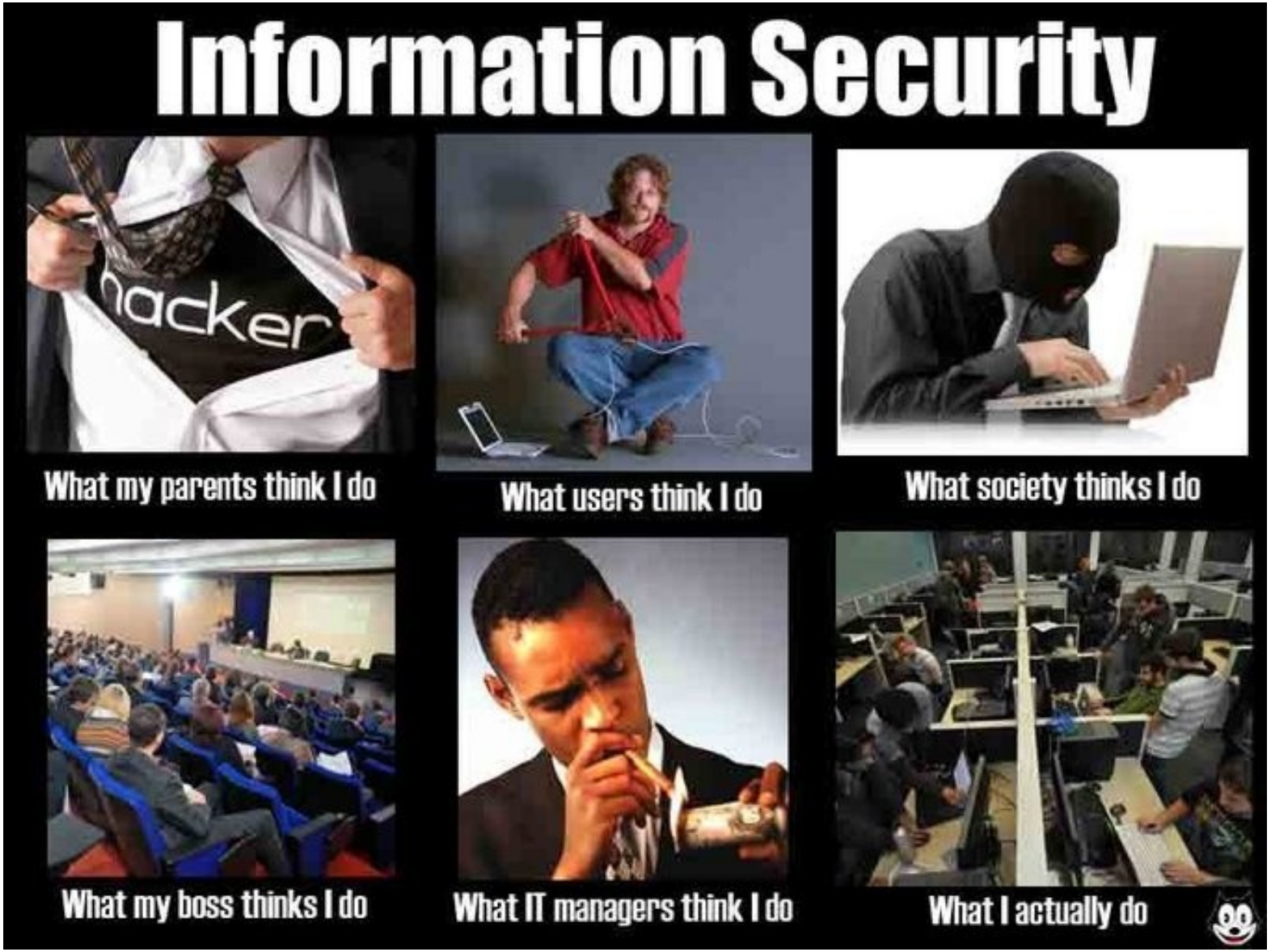
Sécurité applicative



First things first

Installez kali sur une virtualbox :

- https://dev.pierre-galvez.fr/installer-kali-sur-virtualbox/

Installez DVWA

- https://www.shellvoide.com/hacks/setup-hacking-environment-install-dvwa-application-on-linux/

Divers composants

Différents composants de la sécurité d'une application

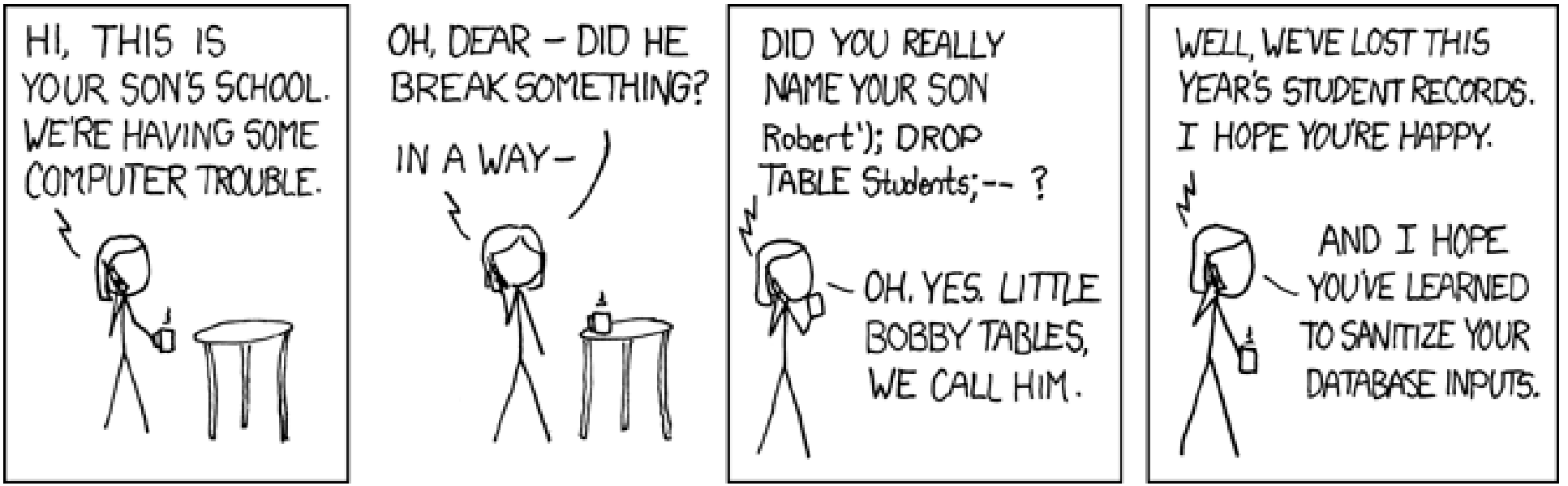
- l'application en tant que telle (code, faille, etc)
- Son environnement d'exécution (local, serveur)
- Sa communication avec de tierces entités (serveurs, dépôts de MAJ, API diverses)
- Etc

→ la première et plus importante tâche consiste à distinguer ces différents composants

Du côté du code de l’application en lui même

Les attaques par injection

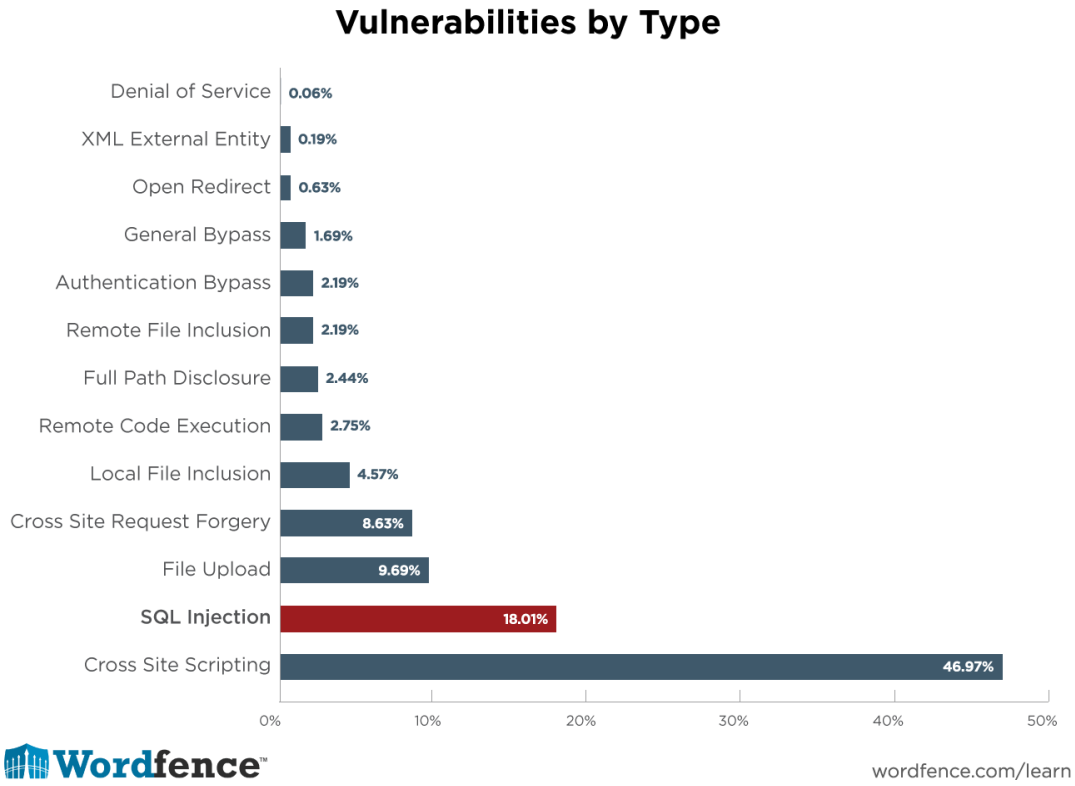
L’injection SQL : exploit of a Mum



L’injection SQL : comment ?

Principe de base

- Insérer une commande dans une requête SQL à destination du script PHP (ou autre) d’un site dynamique (i.e. doté d’un BDD)
- Nombreuses instance de ce schéma général
- Une des vulnérabilités les plus exploités




```
SELECT uid FROM Users WHERE name = 'Smith';-- ' AND password=
'4e383a1918b432a9bb7702f086c56596e';
```

L’injection SQL : principe de base

Injection de code dans une requête SQL par exploitation de failles syntaxiques

Finalités possibles

- Contourner un formulaire d’authentification
- Récupérer la BDD
- Voire la modifier

L’injection SQL : principe de base

Disjonction avec une tautologie

```
SELECT UserId, Name, Password FROM
Users WHERE UserId = 105 or 1=1;
```

```
SELECT * FROM Users WHERE Name ="" or
""="" AND Pass ="" or ""=""
```

L’injection SQL : principe de base

Exploitation des **batched statements**

SELECT * FROM Users; DROP TABLE Suppliers

txtUserId = getRequestString("UserId");

txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;

input dans le formulaire = 105; DROP TABLE Suppliers

A vous

Attaquez la BDD de DVWA avec SQLMAP

Les attaques cross script (XSS) : généralités

- Probablement le type d’attaque le plus courant depuis la banalisation du js dans les technos web
- Attaque par **injection de code**
- Cible : internaute
- Medium : site (html) + navigateur (js)

Le principe général :

- Introduire sur une page (ex. via un formulaire) du script qui sera interprété par le navigateur de la cible

// « marcher » est bon pour la santé et a 6 lettres (**confusion** des ordres)

Finalités poursuivies

- Vol de cookies
- Keylogging
- Phishing

NB : le **serveur** et **l'application** demeurent **indemnes**, l'attaque n'a d'existence que par la présence d'un **client/victime**

- Un certain **flou** sur la responsabilité

  → Un certain désintérêt des devs pour la protection contre ce type d'attaques (l'infrastructure dont ils ont la **responsabilité** ne craint pas grand-chose)

XSS plus en détail

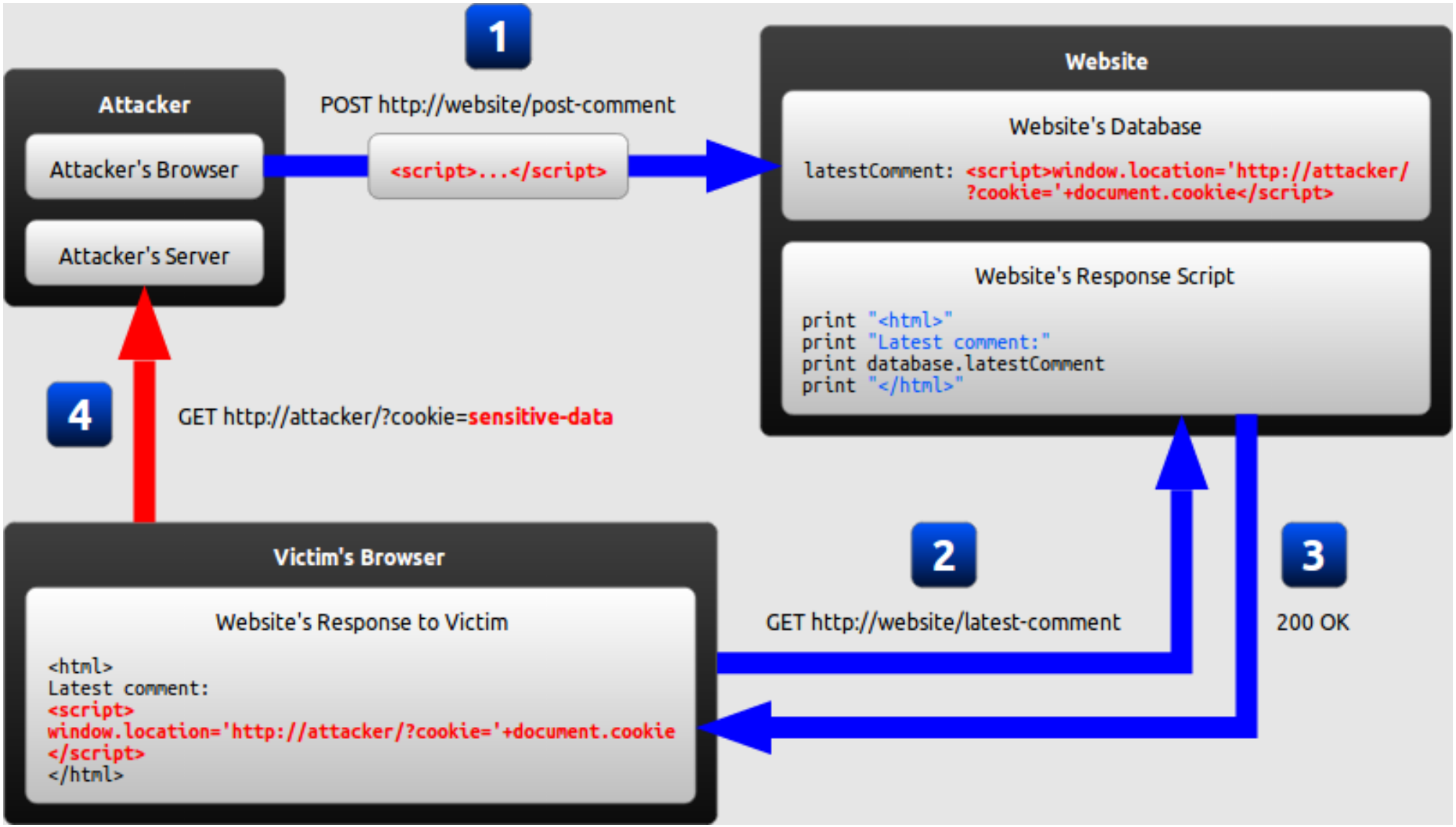
Composants

- **Site** – html (en dur ou généré par script (PHP, JS, Python, etc etc))
  - BDD du site
- **Victime** : internaute lambda avec son navigateur
- **Hacker**
  - serveur du hacker

3 variantes selon l'origine de la chaîne à l'origine du script

- Persistent XSS : BDD du site
- Reflected XSS : requête de la victime
- DOM-Based

Une persistent XSS : détail

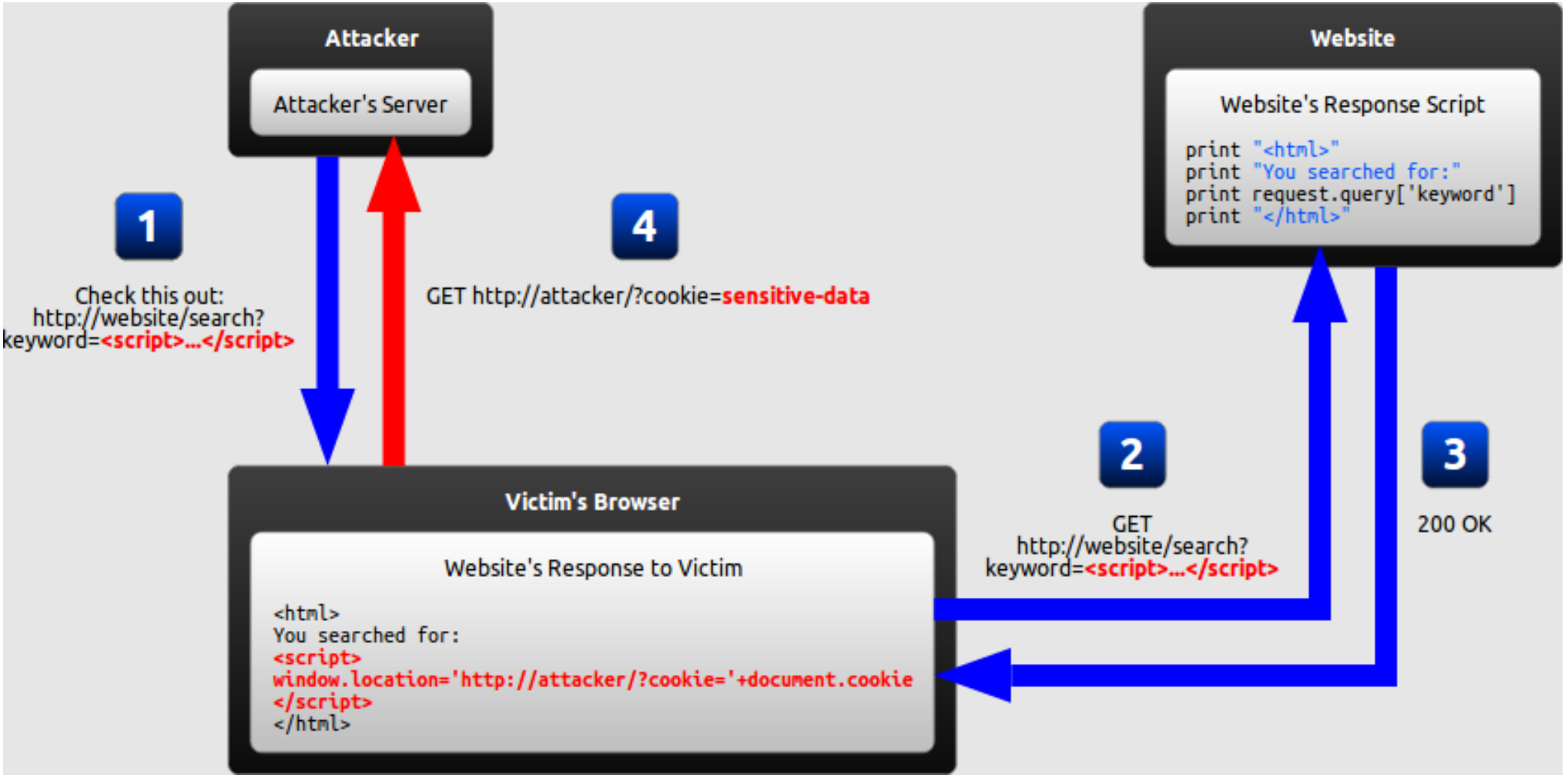


A vous

Réalisez une attaque de type Persistent XSS contre DVWA

Pour identifier les vulnérabilités explotables, servez vous d'OWASP (pré-installé sur Kali)

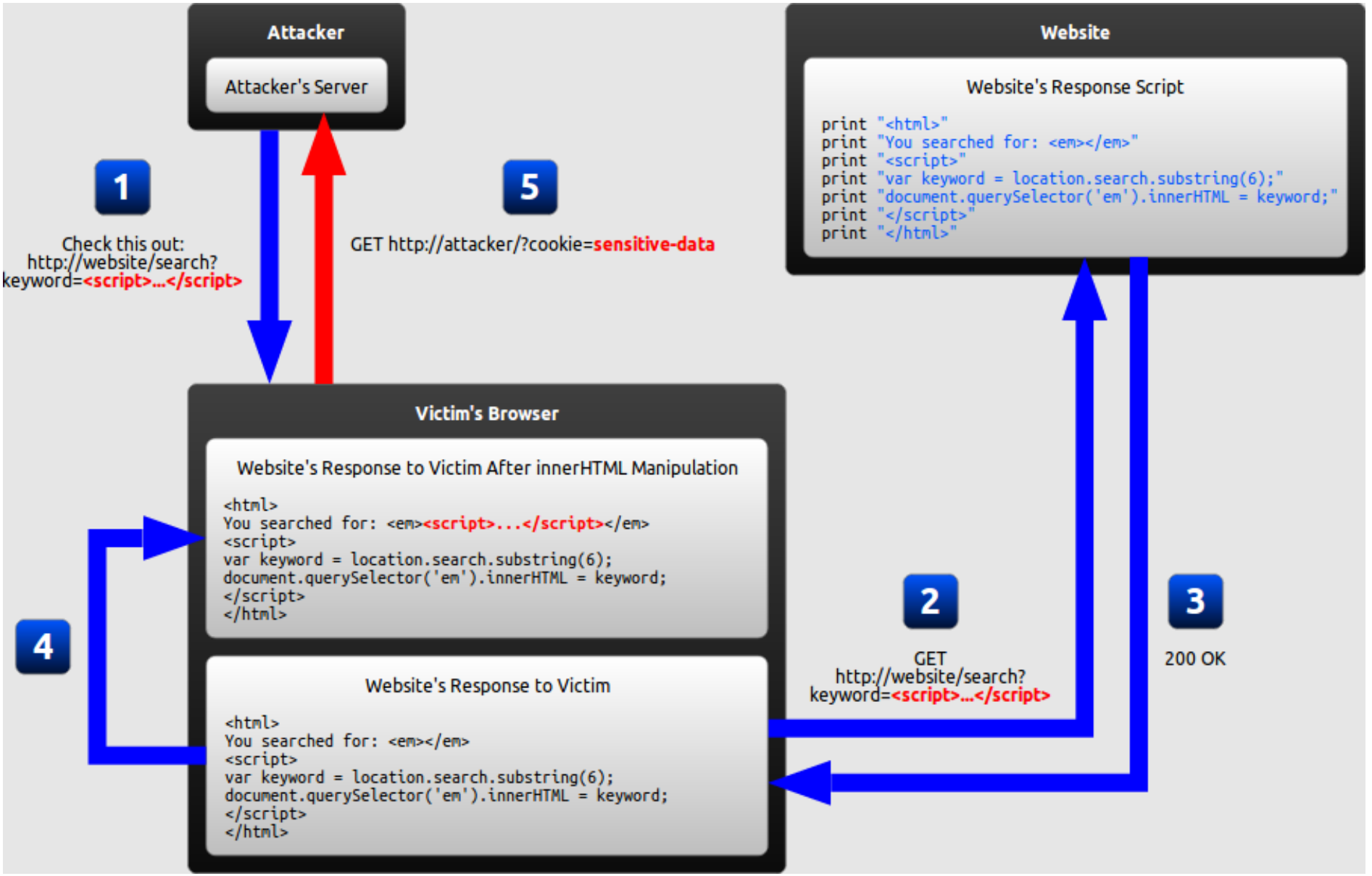
Reflected XSS



Reflected XSS : remarques

- Le client serait bien c** d’envoyer la requête responsable de son attaque
- Certes, mais il peut très fortement être induit à le faire (part de social engineering, emails, messageries instantanées, etc etc) → attaques ciblées

Plus subtile : DOM XSS



Remarques sur les DOM XSS

- La string n’est pas parsé par le navigateur de la victime avant que le code javacript du site soit exécuté
- c’est l’exécution d’un script légitime qui insère le script malicieux dans la page

Les spécificités des DOM

- Les vulnérabilités peuvent être présentes non seulement côté serveur mais aussi côté client
- Particulièrement pertinent dans un contexte d’app web générant l’html par le Javascipt sans rafraichissement de la page (react js et autres frameworks sexy du moment)

  → même un serveur sécurisé peut servir de vecteur à cette attaque

Prévention

Abandonner js……

…….. Je blague

Prévention : marcher VS « marcher »

Éviter que du code soit exécuté → contraindre le navigateur à le traiter comme du *plain text*

marcher est bon pour la santé et le mot dont nous nous servons pour le désigner (« marcher ») a 6 lettre

Prévention : plus techniquement

L’idée générale consiste à sécuriser l’input pour que le code ne soit pas exécuté. Pour cela, deux familles de mesures concrètes (complémentaires)

Encodage

neutraliser le code afin de le rendre inoffensif (e.g. < → &lt)

Validation

filtrer les user inputs afin que le navigateur ne les interprète pas comme du code (e.g. virer les <script>)

Encodage : dans les faits

« la base, quoi !»

userInput = <script>...</script>

code côté serveur

print "<html>"

print "Latest comment: "

print encodeHtml(userInput)

print "</html>"

html rendu

<html>

Latest comment:

<script>...</script>

</html>

Côté client (js)

utiliser les méthodes propres à chaque contexte

HTML (contenu)	node.textContent = userInput
HTML (attribut)	element.setAttribute(attribute, userInput)
Requête URL	window.encodeURIComponent(userInput)
Valeur CSS	element.style.property = userInput

Encodage : limitations

De nombreux contextes inadéquats :

- <input value=<%= HtmlEncode(somevar) %> id=textbox>
- a onclick=alert(document.cookie)
- <input value=a onclick=alert(document.cookie) id=textbox>

→ sensibilité au contexte

L’encodage ne se suffit pas à lui même

Validation : entre Charybde et Scylla

Filtrer les inputs utilisateurs (i.e. autoriser mais pas script).

Blacklist

Interdire certains parterns mais oblige à énumérer tous les patterns possibles

(e.g. javascript / javascript:)

→ problème de la **complexité**

Whitelist

N’autoriser que certains patterns mais limite les fonctionnalités du site

→ problème de la **simplicité**

→ risque **obsolescence** rapide du code

Content Security Policy (CSP)

Le vrai problème des XSS, c'est qu'une seule faille compromet l'ensemble de l'application / site

→ les CSP permettent de **mitiger ce risque**

https://developer.mozilla.org/fr/docs/HTTP/CSP

Les attaques par Social Engineering

Illustration par l’exemple : autopsie d’un coupable ; **I love You**

- Élu le virus le plus « virulent » de son époque par le Guinness World record
- Début des années 2000 (une certaine naïveté)

Mode d’action

- Entrée dans le système : pièce jointe d’un mail **LOVE-LETTER-FOR-YOU.TXT.vbs** NB : c’est un **script VB** mais le nom donne l’impression qu’il s’agit d’un TXT du fait que Windows, par défaut, n’affiche pas l’extension des fichiers
- Mode d’action : pensant ouvrir un txt, l’usagé exécute en réalité un script qui modifie des fichiers de divers types et s’envoie lui-même à tous les contacts de contenus sur **Microsoft Outlook**

Ici encore, l’analogie avec l’épidémiologie fonctionne assez bien (cycles transmission incubation transmission)

Illustration par l'exemple : autopsie d'un coupable ; **I love You**

Part humaine

- Les homo-sapiens sont curieux de nature et ouvrent leurs pièces jointes
- Ils communiquent entre-eux autant que faire se peut

  I love you, c'est le **Père Fouettard du Social Engineering** (comportements à risque)

  un exemple moderne

Part système

- Extensions cachées
- Les scripts VBS sont exécutables par défaut, par simple double clic
- Les contacts d'Outlook sont stockés en clair, accessibles à tout script

  part de responsabilité du **design d'un système** dans le succès d'un malware (incompétence du système immunitaire)

Du rôle de l’usager : l’expert cyber-sécu // psychologue // UX

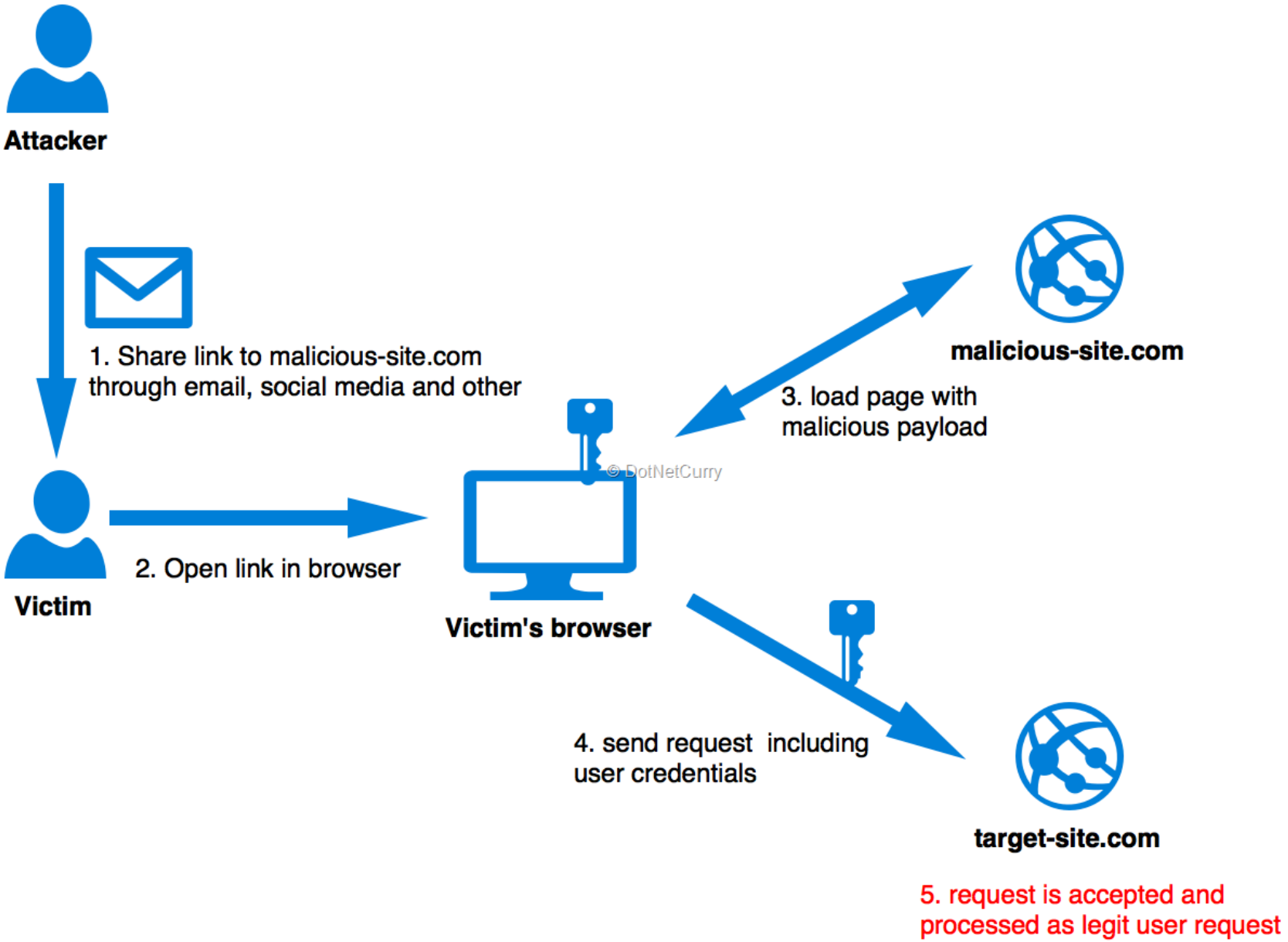
De nombreuses failles exploitent directement certaines tendances inhérentes à la cognition humaine (limites mnésiques, répugnance à la répétition, sensibilité aux propriétés de surface)

De nombreux exemples historiques

- Opérateurs enigma
- I love you

Une réponse adéquate à tout défi sécuritaire impose de prendre cette variable en compte

Cross-Site Request Forgery (CSRF)



CSRF : plus en détail

- Quelques synonymes : One-Click attack, Sea Surf, Hostile Linking
- Exploiter le fait que l'administrateur soit authentifié sur le serveur au moment de l'attaque
- Influer sur les paramètres d'un GET (ou d'un post, d'ailleurs) à l'aide d'un lien camouflé dans un mail, par exemple (ou autre contenu HTML)

CSRF : à vous

- DVWA Low : https://medium.com/@dannybeton/dvwa-csrf-tutorial-low-security-bc8474d2f49c
- DVWA Medium : https://medium.com/@dannybeton/dvwa-csrf-tutorial-medium-security-e14724ceee8a

→ analysez la différence de méthodes

De l’inefficience des mesures de prévention usuelles

- Cookie de session
- Multi-Step Transactions
- URL Rewriting
- HTTPS

Attaques par usurpation d’identité

Deux enjeux

S’assurer de l’identité de chacun des agents (serveurs, clients,)

S’assurer de l’inintelligibilité des éventuelles données capturées par un Tiers

L’installation de softwares : vecteur de contamination privilégié

Une instance d’attaque indirecte de type ***man in the middle***

le hacker se fait passer pour une source logiciel légitime (page de téléchargement, dépôt de logiciels) afin d’introduire une version modifiée du soft

Quelques exemples

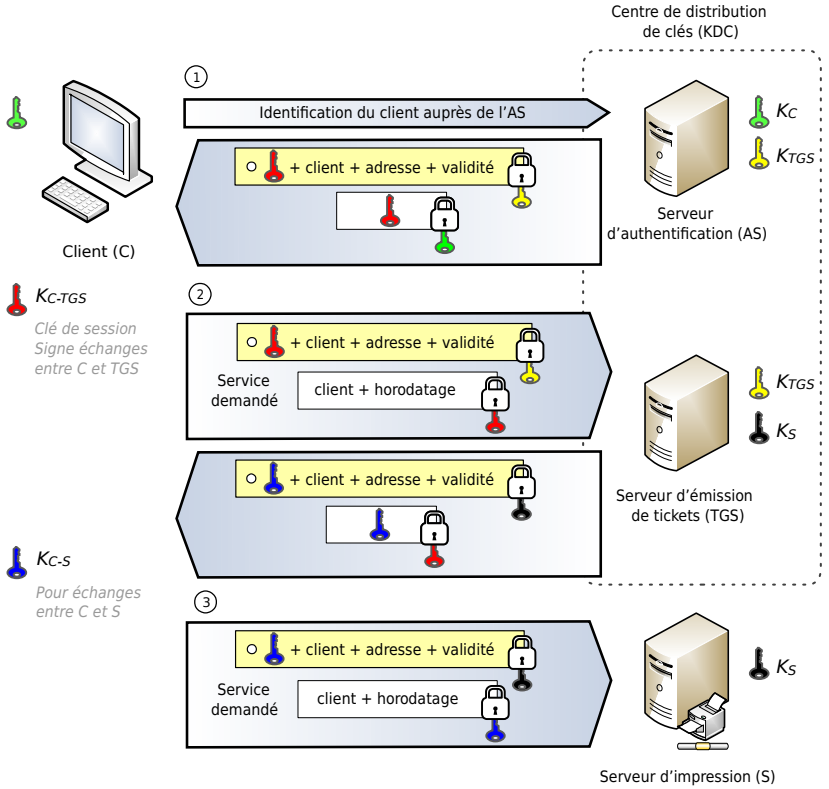
- fausses librairies
- Frameworks modifiés
- Diverses backdoors
- passerelle d'exécution
- Etc etc.

Exercice : analysez cette faille

Comment s'assurer de l'authenticité d'un soft

Sur les dépôts :

- Clef d'authentification des archives
- Ex : Kerberos



Pour les bin, packages et autres sources :

- Empreinte numérique par MD5, SHA256 ou signature PgP

- Kc : clé client
- Ks : clé serveur
- Kc-TGS : clé de session

Kerberos en détail dans un contexte d'authentification de sources software

Trois agents et leurs clés :

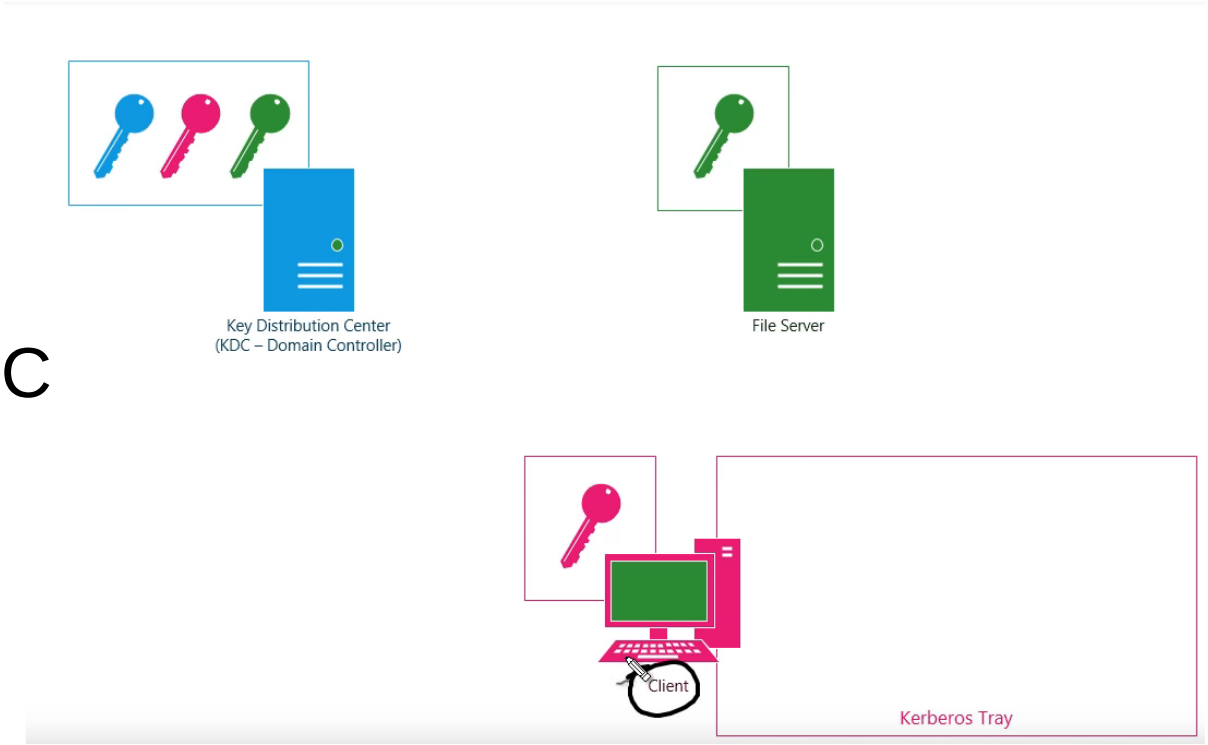
- Client
- Serveur de fichier
- Key Distribution Center

Points notables :

- Pas de communication entre le Serveur et le KDC
- Système à 3 clefs

Etape 1 :

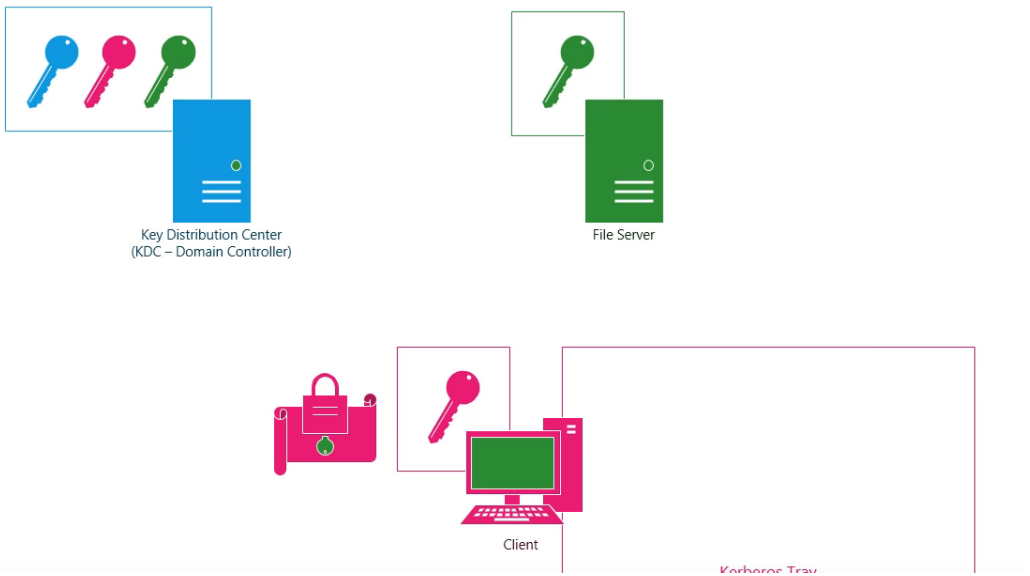
- Le client veut accéder aux ressources du serveur



Kerberos en détail dans un contexte d'authentification de sources software

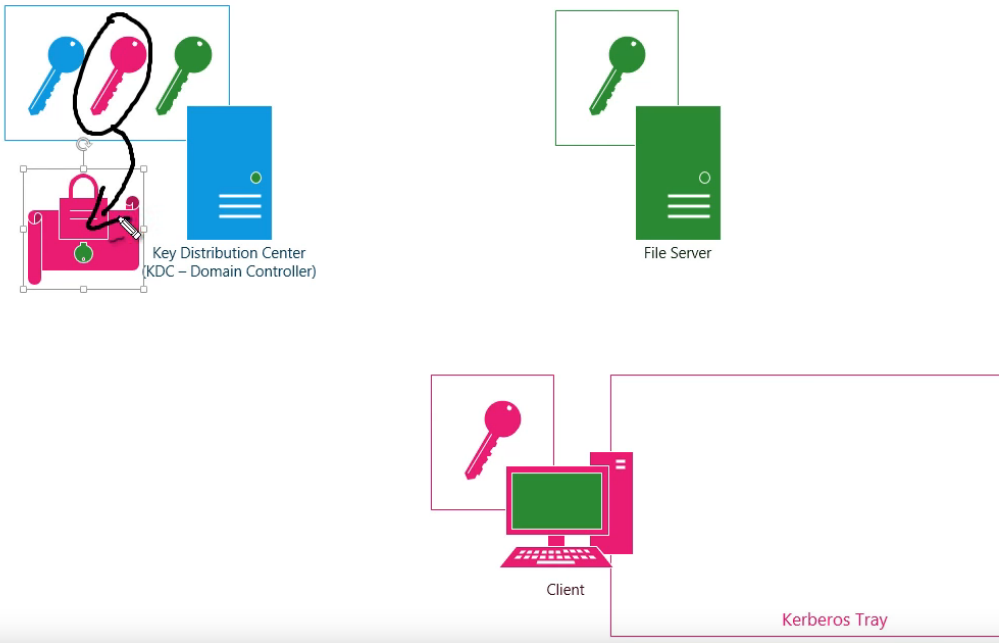
Etape 2

- Il encrypte la clé du serveur avec sa clé
- Envoi au KDC



Etape 3

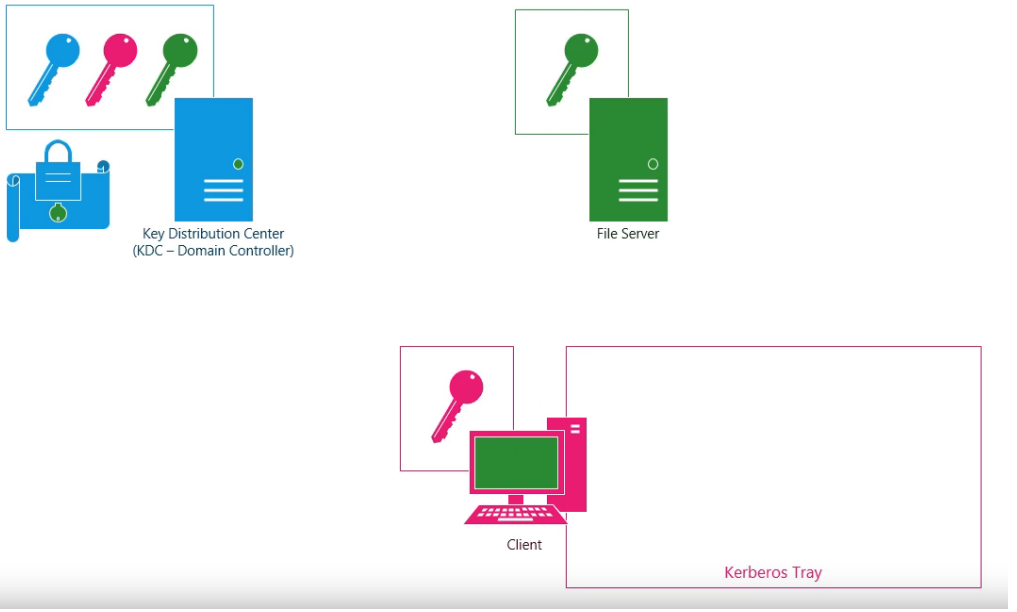
- KDC : « hey, mais je le connais ce p'tit gars »
- « en plus, je reconnais la clé serveur dont il me parle »



Kerberos en détail dans un contexte d'authentification de sources software

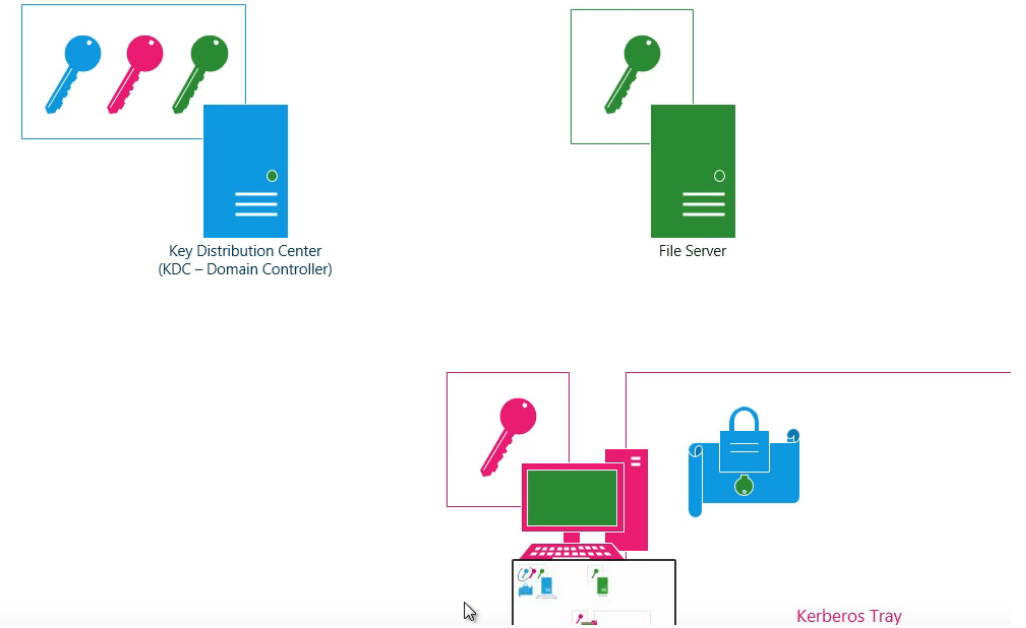
Etape 4

KDC renvoit la clé serveur encodée avec sa propre clé



Etape 5

- Le client la stocke dans son Kerberos Tray (RAM)
- Date de péremption, 8H



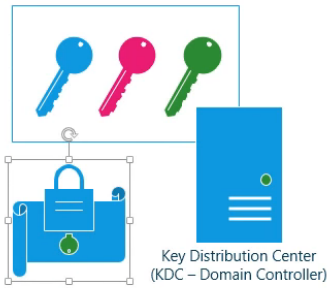
Kerberos en détail dans un contexte d'authentification de sources software

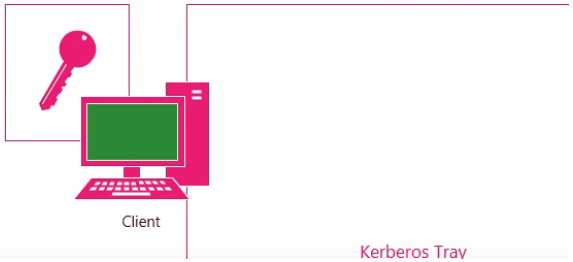
Etape 6

Client : « OK mais maintenant je veux une ressource du serveur »

Serveur : « tu n'as pas de ticket, demande au KDC »

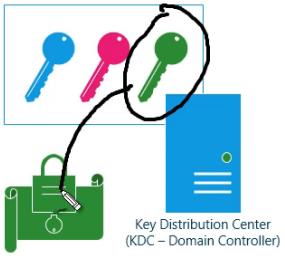
- Le client s'exécute et renvoie

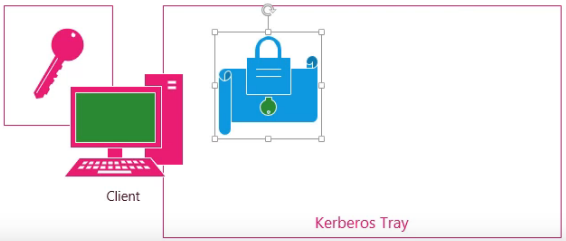




Etape 7

- Le KDC reconnais la clé qu'il avait envoyé au client
- Il lui renvoie ainsi et produit un ticket (clé serveur encodée avec la clé serveur)

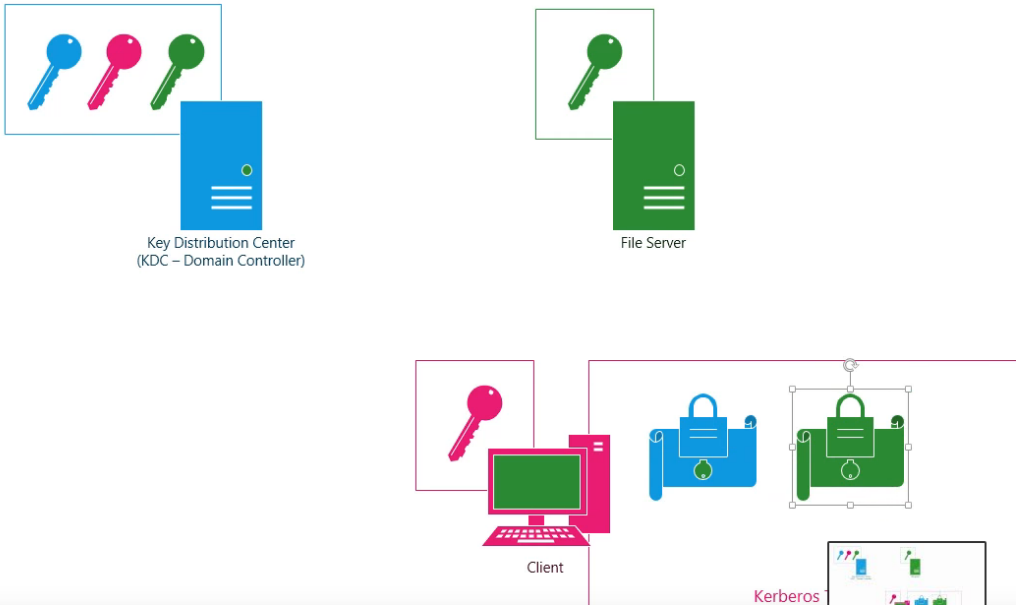




Kerberos en détail dans un contexte d'authentification de sources software

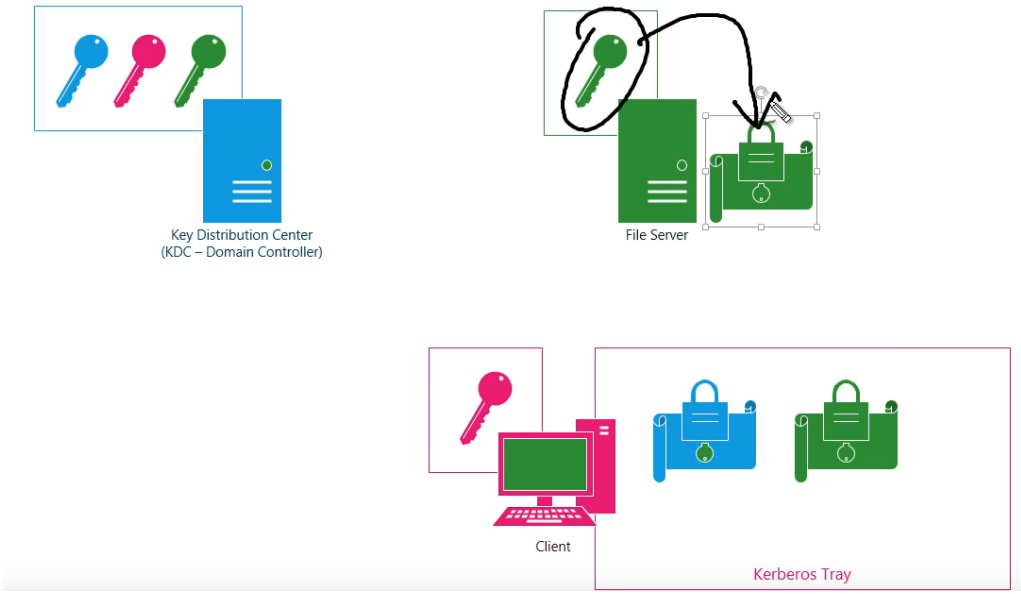
Etape 8

- Le client dispose maintenant du ticket



Etape 9

- Il l'envoie au serveur qui l'ouvre avec sa clé et reconnaît sa clé
- Il sait donc que le ticket a été produit par le KDC
- Il sait donc que le client est légitime et peut accéder aux ressources



Du côté, cette fois, du web applicatif

S’assurer de l’identité de chacun des agents (serveurs, clients,)

→ cookies de session

S’assurer de l’inintelligibilité des éventuelles données capturées par un Tiers

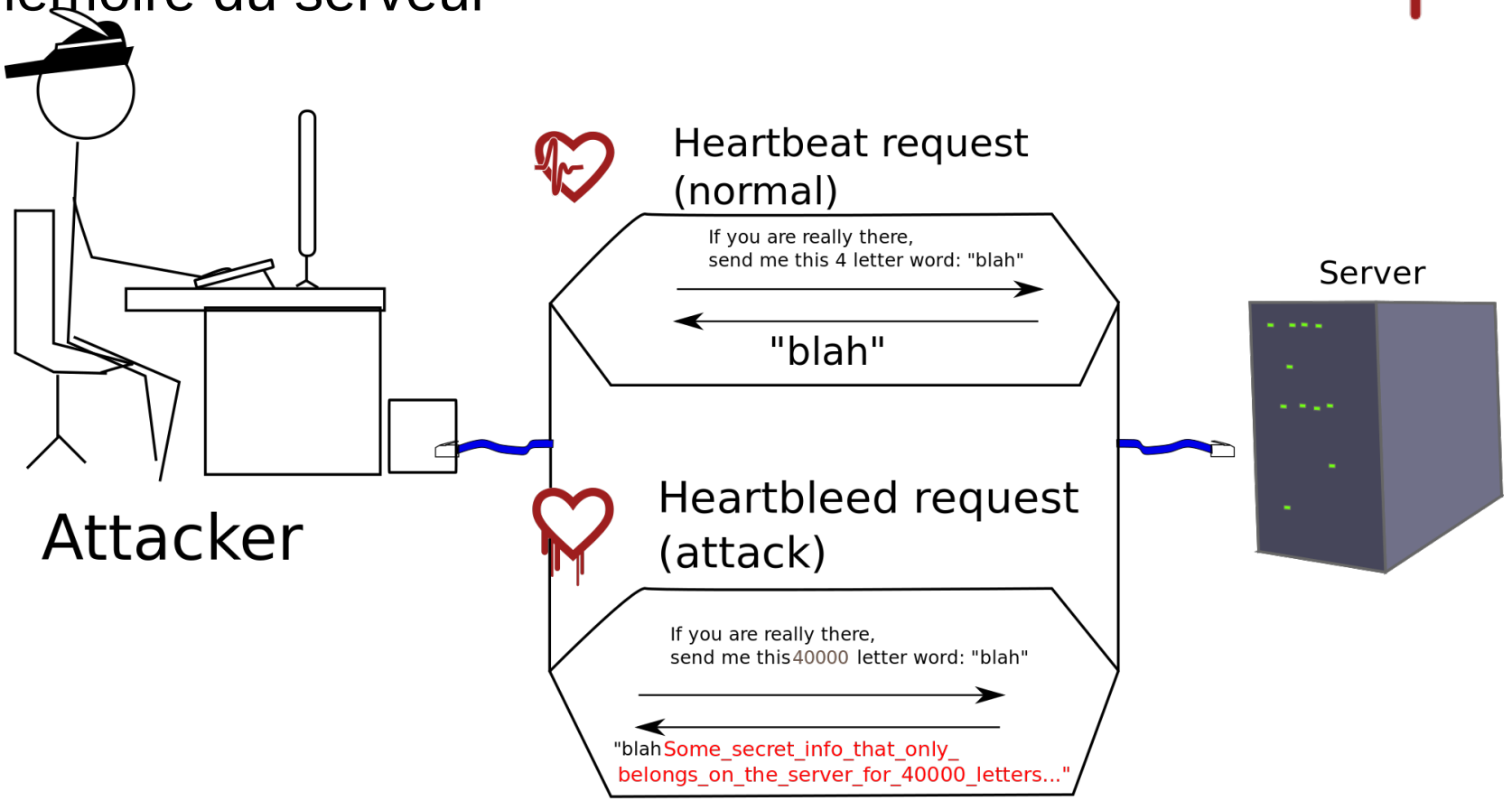
→ chiffrement SSL/TLS (HTTPS)

Parlons un peu d'Heartbleed

L'histoire en condensé

2014, vulnérabilité introduite par erreur dans le code d'**open SSL**

permet de récupérer le contenu en clair de la mémoire du serveur



Par delà Heartbleed, quelques failles SSL/TLS récentes

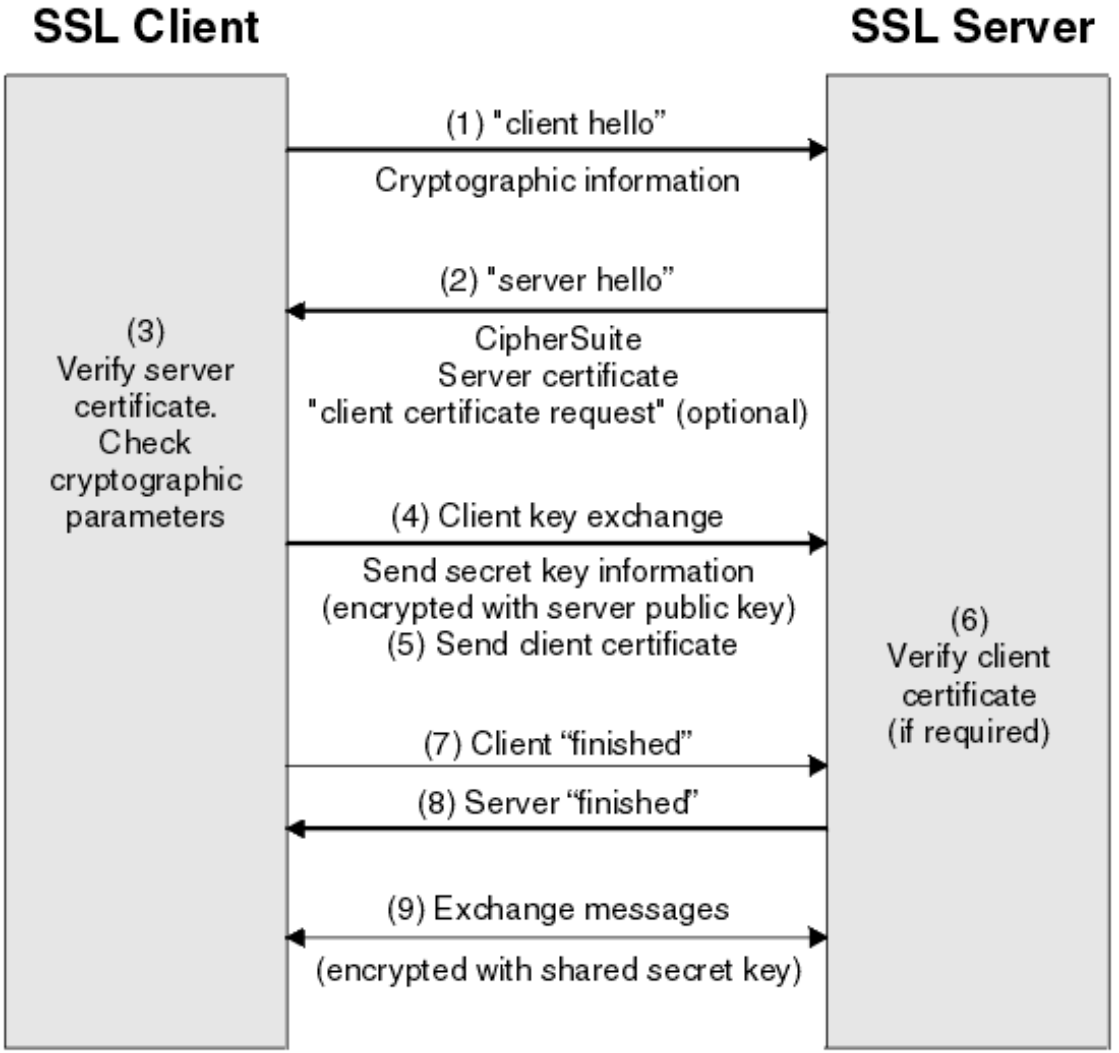
Les **downgrade attacks** en contexte « man-in-the-middle ». L'idée en gros :

- Les algorithmes de chiffrement des versions récentes du protocole TLS rendent la tâche difficile ?

  → forçons le serveur à communiquer selon une **version plus ancienne du protocole** (SSL v3.0, retour en 1996)

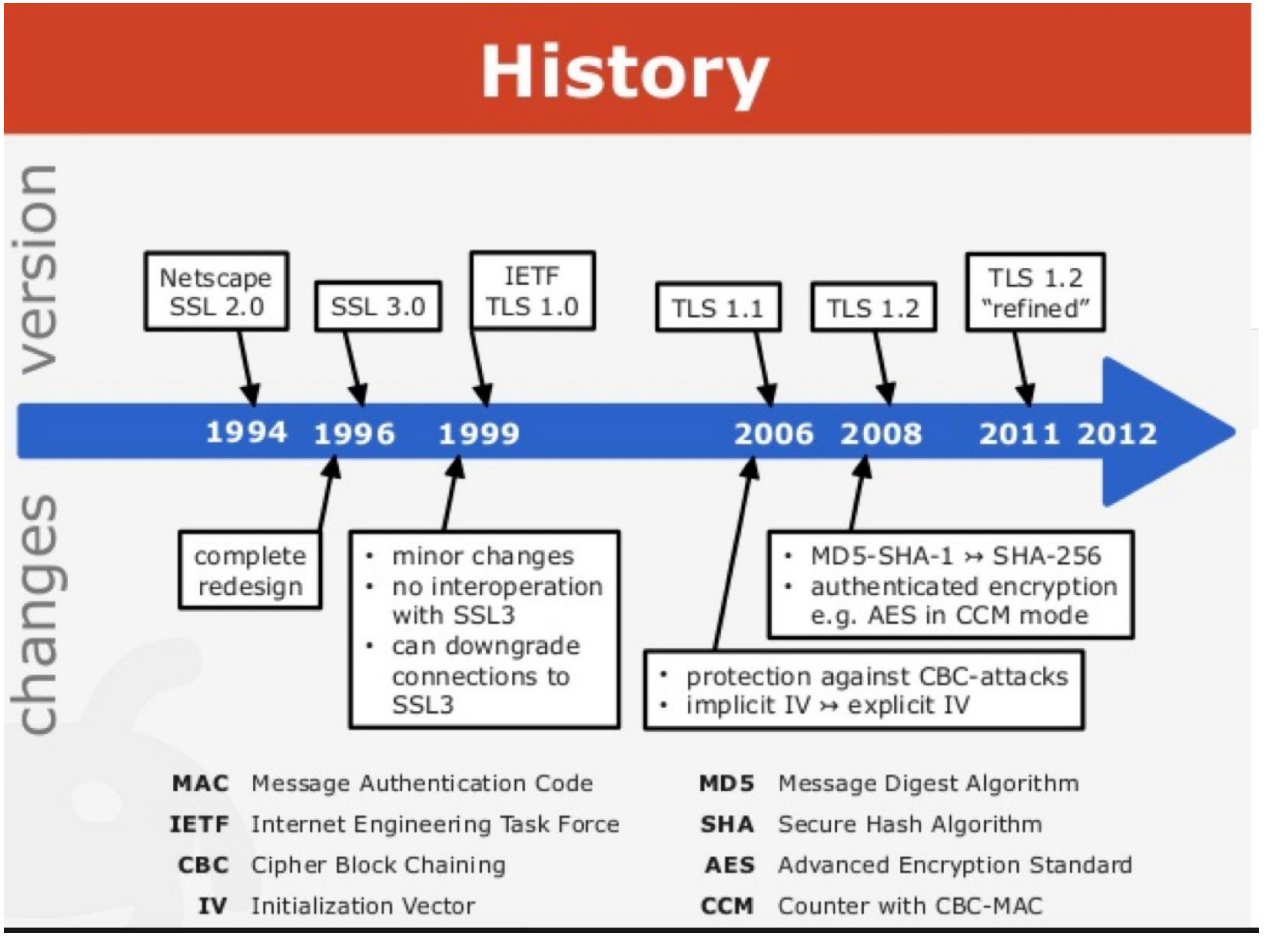
Par delà Heartbleed, quelques failles SSL/TLS récentes

Fonctionnement d'un handshake normal



Par delà Heartbleed, quelques failles SSL/TLS récentes

- Il est possible de forcer le serveur à utiliser un vieux protocole aux étapes 1, 2 ou 4
- Jouer sur ce qu'il perçoit de la compatibilité du client
- Vieux SSL → crypto plus fragile



Par delà Heartbleed, quelques failles SSL/TLS récentes

SSL 3.0

- RC4

  crypto par simple permutation et indexation

TLS 1.2

- SHA256
- AES

  Nous avons vu

TLS 1.3 (draft)

plus de support pour :

- MD5
- SHA-244
- RSA-Static
- Interdiction de SSL RC4
- Etc, etc

Par delà Heartbleed, quelques failles SSL/TLS « récentes »

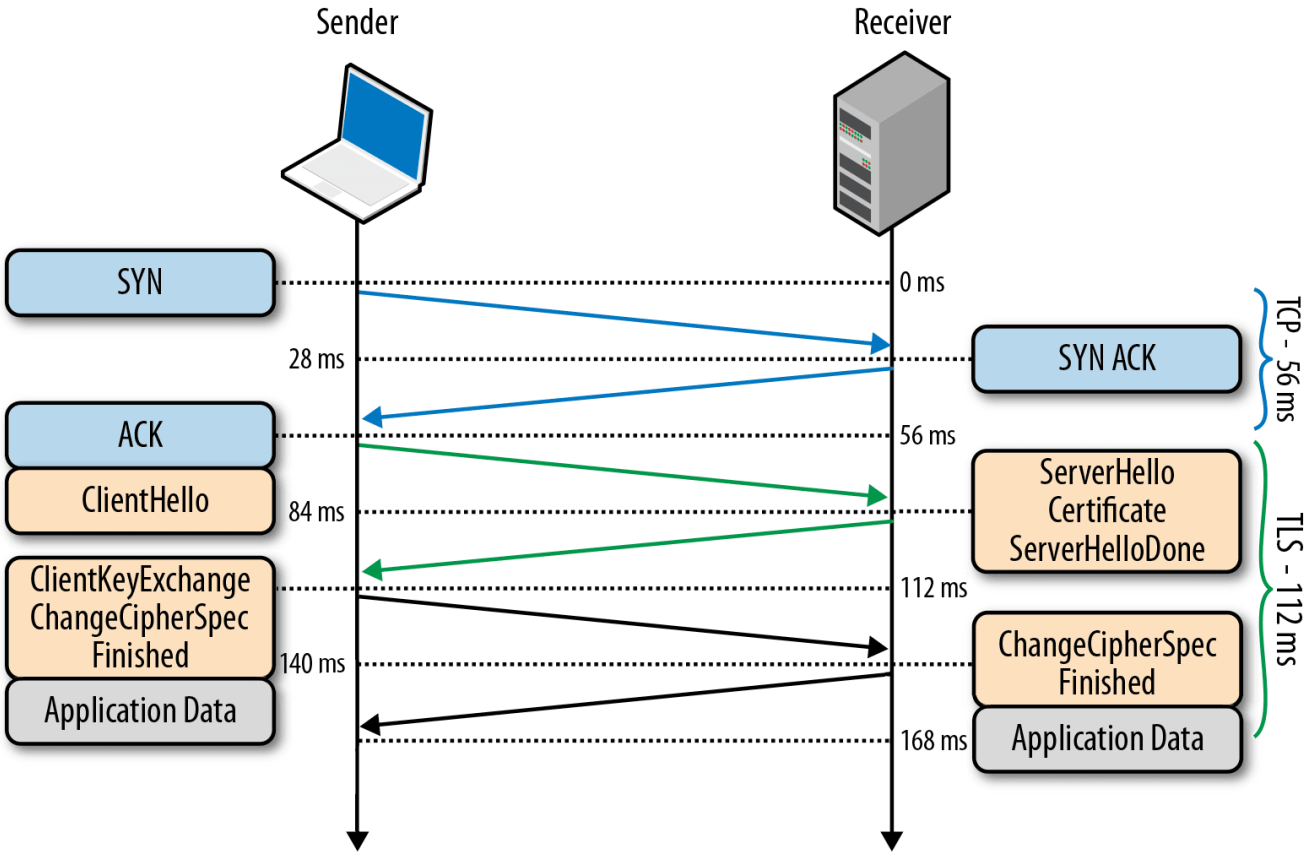
POODLE (bien connue et mitigée):

- TLS_FALLBACK_SCSV

Par delà Heartbleed, quelques failles SSL/TLS « récentes »

FREAK

- Exploite l'extension **FalseStart** (optimisation permettant au serveur d'envoyer des données avant la fin du handshake → un seul aller retour pour établir une nouvelle connexion TLS)



Par delà Heartbleed, quelques failles SSL/TLS « récentes »

Conclusions intermédiaires

- Responsabilité partagée
- Course aux armements
- Tradeoff vitesse / sécurité mal explicité (e.g. le cas de False Start)

Cross-protocol attacks

La même chose, mais en différent 2016: DROWN

Comment ? 2 possibilités

- I) Le serveur supporte le ssl 2.0 (17 % des serveurs

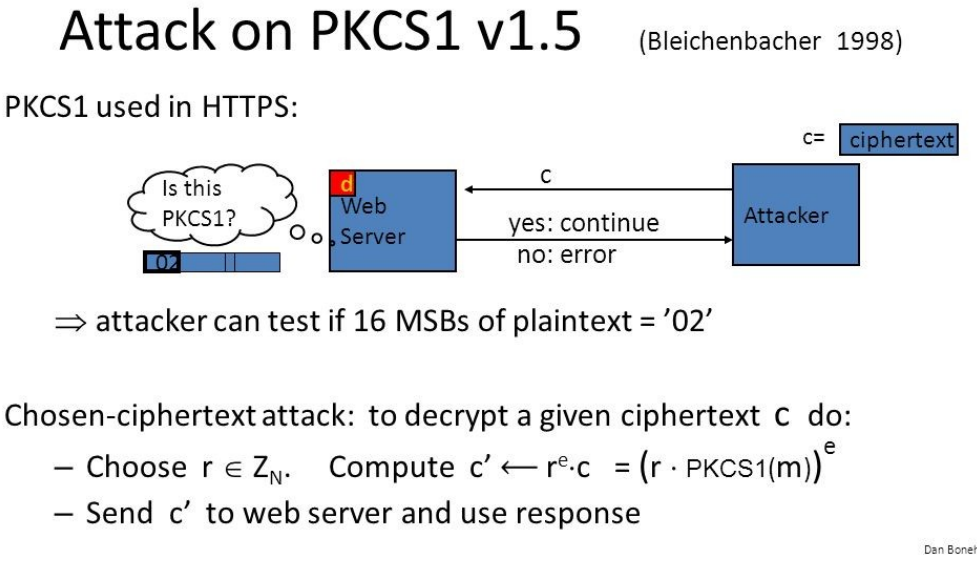
  OU/ET

- II) Une même clé privée est aussi utilisée sur un autre serveur autorisant SSL 2.0 (souvent le cas)

DROWN Plus en détail

I) une variante de Bleichenbacher attack

- Clé RSA PKCS#1 v1.5
- Repose sur un oracle : le serveur répond différemment si ce qu'il prend pour le **Client Key Exchange** est correctement padé (« padded »)



→ connaître la longueur de la clé client réduit considérablement l'espace des solutions

DROWN Plus en détail

In order to decrypt one TLS session, the attacker must passively capture about 1,000 TLS sessions using RSA key exchange, make 40,000 SSLv2 connections to the victim server, and perform 2 50 symmetric encryption operations. We successfully carried out this attack using an optimized GPU implementation and were able to decrypt a 2048-bit RSA ciphertext in less than 18 hours on a GPU cluster and less than 8 hours using Amazon EC2.

Côté serveurs, la place du sys-admin

Administration de serveurs

Ne relève pas de la sécurité applicative en temps que telle mais constitue une variable non négligeable

Interagit fortement avec la logique de fonctionnement de certains frameworks (e.g. temp files, droit de lecture/écriture)

Une illustration

Exploitez cette faille de sécurité

- https://www.root-me.org/en/Challenges/Web-Server/Backup-file

L’environnement réseau/OS de votre application est plus visible que vous le pensez

Pour vous en assurez, analysez cette ip i) avec **nmap** ii) avec **metasploit** scan

- 104.43.195.251

Que pouvez vous me dire de ce serveur ? Quel type d’attaque mettriez-vous en place ?

Administration de serveur et plateforme en ligne

Les conseils obvious

- Mettez à jour vos système, quel qu'ils soient
- Assurez-vous de la sécurité du réseau local
- l'accès physique aux serveurs
- Communication chiffrée avec le serveur

En cas d'hébergement tiers

- Privilégiez **SSH** et **SFTP** au Telnet et autres FTP

  → même si un c**** écoute le port 22, il va bien s'amuser avec les clés DSA

Pare-feu, kesako ?

Assure la sécurité d'un réseau par le contrôle des flux de données entrantes et sortantes

- Matériel ou logiciel
- Idée générale : filtrage
- Ex : empêche les Trojans d'envoyer des informations à l'extérieur
- Sépare le réseaux interne du réseau externe
- Éventuellement, sépare plusieurs zones au sein d'un même réseau local

Selon quelle logique de filtrage ?

- Ip
- Port (TCP, UDP)
- Entrant VS sortant
- Utilisateurs

- Réseau interne → zone de confiance
- Zone démilitarisée DMZ

Implémentation concrète d'une DMZ