryptosystème est souvent voué à l'échec (n'importe qui peut créer un algorithme qu'il ne sait pas casser)
- même des professionnels du domaine se trompent régulièrement

Développement et cryptographie

Développer du code cryptographique est très difficile

- il faut comprendre les difficultés liées à l'écriture de code crypto (importance du temps, gestion de l'aléa)
- créer son cryptosystème est souvent voué à l'échec (n'importe qui peut créer un algorithme qu'il ne sait pas casser)
- même des professionnels du domaine se trompent régulièrement

Utiliser du code cryptographique est difficile

- sans tout comprendre, il faut appréhender les hypothèses (sur l'aléa, sur les clés, etc.)
- les interfaces des bibliothèques sont souvent complexes et parfois dangereuses

Gestion de l'aléa

- En cas de défaillance du générateur d'aléa, la crypto tombe
  - avec DSA et ECDSA, la clé privée est compromise
  - pour RSA, l'impact n'est grave qu'à la génération
  - dans de nombreux protocoles, les garanties s'effondrent en l'absence d'aléa

Gestion de l'aléa

- En cas de défaillance du générateur d'aléa, la crypto tombe
  - avec DSA et ECDSA, la clé privée est compromise
  - pour RSA, l'impact n'est grave qu'à la génération
  - dans de nombreux protocoles, les garanties s'effondrent en l'absence d'aléa
- Difficulté de générer du bon aléa
  - en général, il *faut* se reposer sur le système d'exploitation (`/dev/urandom`)

Gestion de l'aléa

- En cas de défaillance du générateur d'aléa, la crypto tombe
  - avec DSA et ECDSA, la clé privée est compromise
  - pour RSA, l'impact n'est grave qu'à la génération
  - dans de nombreux protocoles, les garanties s'effondrent en l'absence d'aléa
- Difficulté de générer du bon aléa
  - en général, il *faut* se reposer sur le système d'exploitation (`/dev/urandom`)
- Lien dangereux entre gestion d'aléa et multiprocessus
  - *bug* dans Android
  - vulnérabilités classiques dans le cadre de la virtualisation

Timing attacks

- Exemple vu précédemment sur le problème de comparer une chaîne en temps variable
- Même quand la variation est faible, on peut généralement exploiter la faille en récoltant plus d'échantillons
- Exemple de faille réelle mettant en œuvre une telle vulnérabilité : Lucky 13

Autres points délicats en cryptographie

- les besoins en confidentialité des clés privées
- les besoins en intégrité des clés publiques
- certaines hypothèses concernant les IV/nonces/aléas/etc.
- l'utilisation de bons paramètres

Bonnes pratiques
- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

Attention aux règles arbitraires

Est-il pertinent que votre code contienne 30 % de commentaires ?

Attention aux règles arbitraires

Est-il pertinent que votre code contienne 30 % de commentaires ?

Que pensez-vous du bout de code suivant ?

```
i++   // Incrementing i
```

Où placer des commentaires

Deux endroits pertinents

Où placer des commentaires

Deux endroits pertinents

- les interfaces
  - description d'une fonction
  - présentation d'un module

Où placer des commentaires

Deux endroits pertinents

- les interfaces
  - description d'une fonction
  - présentation d'un module
- les points durs de votre code
  - tout morceau de code que vous n'arrivez pas à relire d'un coup d'oeil mérite d'être réécrit ou commenté
  - algorithme difficile
  - astuce pour améliorer les performances
  - etc.

Bonnes pratiques

- Nettoyage des entrées utilisateur
- Architecture logicielle
- Tester le positif et le négatif
- Améliorer la qualité du code
- Offuscation de code ?

Quelques mots sur la cryptographie

Quelques mots sur les commentaires

La connaissance des constructions d'un langage/paradigme

OWASP

`https://www.owasp.org/index.php/Top_10_2013-Top_10`

Dans le contexte du web,

OWASP

`https://www.owasp.org/index.php/Top_10_2013-Top_10`

Dans le contexte du web,

- avantages et inconvénients des vérifications sur les entrées utilisateur côté client ou côté serveur ?

OWASP

`https://www.owasp.org/index.php/Top_10_2013-Top_10`

Dans le contexte du web,

- avantages et inconvénients des vérifications sur les entrées utilisateur côté client ou côté serveur ?
- que doit-on considérer comme une entrée utilisateur : les paramètres d'une URL, les champs d'un formulaire envoyés par la méthode POST, les *cookies* ?

Programmation objet : l'exemple Java

Règle classique : utiliser des accesseurs et rendre tous les champs « privés »

Programmation objet : l'exemple Java

Règle classique : utiliser des accesseurs et rendre tous les champs « privés »

Il est cependant essentiel de comprendre à quoi servent les règles pour les appliquer de manière intelligente.

Illustration.

Programmation objet : l'exemple Java

Règle classique : utiliser des accesseurs et rendre tous les champs « privés »

Il est cependant essentiel de comprendre à quoi servent les règles pour les appliquer de manière intelligente.

Illustration.

Attention cependant à bien comprendre la signification de `private` !

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

- `s1 == s2` en C ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes

Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes

Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?
- `s1 == s2` en Java ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?
- `s1 == s2` en Java ?
  - comment exprimer l'égalité qui vous intéresse ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?
- `s1 == s2` en Java ?
  - comment exprimer l'égalité qui vous intéresse ?
- `l1 = l2` en OCaml ?
- `l1 == l2` en OCaml ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?
- `s1 == s2` en Java ?
  - comment exprimer l'égalité qui vous intéresse ?
- `l1 = l2` en OCaml ?
- `l1 == l2` en OCaml ?
- `s1 == s2` en PHP ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?
- `s1 == s2` en Java ?
  - comment exprimer l'égalité qui vous intéresse ?
- `l1 = l2` en OCaml ?
- `l1 == l2` en OCaml ?
- `s1 == s2` en PHP ?
  - un indice chez vous : si `s1` vaut `"0x10"` et `s2` vaut `"16"`, l'expression est vraie ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?
- `s1 == s2` en Java ?
  - comment exprimer l'égalité qui vous intéresse ?
- `l1 = l2` en OCaml ?
- `l1 == l2` en OCaml ?
- `s1 == s2` en PHP ?
  - un indice chez vous : si `s1` vaut `"0x10"` et `s2` vaut `"16"`, l'expression est vraie ?
- `s1 === s2` en PHP ?

Quid de l'égalité ?

On note `s` les chaînes de caractères et `l` les listes
Que signifient les expressions suivantes ?

- `s1 == s2` en C ?
  - comment exprimer l'égalité qui vous intéresse ?
- `s1 == s2` en Java ?
  - comment exprimer l'égalité qui vous intéresse ?
- `l1 = l2` en OCaml ?
- `l1 == l2` en OCaml ?
- `s1 == s2` en PHP ?
  - un indice chez vous : si `s1` vaut `"0x10"` et `s2` vaut `"16"`, l'expression est vraie ?
- `s1 === s2` en PHP ?
- `d1 == d2` en Python (où `d1` et `d2` sont des dates) ?

Questions

?