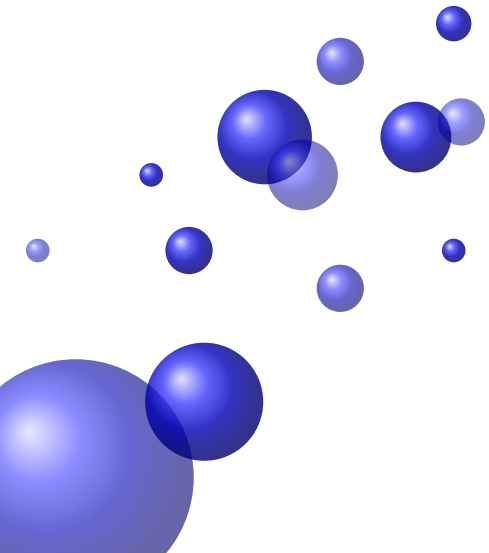
Distribution trouée Damn Vulnerable Web Application

Florian Barbarin - Abdelkader Beldjilali - Alexis Letombe



Le 13 avril 2017

Table des matières

Introduction

Dans un monde où les communications électroniques prennent une importance incommensurable, la sécurité des messages échangés est un enjeu devenu majeur pour l'ensemble des parties prenantes, qu'il s'agisse d'entreprises, d'institutions ou de simples citoyens.

Le secteur de l'aéronautique n'est pas exempt de mettre en œuvre une sécurisation des nombreux messages échangés afin de parer au mieux tout acte malveillant dont le but serait de modifier des messages ou d'intercepter des informations sensibles. On peut aujourd'hui penser aux drones qui échangent avec le sol à la fois des instructions essentielles au vol et des données issues de la mission réalisée (mesures, prises de vue, etc. . .).

La cryptographie donne les moyens à toutes les entités d'assurer la confidentialité, l'authenticité et l'intégrité des échanges. Un processus de chiffrement transforme le message *en clair* en message *chiffré*, incompréhensible, et le mécanisme de déchiffrement réalise l'opération inverse. L'idée sous-jacente de cette discipline est de faire en sorte qu'un message ayant subi un processus de chiffrement ne puisse être déchiffré qu'à l'aide d'un élément bien identifié par les parties prenantes, appelé *clé*. Un *cryptosystème* est défini par les mécanismes ou algorithmes de (dé)chiffrement, l'ensemble des textes en clair et textes chiffrés ainsi que les clés possibles.

1 Titre partie 1

Comme nous avons pu le voir en introduction de cette étude, le point de départ d'un système cryptographique est de trouver un problème dans la classe **NP** voire **NPC** avant de l'adapter aux exigences de chiffrement et déchiffrement des messages. Une fois ces étapes explicitées, nous verrons comment mettre en œuvre le chiffrement de Merkle-Hellman.

1.1 Choix d'un problème difficile

1.2 Adaptation du problème

1.2.1 Fonction à sens unique

1.2.2 Introduction d'une trappe

1.2.3 Perturbation

1.2.4 Application au chiffrement à clé publique

Dans le cadre d'un système de chiffrement à clé publique, si Alice souhaite envoyer un message à Bob, cette première chiffre son message avec la clé publique de Bob qui le déchiffrera alors avec sa clé privée. Il est donc nécessaire de définir les éléments constitutifs de la clé publique et de la clé privée de Bob.

Clé privée de Bob Comme nous l'avons esquissé à la section 5.2.3, Bob va tout d'abord générer une suite d'entiers super-croissante

- dans le « pire cas » et par le caractère super-croissant de la clé privée, le dernier élément s_n vaudra $D2^{n-1}$;
- on aura alors p de l'ordre de grandeur de $D2^n$ à $FD2^n$;
- les éléments de la clé publique T sont majorés par p par le modulo et un bloc chiffré y sera donc borné par $n \times FD2^n$.

Ce dernier bloc est chiffré comme les précédents. La figure 4 illustre le chiffrement d'un flux d'entrée avec une clé de taille 11. Cette méthode de bourrage est simple et ne crée pas d'ambiguïté puisque l'on sait dans les deux cas de figure que l'information utile se termine exactement au bit qui précède le premier bit non nul, en partant de la fin. Néanmoins, certaines méthodes de *padding* introduisent des vulnérabilités dans le code que la cryptanalyse pourra exploiter pour obtenir de l'information sur le message en clair. Il existe ainsi des méthodes standardisées de *padding* aléatoire que nous n'aborderons pas dans notre étude [1].

FIGURE 1 – Chiffrement d'un flux avec une clé de taille $n = 11$

1. Utiliser du *padding* aléatoire peut avoir l'avantage, y compris lorsque le message a déjà une taille multiple de celle d'un bloc, de produire des messages chiffrés différents pour un même message clair.

2 File inclusion

De nombreux langages de programmation permettent d'inclure des portions de code contenues dans d'autres fichiers que celui en cours d'exécution. Le mécanisme mis à disposition permet de recopier dans le script principal le code contenu dans un autre fichier. Cette procédure est transparente à l'œil de l'utilisateur et peut-être très avantageuse pour le développeur d'un site internet.

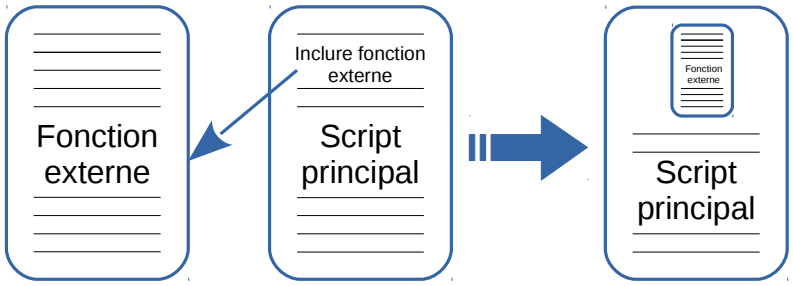


FIGURE 2 – Mécanisme d'inclusion d'un fichier

En effet, inclure du code contenu dans un autre fichier permet, entre autre, les deux utilisations suivantes :

— inclure des portions de code différentes en fonction de choix de l'utilisateur ou de l'environnement de ce dernier ;

— inclure des portions de code utilisées dans plusieurs scripts (par exemple une fonction de connexion à une base de données) afin de ne pas avoir à recopier les mêmes lignes à différents endroits et de ne modifier qu'un seul fichier en cas de modification de la fonction.

Nous voyons donc que le premier point ci-dessus permet d'obtenir une réelle adaptabilité du code alors que le second point donne la possibilité au développeur d'écrire du code concis et factorisé. Nous allons cependant voir que ce mécanisme n'est pas dépourvu de vulnérabilités.

2.1 Description de la vulnérabilité

La principale vulnérabilité connue dans le mécanisme que nous venons d'expliciter intervient lorsque l'inclusion d'un script est gérée par une variable pouvant être contrôlée par un attaquant. On se retrouve alors plutôt dans le premier cas d'utilisation indiqué, c'est à dire inclure des portions de code différentes en fonction de choix de l'utilisateur ou de l'environnement de ce dernier. En effet, dans le second cas d'utilisation, l'inclusion du fichier est généralement écrite "en dur" dans le script principal et ne peut donc pas être facilement modifié par un attaquant.

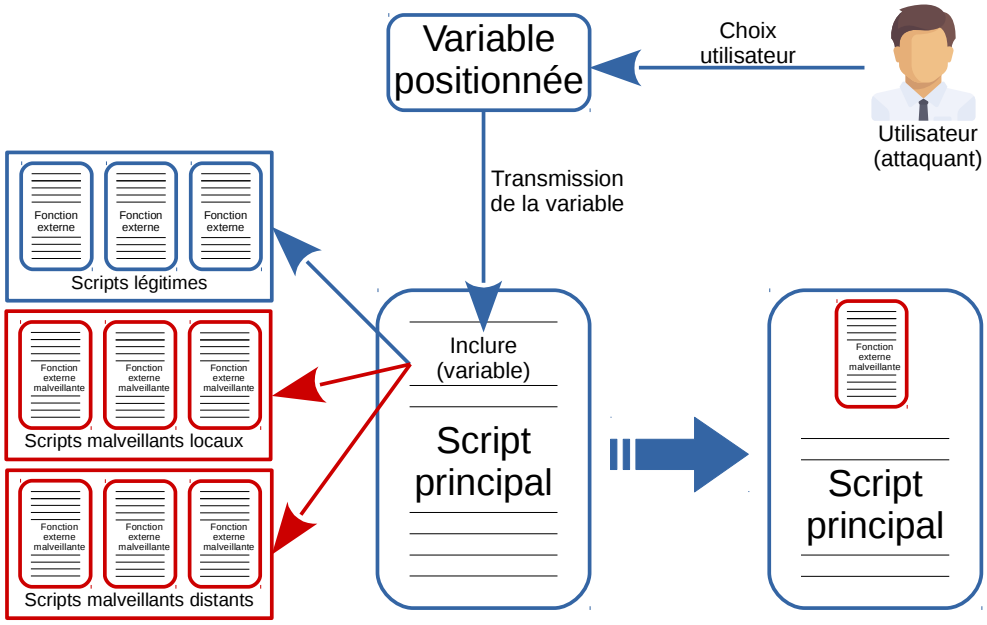


FIGURE 3 – Vulnérabilité d'inclusion d'un fichier

On remarque dans le schéma 2.1 que dans le cas où un attaquant peut avoir accès à la variable permettant de sélectionner le script légitime, celui-ci peut en modifier le contenu de deux façons. On parle alors de Local File Inclusion (LFI) et de Remote File Inclusion (RFI).

2.1.1 Local File Inclusion

Une fois que l'attaquant est en capacité de modifier le contenu de la variable indiquant le nom du script à inclure, celui-ci peut y indiquer un chemin local (i.e. directement sur le serveur) vers un script contenant du code malveillant. Il peut s'agit d'un script que l'attaquant a au préalable placé sur le serveur ou d'un script déjà présent qui effectue des opérations pouvant porté atteinte à la disponibilité de la machine voire à l'intégrité ou la confidentialité des données.

2.1.2 Remote File Inclusion

L'attaquant peut également indiquer dans la variable un chemin distant (i.e. vers un autre serveur) pointant vers un script contenant du code malveillant. Cette technique a pour avantage de faciliter la gestion du contenu du script malveillant par l'attaquant qui peut y inclure toutes les fonctionnalités qu'il souhaite voire le faire évoluer en fonction de la réponse de la machine attaquée.

Dans les deux cas, les scripts malveillants sont recopiés au sein du code du script principal qui sera au final exécuté par le serveur. Cette vulnérabilité offre donc de vastes possibilités à un attaquant qui peut alors faire exécuter par un serveur n'importe quelles fonctionnalités qu'il souhaite.

2.2 Exploitation de la vulnérabilité

2.3 Contre-mesure

3 File upload

3.1 Description de la vulnérabilité

3.2 Exploitation de la vulnérabilité

3.3 Contre-mesure

4 Insecure CAPTCHA

4.1 Description de la vulnérabilité

4.2 Exploitation de la vulnérabilité

4.3 Contre-mesure

5 Titre partie 3

Comme nous avons pu le voir en introduction de cette étude, le point de départ d'un système cryptographique est de trouver un problème dans la classe **NP** voire **NPC** avant de l'adapter aux exigences de chiffrement et déchiffrement des messages. Une fois ces étapes explicitées, nous verrons comment mettre en œuvre le chiffrement de Merkle-Hellman.

5.1 Choix d'un problème difficile

5.2 Adaptation du problème

5.2.1 Fonction à sens unique

5.2.2 Introduction d'une trappe

5.2.3 Perturbation

5.2.4 Application au chiffrement à clé publique

Dans le cadre d'un système de chiffrement à clé publique, si Alice souhaite envoyer un message à Bob, cette première chiffre son message avec la clé publique de Bob qui le déchiffrera alors avec sa clé privée. Il est donc nécessaire de définir les éléments constitutifs de la clé publique et de la clé privée de Bob.

Clé privée de Bob Comme nous l'avons esquissé à la section 5.2.3, Bob va tout d'abord générer une suite d'entiers super-croissante

— dans le « pire cas » et par le caractère super-croissant de la clé privée, le dernier élément s_n vaudra $D2^{n-1}$;

— on aura alors p de l'ordre de grandeur de $D2^n$ à $FD2^n$;

— les éléments de la clé publique T sont majorés par p par le modulo et un bloc chiffré y sera donc borné par $n \times FD2^n$.

Ce dernier bloc est chiffré comme les précédents. La figure 4 illustre le chiffrement d'un flux d'entrée avec une clé de taille 11. Cette méthode de bourrage est simple et ne crée pas d'ambiguïté puisque l'on sait dans les deux cas de figure que l'information utile se termine exactement au bit qui précède le premier bit non nul, en partant de la fin. Néanmoins, certaines méthodes de *padding* introduisent des vulnérabilités dans le code que la cryptanalyse pourra exploiter pour obtenir de l'information sur le message en clair. Il existe ainsi des méthodes standardisées de *padding* aléatoire que nous n'aborderons pas dans notre étude [1].

FIGURE 4 – Chiffrement d'un flux avec une clé de taille $n = 11$

1. Utiliser du *padding* aléatoire peut avoir l'avantage, y compris lorsque le message a déjà une taille multiple de celle d'un bloc, de produire des messages chiffrés différents pour un même message clair.

6 Conclusion et perspectives

Bien que le problème de somme de sous-ensembles (SSP) soit NP-complet, des algorithmes peuvent être mis en place pour tenter de casser le cryptosystème de Merkle-Hellman dans sa version basique. Si les algorithmes de force brute montrent très vite leurs limites, l'apport de théories plus ou moins récentes telles que la géométrie des nombres et la notion de réseau permet d'envisager des algorithmes nouveaux et élégants, bien que leur succès soit loin d'être garanti. Enfin, tous les sacs à dos ne se valent pas. Comme nous avons eu l'occasion de le voir, certaines méthodes comme la programmation dynamique ont horreur des sacs à dos dilatés alors que la densité est l'ennemie de l'algorithme LLL. La diversité des méthodes de cryptanalyse et la difficulté de se protéger contre l'une sans s'exposer à une autre justifie pleinement l'abandon de ce cryptosystème au profit de systèmes réputés plus sûrs, du moins à l'heure actuelle, comme RSA.

Il n'en reste pas moins que le SSP reste un problème intéressant à étudier sur le plan théorique et pratique. L'application de méthode hybrides associant les techniques de programmation dynamique et arborescentes avec heuristiques ont fait l'objet de nombreuses publications. Enfin, appliquer des algorithmes endémiques au domaine de l'intelligence et de l'apprentissage artificielle, tels les « colonies de fourmis », les algorithmes génétiques ou les réseaux de neurones pourraient fournir des pistes plus innovantes.

A Annexe

— Pour écrire simplement les clés sur le disque, nous utilisons Marshal, qui garantit la compatibilité entre toutes les plateformes pour une même version de OCaml.

— Bien que BatIO propose une API pour manipuler les canaux au niveau du bit, nous avons préféré rester au niveau de l'octet car il s'agit d'une solution plus évolutive – rares sont les bibliothèques proposant ce genre de fonctions. D'ailleurs, son fonctionnement est identique à ce que nous implémentons, reposant sur une lecture octet par octet.

— Dans la version actuelle du code, les canaux d'entrées-sorties ne sont pas toujours fermés proprement lorsqu'une exception « fatale » est rencontrée...

— Les blocs chiffrés sont écrits sur le canal de sortie sous forme de chaîne de caractères. Ainsi, le message chiffré constitué de deux blocs « 1234 5678 » est écrit, sous forme hexadécimale, « 31 32 33 34 00 35 36 37 38 00 ». Pour déchiffrer, on lit donc le canal d'entrée « chaîne par chaîne ». Cela a l'avantage de produire une sortie lisible mais présente l'inconvénient de consommer bien plus d'espace qu'une représentation binaire qui serait spécialement conçue pour le problème.

B Annexe

— Pour écrire simplement les clés sur le disque, nous utilisons Marshal, qui garantit la compatibilité entre toutes les plateformes pour une même version de OCaml.

— Bien que BatIO propose une API pour manipuler les canaux au niveau du bit, nous avons préféré rester au niveau de l'octet car il s'agit d'une solution plus évolutive – rares sont les bibliothèques proposant ce genre de fonctions. D'ailleurs, son fonctionnement est identique à ce que nous implémentons, reposant sur une lecture octet par octet.

— Dans la version actuelle du code, les canaux d'entrées-sorties ne sont pas toujours fermés proprement lorsqu'une exception « fatale » est rencontrée...

— Les blocs chiffrés sont écrits sur le canal de sortie sous forme de chaîne de caractères. Ainsi, le message chiffré constitué de deux blocs « 1234 5678 » est écrit, sous forme hexadécimale, « 31 32 33 34 00 35 36 37 38 00 ». Pour déchiffrer, on lit donc le canal d'entrée « chaîne par chaîne ». Cela a l'avantage de produire une sortie lisible mais présente l'inconvénient de consommer bien plus d'espace qu'une représentation binaire qui serait spécialement conçue pour le problème.

Damn Vulnerable Web App (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goals are to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and aid teachers/students to teach/learn web application security in a class room environment.