Programmation orientée sécurité : vulnérabilités classiques

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`

Le développement sécurisé : un vaste problème

Il existe une grande diversité de catégories de vulnérabilités

- débordements de tampons (*buffer overflow*)
- débordements d'entiers (*integer overflow*)
- injections variées dues un manque d'assainissement des données en entrée

- mauvaise configuration des permissions
- utilisation dangereuse de répertoires partagés (`/tmp` par ex.)
- utilisation de mauvaises valeurs par défaut

- confusion dans l'interprétation de données
- erreurs de logique dans le traitement des erreurs

Comment détecter et se protéger des vulnérabilités

Il existe plusieurs approches

Comment détecter et se protéger des vulnérabilités

Il existe plusieurs approches

- *Find and Patch* : stratégie réactive uniquement
  - stratégie la plus utilisée
  - en l'absence de détection, une faille n'est jamais corrigée
  - le déploiement d'un *patch* peut être long

Comment détecter et se protéger des vulnérabilités

Il existe plusieurs approches

- *Find and Patch* : stratégie réactive uniquement
  - stratégie la plus utilisée
  - en l'absence de détection, une faille n'est jamais corrigée
  - le déploiement d'un *patch* peut être long

- prouver formellement du code
  - les garanties apportées sont très fortes
  - cela nécessite des développeurs très qualifiés et du temps
  - il peut être très difficile de prouver certaines propriétés
  - cela nécessite souvent l'approche *clean slate*...

Comment détecter et se protéger des vulnérabilités

Il existe plusieurs approches

- *Find and Patch* : stratégie réactive uniquement
  - stratégie la plus utilisée
  - en l'absence de détection, une faille n'est jamais corrigée
  - le déploiement d'un *patch* peut être long
- prouver formellement du code
  - les garanties apportées sont très fortes
  - cela nécessite des développeurs très qualifiés et du temps
  - il peut être très difficile de prouver certaines propriétés
  - cela nécessite souvent l'approche *clean slate*...
- auditer le code

Comment détecter et se protéger des vulnérabilités

Il existe plusieurs approches

- *Find and Patch* : stratégie réactive uniquement
  - stratégie la plus utilisée
  - en l'absence de détection, une faille n'est jamais corrigée
  - le déploiement d'un *patch* peut être long

- prouver formellement du code
  - les garanties apportées sont très fortes
  - cela nécessite des développeurs très qualifiés et du temps
  - il peut être très difficile de prouver certaines propriétés
  - cela nécessite souvent l'approche *clean slate*...
- auditer le code

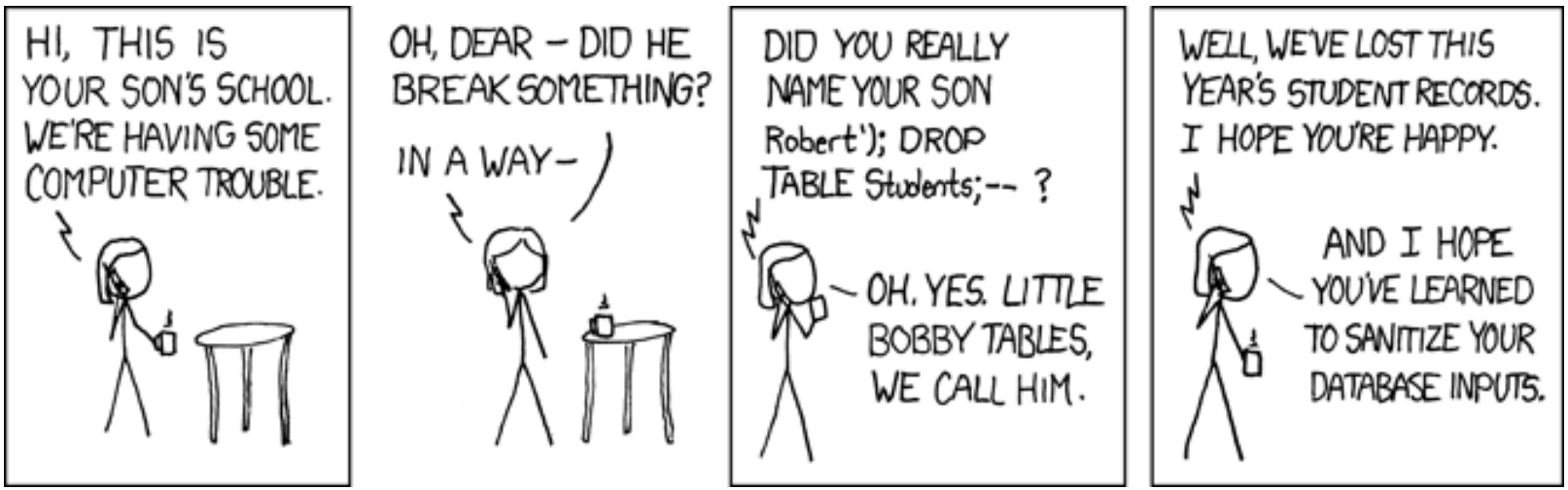
- entre les deux extrêmes, le durcissement
  - stratégie de défense en profondeur
  - le coût peut être dur à justifier
  - bien qu'efficace, tout ne peut pas être couvert ainsi

Au menu

Présentations de quelques vulnérabilités classiques

- injections
- erreurs de logique
- *buffer overflows*
- *confused deputy*

Connaissez-vous Bobby Tables ?



Source : `http://xkcd.com/327`

SQL Injections 101

Cas d'école

- une application web prenant en entrée
  - `$USER` le nom de l'utilisateur
  - `$PASS` son mot de passe

SQL Injections 101

Cas d'école

- une application web prenant en entrée
  - `$USER` le nom de l'utilisateur
  - `$PASS` son mot de passe
- la vérification est faite auprès d'une base de données SQL

```
$query = 'SELECT * FROM user ' .
         'WHERE username = "' . $USER . '" '
         'AND password = "' . $PASS . '"';
$result = sql_query (query);
if ($result) { /* Authentication OK */ }
```

SQL Injections 101

Cas d'école

- une application web prenant en entrée
  - `$USER` le nom de l'utilisateur
  - `$PASS` son mot de passe
- la vérification est faite auprès d'une base de données SQL

```
$query = 'SELECT * FROM user ' .
         'WHERE username = "' . $USER . '" '
         'AND password = "' . $PASS . '"';
$result = sql_query (query);
if ($result) { /* Authentication OK */ }
```

Explication de texte

- dès que la requête renvoie un résultat, on passe l'authentification
- que se passe-t-il si `$PASS` contient des guillemets ?

Réflexions sur les injections

Concept généralisable

- dès qu'un programme/script utilise une chaîne de caractères pour transmettre une structure complexe
- il y a possibilité de confusion si les caractères décrivant la structure (ici les guillemets) peuvent être contrôlés par l'attaquant

Réflexions sur les injections

Concept généralisable

- dès qu'un programme/script utilise une chaîne de caractères pour transmettre une structure complexe
- il y a possibilité de confusion si les caractères décrivant la structure (ici les guillemets) peuvent être contrôlés par l'attaquant

- plus bas niveau, on peut rencontrer des problèmes similaires (*packets in packets*)

Réflexions sur les injections

Concept généralisable

- dès qu'un programme/script utilise une chaîne de caractères pour transmettre une structure complexe
- il y a possibilité de confusion si les caractères décrivant la structure (ici les guillemets) peuvent être contrôlés par l'attaquant

- plus bas niveau, on peut rencontrer des problèmes similaires (*packets in packets*)

Contre-mesures classiques

- échapper tous les caractères de structure

Réflexions sur les injections

Concept généralisable

- dès qu'un programme/script utilise une chaîne de caractères pour transmettre une structure complexe
- il y a possibilité de confusion si les caractères décrivant la structure (ici les guillemets) peuvent être contrôlés par l'attaquant

- plus bas niveau, on peut rencontrer des problèmes similaires (*packets in packets*)

Contre-mesures classiques

- échapper tous les caractères de structure
  - l'approche en liste noire ne marche généralement pas

Réflexions sur les injections

Concept généralisable

- dès qu'un programme/script utilise une chaîne de caractères pour transmettre une structure complexe
- il y a possibilité de confusion si les caractères décrivant la structure (ici les guillemets) peuvent être contrôlés par l'attaquant

- plus bas niveau, on peut rencontrer des problèmes similaires (*packets in packets*)

Contre-mesures classiques

- échapper tous les caractères de structure
  - l'approche en liste noire ne marche généralement pas
- conserver la structure lors du passage (requêtes préparées)

Application réelle de ce *bug*

CVE-2010-3088

- `pidgin-knotify`
- `command = g_strdup_printf("kdialog --title '%s'", title);`
- `result = system(command);`
- où `title` est le titre du message reçu...

Et *shell shock*

En 2014, plusieurs vulnérabilités de `bash`

- CVE-2014-6271 et ses petites sœurs

- injection *shell* dans le *parsing* des fonctions exportées
- vecteur d'attaque : une variable d'environnement
- faille dévastatrice
  - serveurs HTTP utilisant des scripts CGI
  - clients DHCP
  - serveurs SSH avec accès restreint

CVE-2014-1266 : goto fail Apple

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .
```

Correctif et enseignements tirés

Correctif

- retirer la ligne `goto fail` redondante

Et au-delà ?

- détecter le code mort
- tester que ce qui ne doit pas fonctionner échoue
- rejouer ces tests à chaque version pour éviter les régressions
- partager ces tests entre implémentations

L'autre `goto fail`

Un *bug* intéressant dans GnuTLS

- une fonction `verify` de vérification de certificat renvoyant
  - `-1` en cas d'erreur de *parsing*
  - `0` si le certificat est décodé, mais avec une signature invalide
  - `1` si le certificat est décodé, avec une signature valide

- que se passe-t-il lorsque l'on utilise la fonction de la manière suivante ?

```
if (verify (certificate)) {
    // The certificate is valid so
    // we continue the process
} else {
    error ("Invalid certificate");
}
```

Explications au tableau

- Rappels sur les projections mémoire d'un processus
- Description de la pile
- Un *Buffer Overflow* simple
- Exécution de code arbitraire dans la pile
- Contre-mesure : la non exécution de la pile
- Le ROP
- Contre-mesure : la randomisation
- Limites d'une randomisation partielle, d'une fuite d'information ou d'un `fork()` à répétition
- Contre-mesure : les canaris
- Perspectives : JOP, débordements dans le tas...

Le problème du *confused deputy* (1/2)

On considère un programme ayant les caractéristiques suivantes :

- il tourne avec les privilège de l'utilisateur `root`
- il réalise une action au profit d'un utilisateur non privilégié
- que peut-il arriver de fâcheux ?

Le problème du *confused deputy* (1/2)

On considère un programme ayant les caractéristiques suivantes :

- il tourne avec les privilège de l'utilisateur `root`
- il réalise une action au profit d'un utilisateur non privilégié
- que peut-il arriver de fâcheux ?

Exemple de tels programmes (*setuid* `root`) :

- `passwd`
- `sudo`
- `su`

Le problème du *confused deputy* (2/2)

Supposons maintenant qu'un tel programme

- prenne en argument un nom de fichier temporaire à créer
- crée un fichier dans `/tmp` ayant ce nom pour y écrire le contenu de son second argument

Le problème du *confused deputy* (2/2)

Supposons maintenant qu'un tel programme

- prenne en argument un nom de fichier temporaire à créer
- crée un fichier dans `/tmp` ayant ce nom pour y écrire le contenu de son second argument

Que se passe-t-il si on donne comme argument `../../../etc/passwd` ?

- une vulnérabilité de type *directory traversal*
- que l'on rencontre souvent dans le monde du Web

- ici, en plus, on peut créer une confusion et modifier un fichier existant au nom de `root`
- ce type de vulnérabilité est appelé *confused deputy*
- l'attaquant profite des privilèges d'une application pour lui faire réaliser une action autorisée, mais non légitime

Mécanismes de protections

- Vérification de l'absence du symbole / dans le fichier

Mécanismes de protections

- Vérification de l'absence du symbole / dans le fichier
  - comme souvent, ce mécanisme en liste noire n'est pas suffisant
  - l'attaquant peut jouer sur des liens symboliques dans `/tmp`

Mécanismes de protections

- Vérification de l'absence du symbole / dans le fichier
  - comme souvent, ce mécanisme en liste noire n'est pas suffisant
  - l'attaquant peut jouer sur des liens symboliques dans `/tmp`
- Il faut donc restreindre les possibilités du programme lorsqu'il ouvre le fichier

Mécanismes de protections

- Vérification de l'absence du symbole / dans le fichier
  - comme souvent, ce mécanisme en liste noire n'est pas suffisant
  - l'attaquant peut jouer sur des liens symboliques dans `/tmp`
- Il faut donc restreindre les possibilités du programme lorsqu'il ouvre le fichier
  - descente de privilège permanente (changement d'utilisateur)

Mécanismes de protections

- Vérification de l'absence du symbole / dans le fichier
  - comme souvent, ce mécanisme en liste noire n'est pas suffisant
  - l'attaquant peut jouer sur des liens symboliques dans `/tmp`
- Il faut donc restreindre les possibilités du programme lorsqu'il ouvre le fichier
  - descente de privilège permanente (changement d'utilisateur)
  - descente de privilège temporaire (changement d'utilisateur en conservant l'identité privilégiée en tant qu'identité sauvegardée)

Mécanismes de protections

- Vérification de l'absence du symbole / dans le fichier
  - comme souvent, ce mécanisme en liste noire n'est pas suffisant
  - l'attaquant peut jouer sur des liens symboliques dans `/tmp`
- Il faut donc restreindre les possibilités du programme lorsqu'il ouvre le fichier
  - descente de privilège permanente (changement d'utilisateur)
  - descente de privilège temporaire (changement d'utilisateur en conservant l'identité privilégiée en tant qu'identité sauvegardée)
  - restriction de la visibilité sur le système de fichiers (`chroot`) voire sur d'autres ressources (utilisation de *namespaces*)

Conclusion

Quelques éléments de conclusion

- développer de manière sécurisée est compliqué
- la complexité des algorithmes et du code rend les failles plus fréquentes
- les exemples présentés ici ne sont qu'une partie du panorama

Conclusion

Quelques éléments de conclusion

- développer de manière sécurisée est compliqué
- la complexité des algorithmes et du code rend les failles plus fréquentes
- les exemples présentés ici ne sont qu'une partie du panorama

Conseils pour le développeur

Conclusion

Quelques éléments de conclusion

- développer de manière sécurisée est compliqué
- la complexité des algorithmes et du code rend les failles plus fréquentes
- les exemples présentés ici ne sont qu'une partie du panorama

Conseils pour le développeur

- valider les entrées utilisateur

Conclusion

Quelques éléments de conclusion

- développer de manière sécurisée est compliqué
- la complexité des algorithmes et du code rend les failles plus fréquentes
- les exemples présentés ici ne sont qu'une partie du panorama

Conseils pour le développeur

- valider les entrées utilisateur
  - ne *jamais* supposer quelque chose sur une entrée sans le vérifier

Conclusion

Quelques éléments de conclusion

- développer de manière sécurisée est compliqué
- la complexité des algorithmes et du code rend les failles plus fréquentes
- les exemples présentés ici ne sont qu'une partie du panorama

Conseils pour le développeur

- valider les entrées utilisateur
  - ne *jamais* supposer quelque chose sur une entrée sans le vérifier
- tester que ce qui ne doit pas fonctionne échoue effectivement
  - certains *bugs* des piles TLS sont apparus dans des implémentations indépendantes
  - et maintenir des tests de non-régression

Conclusion (suite)

Pour aller plus loin (mais ça vient vite !)

- connaître le(s) langage(s) et les outils associés
  - *Mind your languages* (références sur le site du cours)
  - utiliser les outils correctement

Conclusion (suite)

Pour aller plus loin (mais ça vient vite !)

- connaître le(s) langage(s) et les outils associés
  - *Mind your languages* (références sur le site du cours)
  - utiliser les outils correctement
  - un projet C sans `-Wall -Wextra -Werror` dans son `Makefile` est voué à l'échec

Conclusion (suite)

Pour aller plus loin (mais ça vient vite !)

- connaître le(s) langage(s) et les outils associés
  - *Mind your languages* (références sur le site du cours)
  - utiliser les outils correctement
  - un projet C sans `-Wall -Wextra -Werror` dans son `Makefile` est voué à l'échec
- le développement est une étape du parcours, qui va de la compilation au déploiement...
- et qui requiert une évaluation

Conclusion (suite)

Pour aller plus loin (mais ça vient vite !)

- connaître le(s) langage(s) et les outils associés
  - *Mind your languages* (références sur le site du cours)
  - utiliser les outils correctement
  - un projet C sans `-Wall -Wextra -Werror` dans son `Makefile` est voué à l'échec
- le développement est une étape du parcours, qui va de la compilation au déploiement...
- et qui requiert une évaluation
- appliquer des mesures de défense en profondeur
  - réduction des privilèges
  - cloisonnement
  - mettre en place des mécanismes d'atténuation des effets
  - mettre en place des mécanismes de détection

Questions

?