ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « INFORMATIQUE et RESEAUX »

présentée et soutenue publiquement par

Marco BALDUZZI

le 15/12/2011

Mesures automatisées de nouvelles menaces sur Internet

Directeur de thèse : **Prof. Engin KIRDA**

Jury

Refik Molva, Professeur, Institut EURECOM, Sophia Antipolis	Examinateur et Président
Christopher Kruegel, Professeur, University of California, USA	Rapporteur
Evangelos Markatos, Professeur, FORTH-ICS, Grèce	Rapporteur
Herbert Bos, Professeur, Vrijie Universiteit, Pays-Bas	Examinateur
Marc Dacier, Senior Director, Symantec Research Labs, USA	Examinateur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

THÈSE

Résumé

Pendant les vingt dernières années, Internet s'est transformé d'un réseau simple de taille limitée à un système complexe de grandes dimensions. Alors que Internet était initialement utilisé pour offrir du contenu statique, organisé autour de simples sites web, aujourd'hui il fournit en même temps contenu et services complexes (comme chat, e-mail ou le web) ainsi que l'externalisation de calculs et d'applications (cloud computing). En 2011, le nombre d'utilisateurs d'Internet a dépassé deux milliards (un tiers de la population mondiale) et le nombre de domaines enregistrés a atteint 800 millions. Les sites web sont accessibles à partir d'ordinateurs personnels, de tablettes, smartphones en permanence et depuis n'importe où.

Les attaquants ne sont pas indifférents à l'évolution d'Internet. Souvent motivées par un florissant marché noir, les attaquants sont constamment à la recherche de vulnérabilités, d'erreurs de configuration et de nouvelles techniques. Ceci afin d'accéder à des systèmes protégées, de voler des informations privées, ou propager du contenu malveillant.

Les vulnérabilités traditionnelles comme les buffer overflows ou les injections SQL sont encore exploitées. Toutefois, de nouveaux vecteurs d'attaque qui exploitent des canaux non conventionnels à grande échelle (dans le cloud computing, par exemple) sont également et régulièrement découverts. À ce jour, peu de recherches ont été réalisées pour mesurer la prévalence et l'importance de ces menaces émergentes sur Internet. Or, les techniques traditionnelles de détection ne peuvent pas être facilement adaptés aux installations de grandes dimensions, et des nouvelles méthodologies sont nécessaires pour analyser et découvrir failles et vulnérabilités dans ces systèmes complexes.

Cette thèse avance l'état de l'art dans le test et la mesure à grande échelle des menaces sur Internet. Nous analysons trois nouvelles classes de problèmes de sécurité sur ces infrastructures qui ont connu une rapide augmentation de popularité: le clickjacking, la pollution de paramètres HTTP et les risque liés au cloud computing. De plus, nous introduisons la première tentative d'estimer à grande échelle la prévalence et la pertinence de ces problèmes sur Internet.

Abstract

In the last twenty years, the Internet has grown from a simple, small network to a complex, large-scale system. While the Internet was originally used to offer static content that was organized around simple websites, today, it provides both content and services (e.g. chat, e-mail, web) as well as the outsourcing of computation and applications (e.g. cloud computing). In 2011, the number of Internet users has surpassed two billion (i.e., a third of the global population [130]) and the number of Internet hosts are approximately 800 million. Websites are reachable via a wide range of computing devices such as personal computers, tablet PCs, mobile phones. Also, users often have anywhere, any time access to the Internet.

Attackers are not indifferent to the evolution of the Internet. Often driven by a flourishing underground economy, attackers are constantly looking for vulnerabilities, misconfigurations and novel techniques to access protected and authorized systems, to steal private information, or to deliver malicious content. Traditional vulnerabilities such as buffer overflows or SQL injections are still exploited. However, new alternative attack vectors that leverage unconventional channels on a large scale (e.g. cloud computing) are also being discovered. To date, not much research has been conducted to measure the importance and extent of these emerging Internet threats. Conventional detection techniques cannot easily scale to large scale installations, and novel methodologies are required to analyze and discover bugs and vulnerabilities in these complex systems.

In this thesis, we advance the state of the art in large scale testing and measurement of Internet threats. We research into three novel classes of security problems that affect Internet systems that experienced a fast surge in popularity (i.e., ClickJacking, HTTP Parameter Pollution, and commercial cloud computing services that allow the outsourcing of server infrastructures). We introduce the first, large scale attempt to estimate the prevalence and relevance of these problems on the Internet.

Table des matières

Table des figures

Liste des tableaux

Première partie

Résumé

Chapitre 1

Introduction

Au début des années 90, Arpanet venait d'être mis hors service et Internet était limité à une collection d'environ 500.000 hôtes [48]. En 1991, Tim Berners-Lee publia son projet appelé le World Wide Web qu'il avait développé au CERN à Genève. L'idée derrière le projet World Wide Web était d'utiliser un programme spécifique, appelé navigateur Web, pour permettre l'accès et l'affichage de contenu hypertexte en utilisant une architecture client-serveur. Deux ans plus tard, son projet World Wide Web est tombé dans le domaine public sous la forme d'un système d'information collaboratif et indépendant du type de plateforme matérielle et logicielle utilisé [42]. De nouveaux navigateurs avec capacités graphiques, comme ViolaWWW et Mosaic, sont rapidement devenus la norme pour accéder aux documents fournis par le service World Wide Web. Ces documents ont été organisés autour de sites Web et distribués sur Internet. Ces premiers sites étaient simples et petits, et les fonctionnalités offertes étaient limitées par le coût de la technologie. Le contenu multimédia comme les images et la vidéo étaient rarement employé car le stockage de fichiers était lent et l'accès très cher. Par exemple, en 1989, un disque dur coûtait en moyenne 36$ par mégabyte [59].

Vingt ans plus tard, le scénario a radicalement changé. La figure 1.1 montre comment Internet s'est développé jusqu'atteindre environ 800 millions d'hôtes. Le nombre d'utilisateurs d'Internet a dépassé deux milliards en début 2011, doublé au cours des dernières cinq années, et approché un tiers de la population mondiale [130]. Dans le même temps, la capacité des disques durs a augmenté linéairement sous l'influence du marché dont la demande de stockage de données n'est jamais satisfaite [103] (Fig. 1.2).

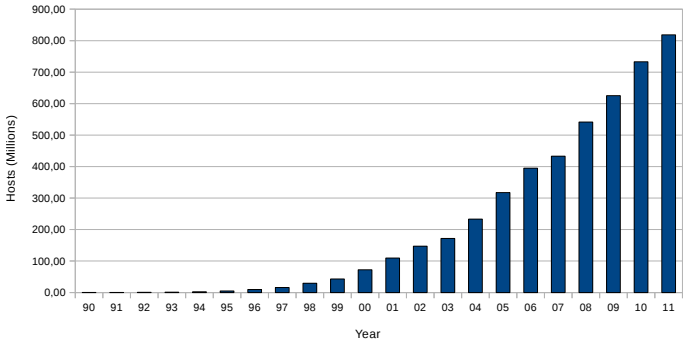


Fig. 1.1: Nombre de hôtes Internet par an

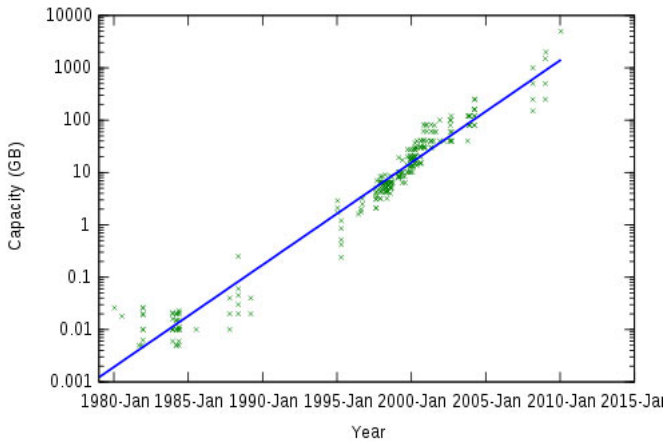


FIG. 1.2: Histoire de la capacité des disques durs

Aujourd'hui le stockage est moins cher que jamais, par exemple moins de 10 cents par gigabyte, et la puissance de calcul est disponible au prix des matières premières.

Alors que dans le passé, Internet était principalement utilisé pour accéder du contenu provenant de sites Web sous forme de documents hypertextes, il permet désormais l'accès à des services (comme le chat, l'e-mail ou le Web) ou encore à des capacités de calcul et à des applications externalisées (cloud computing, par exemple).

Le Web est une des principales manières dont les gens utilisent Internet aujourd'hui. On estime que les trois moteurs de recherche les plus populaires que sont Google, Bing et Yahoo indexent aujourd'hui plus de 13 milliards de pages Web [49]. Les sites Web sont accessibles depuis les ordinateurs personnels, les tablettes PC, les téléphones mobiles, les points d'accès et les hotspots publics y compris les aéroports, les gares, les bars, ou places de la ville qui offrent souvent un accès Internet gratuit. Les opérateurs de téléphonie offrent des contrats pour l'accès Web en utilisant les réseaux mobiles partout et 24/7. Toutes sortes de services, comme la réservation d'un vol ou l'accès aux comptes bancaires sont maintenant disponibles sur le Web de façon confortable.

Les applications complexes qui étaient auparavant installées sur les ordinateurs des utilisateurs sont maintenant déployées sur Internet sous forme d'applications Web. Celles-ci ne sont désormais plus qu'une simple collection de documents statiques; elles ont maintenant évolué vers des applications complexes qui fournissent des centaines de fonctionnalités via des pages générées dynamiquement.

L'utilisation combinée de coté client et serveur a permis aux développeurs de fournir des interfaces graphiques hautement sophistiquées avec un *look-and-feel* et des fonctionnalités qui étaient auparavant réservés aux applications de bureau traditionnelles. Dans le même temps, beaucoup d'applications web permettent également l'aggrégation de contenus hétérogènes en combinant les informations et des fonctionnalités fournies par différentes sources. Aujourd'hui, même de simples applications peuvent utiliser des architectures multi-tiers et fournir un contenu multimédia riche via une interface graphique sophistiquée. Elles peuvent impliquer un très grand nombre de lignes de code, une combinaison de langues multiples et faire appel à des centaines de composants interconnectés.

Malheureusement, en raison de leur grande popularité et du nombre croissants d'utilisateurs, les applications web sont également devenues la cible privilégiée des attaquants. Selon l'étude menée par Christey et Martin sur la base de données CVE de Mitre [46] le nom-

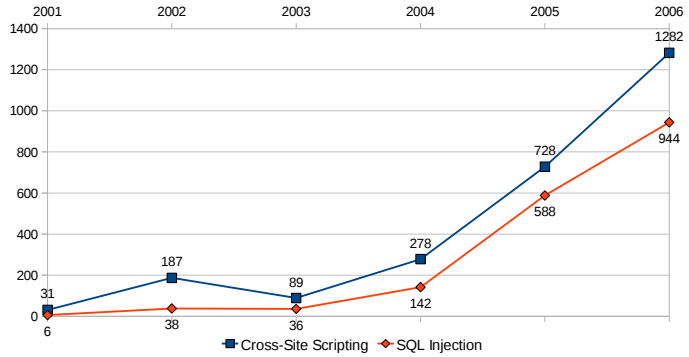


FIG. 1.3: Nombre de failles XSS et SQLi par an (MITRE)

bre de failles découvertes dans les applications web (par exemple, cross-site scripting et injection SQL) est en constante augmentation (Fig. 1.3).

En Juillet 2001, Symantec [132] a identifié une moyenne de 6.797 sites Web par jour hébergeant malware et autres logiciels potentiellement indésirables, y compris spyware et adware. Selon l'IBM Managed Security Services [126], dans l'année 2010 nous avons assisté au plus grand nombre de vulnérabilités divulgués dans l'histoire (8.562); un chiffre en augmentation de 27 pour cent par rapport à l'année 2009. 49 pour cent de ces vulnérabilités concernent les applications web, montrant que les attaquants déplacent leur attention vers les applications en ligne. En fait, aujourd'hui les attaques contre les applications web représentent plus de 60% des tentatives d'attaques totales observées sur Internet [76].

Lorsque nous analysons les comportements de ces attaques, il est possible d'observer qu'elles augmentent en quantité mais également en complexité. Par exemple, si les failles traditionnelles telles que le SQL injection et le cross-site scripting peuvent être utilisées pour voler des informations sensibles (par exemple de bases de données) ou lancer des attaques de phishing, beaucoup d'applications web sont maintenant exploitées pour les convertir en serveurs des logicielles malveillants. Selon SANS [76] la plupart des développeurs ne vérifient pas leurs applications contre les vulnérabilités classiques, tandis que nombreux outils simplifient aux attanquants la découverte et l'infection de plusieurs milliers de sites Web.

Outre la fourniture de contenu sous forme de pages web accessibles via un navigateur, Internet offre maintenant des services d'externalisation pour le calcul et des applications à distance (Cloud Computing). Le Cloud Computing a rapidement changé la façon dont les organisations gèrent leur infrastructure informatique et fournissent leurs services en ligne. Les entreprises qui ne peuvent pas se permettre d'installer une grande infrastructure chez eux, peuvent facilement louer un serveur sur le cloud et le gérer via des API. Les ressources sont disponibles en ligne comme un service offert par des fournisseurs de cloud computing. Ceux-ci louent aux organisations qui en ont le besoin leurs capacité de calculs et de stockage.

Les serveurs peuvent être rapidement démarrés, accédés et fermés à la demande, offrant à l'utilisateur un plus grand niveau de flexibilité par rapport aux serveurs traditionnels. Amazon est probablement le fournisseur de cloud computing le plus connu avec un augmentation de son activité de 375% en 2009 et 121% en 2010. Sur le blog de l'Amazon Web Services [33], Jeff Barr a rapporté que, à compter de la fin du deuxième trimestre de 2011, l'Amazon Simple Storage Service (S3) stocke environ 450 milliard d'objets et traite

jusqu'au 290.000 clients par seconde. Ceci équivaut à 64 objets pour chaque personne sur la planète.

Cet accroissement de popularité que les fournisseurs de cloud computing, comme Amazon, ont attendu est une conséquence directe de l'évolution du Web. En pratique, différents fournisseurs Internet offrant services d'hébergement (par exemple pour le stockage des emails et fichiers) ont déjà déplacé leurs infrastructures vers le Cloud. En fait, l'énorme quantité de données qu'ils traitent quotidiennement nécessite une réduction des coûts de stockage. Un de ces fournisseurs, DropBox, utilise le cloud de Amazon pour stocker les données personnelles de ses clients [74].

Les utilisateurs malveillants ne sont pas indifférents à l'évolution d'Internet. Les attaquants, souvent motivés par un florissant marché noir, sont constamment à la recherche des failles, erreurs de configuration et nouvelles techniques pour accéder aux systèmes, voler des informations privées, ou livrer du contenu malveillant. Même si les dépassements de tampon (buffer overflow), les format strings ou les injections SQL (parmi d'autres) sont encore exploitées, de nouveaux vecteurs d'attaque exploitant des canaux non conventionnels à grande échelle (par exemple les réseaux de cloud computing) ont été découverts. Récemment Amazon a mis en garde ses clients sur la présence d'images virtuelles corrompues dans son service de Elastic Computer Cloud (EC2) [113]. Le problème identifié par Amazon avec ces images est que certaines d'entre elles contiennent une clé SSH valide transformant le service SSH en une faille de sécurité potentielle (backdoor). Amazon a suggéré que toutes instances exécutant une image infectée devaient être considérées comme compromises, et que les services correspondants devaient être migré vers une nouvelle installation.

Les systèmes Internet comme le Web et le Cloud Computing sont déployées à grande échelle, ils sont complexes, et ils sont régulièrement exposés à de nouveaux risques de sécurité. De nouvelles méthodologies sont nécessaires pour analyser et découvrir les failles et les vulnérabilités de ces systèmes complexes. Les techniques classiques doivent être adapté pour faire face aux infrastructures à grande échelle dans lequelles les applications et les services sont déployés. En fait, les techniques traditionnelles utilisées pour évaluer la sécurité des applications classiques, par exemple installées côté client, ne sont plus suffisantes pour évaluer la sécurité de ces systèmes en ligne. Une des raisons, par exemple, c'est que le code source d'un service en ligne n'est pas normalement accessibles. En conséquence, ni les techniques d'analyse statique de code ([83, 131, 145]) ni les outils d'analyse dynamique ([50, 129]) ne peuvent pas être appliqués.

L'analyse des systèmes déployés à grande échelle est rendu encore plus complexe en raison de contraintes techniques liées au réseau, comme les problèmes de timing. Enfin, alors que les applications critiques comme les systèmes d'exploitation ou les services réseau (par exemple, SSH) sont normalement développés par des programmeurs conscients de l'importance de la sécurité, le code source des applications Internet et Cloud est souvent écrit par des développeurs avec très peu ou pas formés en sécurité informatique.

Comme les techniques existantes ne peuvent pas s'adapter facilement aux installations à grande échelle, peu de chercheurs font l'effort de mesurer l'extension de nouvelles menaces Internet. Pourtant, la recherche de mesure, quand elle est menée avec diligence et précision, peut avoir un impact significatif sur l'amélioration de la sécurité. La communauté scientifique considère une méthode *scientifique* lorsque les phénomènes qui sont étudiés peuvent être mesurés. La mesure devient donc le moyen permettant de quantifier les résultats d'une expérience et de juger l'entité d'un problème. Par exemple, dans la

lutte contre les campagnes de spam, en mesurant combien d'emails contiennent un contenu indésirable, on peut évaluer l'efficacité d'une solution anti-spam ou l'évolution du problème sur une longue période. La recherche en mesure est donc utile pour déterminer quelles classes de problèmes sont en augmentation et sur lequelles il est nécessaire d'investir. Les mesures, ainsi que les enquêtes de recherche, sont utiles aux chercheurs travaillant dans le même domaine; elles permettent la construction d'une base de connaissance commune.

1.1 Contributions

Cette thèse avance l'état de l'art dans le domaine des tests et de la mesure à grande échelle des menaces sur Internet.

Nous avons identifié trois nouvelles classes de problèmes de sécurité affectant les infrastructures Internet qui sont de plus en plus populaires et nous avons tenté d'estimer la prévalence et la pertinence de ces problèmes pour les utilisateurs d'Internet. En pratique, si l'on considère la sécurité Web dans son ensemble, les menaces traditionnelles (comme les XSS et SQLi) ont été largement étudiées [83, 131, 145] et décrites (par exemple par le projet OWASP Top 10 [109]). Il n'en demeure pas moins que de nouvelles classes de problèmes (telles que le clickjacking et la pollution des paramètres HTTP) font leurs apparitions et nous ne savons toujours pas de manière précise le risque qu'elles représentent pour les utilisateurs d'Internet.

Bien que ces menaces soient déjà connues, nous sommes les premiers à mener une étude systématique permettant de définir leur prévalence sur Internet. Notre objectif est de savoir si ces menaces sont connues des développeurs, si elles se produisent davantage dans certaines situations et d'évaluer le nombre de systèmes impactés. Pour répondre à ces questions, nous avons conçu et développé des outils automatisés permettant une analyse à grande échelle sur des milliers (voire des millions) de cibles potentiellement vulnérables.

Le premier problème étudié est connu sous le nom de "clickjacking" (ou détournement de clic). Le clickjacking exploite certaines propriétés visuelles des navigateurs modernes pour inciter l'internaute à fournir des informations confidentielles, à initier des transferts d'argent, en cliquant par example sur des bannières publicitaires frauduleuse, ou encore à effectuer toutes sortes d'action qui peuvent être déclenchées par un simple clic de souris. A titre d'exemple, il est connu que des différents logiciels malveillants se propagent à travers les réseaux sociaux (comme Facebook ou Twitter) en utilisant cette technique.

L'attaque fonctionne en présentant aux profils amis un lien avec un titre attractif (par exemple "LOL This girl gets OWNED after a POLICE OFFICER reads her STATUS MESSAGE" ou "Don't click"). Les internautes qui cliquent sur le lien vont sans le savoir "retwitter" le lien ou l'ajouter à la liste de leurs "J'aime". La figure 1.4 montre une attaque réelle de clickjacking utilisée pour propager un message parmi les utilisateurs de Twitter [94].

L'idée derrière une attaque de clickjacking est simple: une page malveillante est construite et insérée dans le contenu d'un site Web légitime dans le but d'inciter l'internaute à cliquer sur un élément qui est à peine ou pas du tout visible. Le clic peut donc déclencher des actions imprévues dans le contexte du site Web légitime. Le simple fait que ce soit la victime qui effectue (sans le savoir) le clic rend l'action valide du point de vue du navigateur.

La détection des tentatives de clickjacking n'est pas une opération triviale. Le défi principal est de simuler le comportement d'un humain qui interagit avec le contenu de

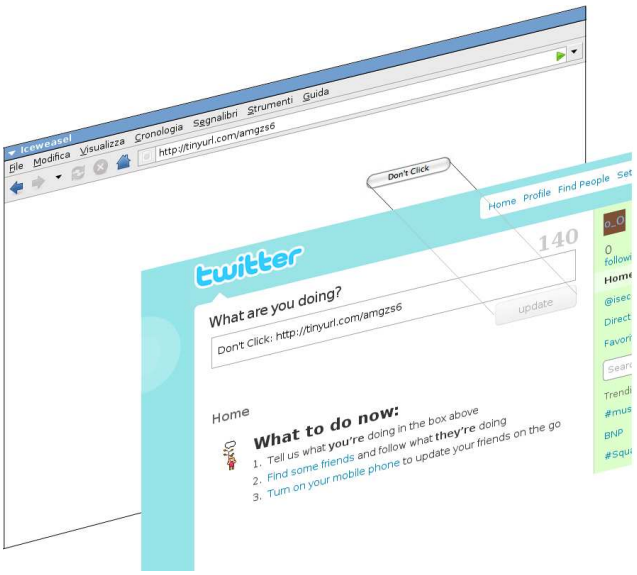


FIG. 1.4: Exemple de attaque Clickjacking contre Twitter

la page Web testée. Les solutions existantes telles que Sélénium [7] et Watir [8] simulent les actions de la souris à l'intérieur du navigateur lui-même en envoyant des événements vers les éléments qui composent la page. Bien que cette approche soit pratique pour tester l'aspect fonctionnel d'une application Web (on peut par exemple vérifier que les liens fonctionnent correctement), elle n'est pas adaptée à nos besoins particuliers. En fait, nous ne savons pas à priori sur quel élément l'utilisateur avait l'intention de cliquer et cela est en définitive le pré-requis à une attaque de type clickjacking. Par conséquent, nous ne voulons pas simuler les clics au niveau du navigateur lui-même (en utilisant une API dédiée), mais plutôt en exploitant les évenements natifs du systèmes d'exploitation; en d'autres termes la souris est contrôlée et déplacée au niveau de la fenêtre afin de simuler un véritable clic aux coordonnées de chaque élément. Cette approche garantit que le code JavaScript de la page est exécuté exactement dans les même conditions et le la même manière que lorsque l'action est effectuée par un vrai utilisateur.

Le second défi a été de pouvoir afficher correctement les pages d'applications Web qui (de plus en plus) construisent leur contenu de manière dynamique en exploitant les possibilités de "scripting" des navigateurs modernes (e.g. Javascript). Il aurait été possible (comme le font d'autres scanners) de récupérer le contenu des pages Web sans vraiment les afficher dans un véritable navigateur. Le défaut principal d'une telle approche est que le contenu dynamique des pages reste pratiquement inexploitable. Il était donc primordial d'afficher la page au sein même d'un vrai navigateur ceci afin de déclencher les mécanismes de mise en page standard de son contenu. Cette approche nous permet d'analyser la page telle qu'elle est censée apparaître à l'utilisateur; l'ensemble des son contenu dynamique étant construit de façon normale.

L'implémentation de notre système automatisé de test et de détection de menaces appelé *ClickIDS*, a été utilisé pour mener une étude empirique sur un million de pages Web afin d'estimer la prévalence des attaques de type clickjacking sur Internet.

Nous nous sommes ensuite concentrés sur une autre menace Web appelée pollution de paramètres HTTP (ou simplement HPP). L'HPP, depuis sa première annonce en

2009 [110], n'a pas vraiment retenu l'attention de spécialiste en sécurité. En étant les premiers à proposer une découverte automatisée des vulnérabilités HPP dans les applications web, nous avons pu mesurer parmi 5.000 sites populaires combien sont potentiellement vulnérables à cette nouvelle menace.

Les attaques HPP consistent à injecter dans les URL(s) des caractères qui sont normalement utilisés pour séparer les paramètres de la requête quelle représente. Si une application Web ne valide pas correctement ses paramètres, un utilisateur malveillant peut compromettre la logique de l'application pour effectuer des attaques aussi bien du côté client que du côté serveur. Ce type d'attaque permet donc d'altérer le comportement normal d'une application, en contournant les contrôles de validité, ou en permettant l'accès et l'exploitation de données qui sont normalement non disponibles.

Les failles HPP peuvent être exploitées de différentes manières et leur détection n'est pas facile. Le problème principal est lié à la complexité même des applications Web modernes ; en effet, leur contenu dynamique peut varier même quand la page est consultée avec les mêmes paramètres. Bannières publicitaires, flux RSS, statistiques en temps réel, gadgets et boîtes à suggestions ne sont que quelques exemples de contenus dynamiques susceptibles de changer le contenu de la page et cela chaque fois que la page est consultée.

Un autre défi provient du fait que certains paramètres n'affectent pas la logique de l'application, ils ne sont utilisés que pour stocker l'URL de la page Web. Ainsi, une injection effectuée sur ces paramètres là à pour conséquence de faire pointer l'application vers une autre URL. Même si cette technique est syntaxiquement très similaire à une vulnérabilité HPP, ce n'est pas un cas d'injection à proprement parler.

Dans cette thèse, nous proposons la première approche automatisée permettant la découverte des vulnérabilités HPP dans les applications Web. Notre approche comprend un composant qui injecte des paramètres de test dans l'application et d'un ensemble d'heuristiques permettant de déterminer si les pages générées par l'application contiennent de failles HPP. La faisabilité de notre approche a été démontrée grâce à un prototype *PAPAS*, que nous avons développé; celui-ci a permis d'effectuer une analyse à grande échelle de plus de 5.000 sites Web populaires.

Enfin, nous avons étudié les nouvelles menaces affectant le cloud computing à grande échelle. Le cloud computing a changé radicalement les infrastructures informatiques; on est passé de ressources logicielles et matérielles pré-payées à des services à la demande.

Plusieurs entreprises comme Amazon Elastic Compute Cloud (EC2) [19], Rackspace [21], IBM SmartCloud [24], Joyent Smart Data Centre [26] ou Terremark vCloud [22] offrent un accès à des serveurs virtuels, stockés dans leurs centres de données, sur une base horaire. Les serveurs peuvent être rapidement démarrés et fermés via des API, offrant ainsi au client davantage de flexibilité par rapport aux traditionnelles salles de serveurs. Ce changement de paradigme a fait évoluer les infrastructures informatiques des organisations, permettant aux petites sociétés qui ne pouvaient pas se permettre une grande infrastructure de créer et de maintenir des services en ligne avec facilité.

Malheureusement, alors que le contrat de confiance entre utilisateur et fournisseur est bien défini (l'utilisateur peut supposer que les fournisseurs de services comme Amazon et Microsoft ne sont pas malveillants), la relation de confiance entre le fournisseur d'image virtuelle et l'utilisateur n'est pas aussi claire.

Nous avons exploré les risques de sécurité associés à l'utilisation des serveurs virtuels (appelé AMIs) qui sont fournis comme service par les fournisseurs de cloud (par exemple Amazon). Sur plusieurs mois, nous avons conçu et effectué des tests de sécurité sur les

AMIs publiques afin d'identifier les failles et risques potentiels encourus aussi bien par les clients que par les fournisseurs de services. Nous avons analysé sur une grande échelle 5.000 images publiques fournies par Amazon, et nous avons identifié trois domaines pertinents d'un point de vue la sécurité: sécuriser les images contre des attaques externes, sécuriser les images contre des fournisseurs malveillants, et valider l'image afin d'empêcher l'extraction et l'exploitation des informations confidentielles (stokées sur le disque virtuel par le fournisseur).

Dans cette thèse, nous avons conçu un ensemble de nouvelles techniques permettant la mesure à grande échelle des menaces sur Internet. Ces techniques ont en commun l'utilisation de fonctionnalités tests et d'indexation permettant de traiter et d'analyser efficacement un grand nombre de cibles potentiellement vulnérables. Nous avons utilisé nos solutions pour mener des études à grande échelle de trois classes de problèmes qui affectent les systèmes Internet les plus importants et les plus populaires - comme le Web et le Cloud.

Nous croyons que dans l'avenir nos solutions resteront valables et permettront d'analyser des classes similaires de problèmes. A titre d'exemple, nous avons introduit, pour l'analyse des vulnérabilités Web, un système "browser-centric" permettant de prendre en charge le contenu dynamique généré par les applications modernes. En utilisant un vrai navigateur pour rendre ces pages, nous sommes capables de les analyser comme si ils seraient censés apparaître au utilisateur après le contenu dynamique a été générée. Cette technique peut facilement être adaptée et réutilisée dans l'analyse des classes de problèmes similaires, tels que les attaques qui exploitent les nouvelles fonctionnalités de HTML5 (par exemple, drag&drop).

1.2 Récapitulatif

Les contributions de cette thèse peut être résumées par les points suivants:

- Nous avons fait évoluer l'état de l'art dans le domaine des tests et de la mesure à grande échelle des menaces sur Internet.
- Nous analysons trois nouvelles classes de problèmes de sécurité affectant les systèmes Internet qui connaissent une augmentation rapide de leur popularité, comme par exemple les applications Web et les services de cloud computing.
- Nous introduisons la première tentative d'estimation à grande échelle de la prévalence et de la pertinence de ces problèmes sur Internet. Nous avons conçu et développé des outils automatisés permettant d'analyser des milliers (voire des millions) de cibles potentiellement vulnérables.
- Nous décrivons les défis auxquels nous avons été confrontés lors de nos tests effectués sur des applications et des services réels.
- Enfin, lorsque nous avons été en mesure de trouver des contacts, nous avons communiqué aux fournisseurs concernés les vulnérabilités découvertes et proposé des solutions. Certains d'entre eux ont reconnu être intéressés et ont mis en ouvre les contre-mesures que nous avons proposé.

1.3 Organisation de la thèse

Le reste de la thèse est organisé comme suit (les chapitres 4 et suivants sont rédigé en Anglais):

- Le chapitre 2 présente les solutions qui ont été conçues et développées pour mener l'étude de mesure décrite dans cette thèse. Ses sections (2.1, 2.2, 2.3) résument respectivement les chapitres 5, 6 et 7.
- Dans le chapitre 4 nous présentons les travaux de recherche qui sont liés à cette thèse.
- Dans le chapitre 5 nous proposons une nouvelle solution permettant la détection automatisée des attaques clickjacking. Nous décrivons *ClickIDS*, le système que nous avons conçu, mise en oeuvre et déployé pour analyser plus d'un million de pages web. Nous avons publié cette recherche dans les actes du 5ème Symposium ACM en Information, Computer and Communications Security (AsiaCCS 2010).
- Dans le chapitre 6 nous introduisons la première approche automatisée permettant la découverte de vulnérabilités HPP dans les applications Web. Grâce à notre prototype, appelé *PAPAS (PArameter Pollution Analysis System)*, nous avons effectué une analyse à grande échelle de plus de 5.000 sites Web populaires. Cette recherche a été publiée, et a reçu le Best Paper Award dans la 18e édition annuelle de l'Annual Network and Distributed System Security Symposium (NDSS 2011).
- Le chapitre 7 explore les risques de sécurité associée à l'utilisation des serveurs virtuels (appelés AMIs) fournis par les fournisseurs de cloud (par exemple Amazon). Nous décrivons la conception et la mise en ouvre d'un système automatisé, appelé *SatanCloud*, que nous avons utilisé pour analyser la sécurité de 5.000 images publiques fournies par le service EC2 de Amazon. Nous avons publié cette recherche dans les actes de la 11e édition de la Computer Security track au 27th ACM Symposium on Applied Computing (SEC@SAC 2012).
- Enfin, dans le chapitre 8 nous concluons et discutons les limites et les possibilités d'améliorations de nos contributions.

Chapitre 2

Principales Contributions

Dans ce chapitre, nous présentons les solutions conçues et développées pour mener les études de mesure descrites dans la thèse. Les sections suivantes (2.1, 2.2, 2.3) résument respectivement les chapitres 5, 6 et 7.

2.1 Détournement de Clic (Clickjacking)

Dans cette section, nous présentons notre approche pour simuler les clics des utilisateurs sur les éléments d'une page web en cours d'analyse, et pour détecter la conséquence de ces clics en termes d'attaque de clickjacking. Notre technique repose sur un vrai navigateur pour charger et rendre une page Web. Lorsque la page a été rendu, nous allons extraire les coordonnées de tous les éléments cliquables. De plus, nous contrôlons par programme les souris et le clavier pour bien faire défiler la page Web et cliquez sur chacun de ces éléments.

La figure 2.1 montre l'architecture de notre système. Il se compose de deux éléments principales: une unité de test qui est en charge de effectuer les clics, et une unité de détection qui est responsable d'identifier éventuelles tentatives de clickjacking sur la page analysée.

L'*unité de détection* combine deux plugins qui fonctionnent parallèlement pour analyser les clics effectués par l'unité de test. Le premier plugin que nous avons développée détecte possibles elements cliquables qui se chevauchent. Pour compléter cette solution, nous avons également adopté NoScript, un outil qui a récemment introduit une fonction anti-clickjacking. Nos résultats expérimentaux montrent que la combinaison des deux techniques de détection différentes réduit le nombre de faux positifs.

L'*unité de test* contient un plugin que extrait les coordonnées des éléments cliquables rendu sur la page, et un composant browser-indépendant qui déplace la souris sur ces coordonnées et simule les clics de l'utilisateur. En outre, l'unité de test est responsable de la navigation web en tapant dans la barre d'adresse l'URL de la page web de visiter.

Résultats Nous avons organisé nos expériences pendant environ deux mois, en visitant un total de 1.065.482 pages web avec une moyenne de 15.000 pages par jour. Environ 7% de ces pages ne contiennent aucun élément cliquable - un signe que la page a été fermée ou qu'elle est encore en construction. Les pages restantes contenait un total de 143,7 millions d'éléments cliquables (soit une moyenne de 146,8 éléments par page).

37,3% des pages visitées contenaient au moins un IFRAME, alors que seulement 3,3%

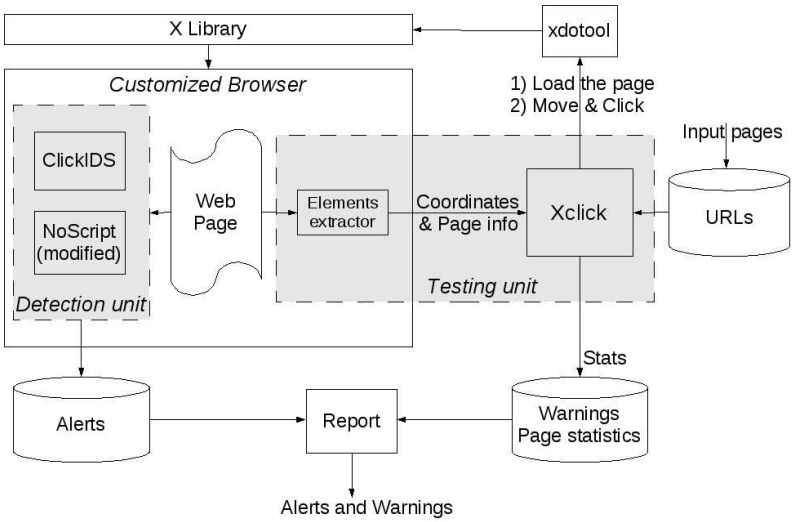


FIG. 2.1: Architecture de ClickIDS

	Valeur	Taux
Pages visitées	1,065,482	100%
Inaccessibles ou vides	86,799	8.15%
Valides	978,683	91.85%
Avec IFRAMEs	368,963	37.70%
Avec FRAMES	32,296	3.30%
Avec (I)FRAMEs transparents	1,557	0.16%
Eléments cliquables	143,701,194	146.83 el./page
Performance de vitesse	71 jours	15,006 pages/jour

TAB. 2.1: Statistiques des pages visitées

des pages incluent un FRAME. Cependant, seulement 930 pages contenaient IFRAMEs transparentes, et 627 pages contenaient IFRAMEs partiellement transparents. Ceci suggère que bien que les IFRAMEs sont couramment utilisés dans une large fraction de sites Internet, l'utilisation de la transparence est encore assez rare (en fait en représente que 0,16% des pages visitées). Le tableau 2.1 résume ces statistiques.

Le tableau 2.2 montre le nombre de pages sur lesquelles notre outil a généré une alerte. Les résultats indiquent que les deux plugins ont soulevé un total de 672 alertes (137 pour ClickIDS et 535 pour NoScript) - en moyenne une alerte tous les 1.470 pages. Cette valeur descend à 6 (une tous les 163.000 pages), si nous considérons les cas où les deux plugins rapportent un attaque. Notez que NoScript a été responsable de la plupart des alertes.

Pour mieux comprendre les alerts qui correspondent aux attaques réelles ou faux positifs, nous avons analysé manuellement toutes les alertes en visitant les pages web correspondantes. Les résultats de notre analyse sont rapportés dans les trois dernières colonnes du tableau 2.2. Environ 5% des alertes soulevées au cours de nos expériences impliqués un FRAME pointant vers le même domaine de la page principale. Comme il est peu probable qu'un site serait tenter de tromper l'utilisateur en cliquant sur un élément caché du site lui-même, nous avons marqué tous ces messages comme faux positifs. Nous avons décidé de parcourir manuellement certains des ces pages pour avoir un aperçu des conditions

	Total	Vrais Positifs	Borderlines	Faux Positifs
ClickIDS	137	2	5	130
NoScript	535	2	31	502
Les deux	6	2	0	4

TAB. 2.2: Résultats d'analyse clickjacking

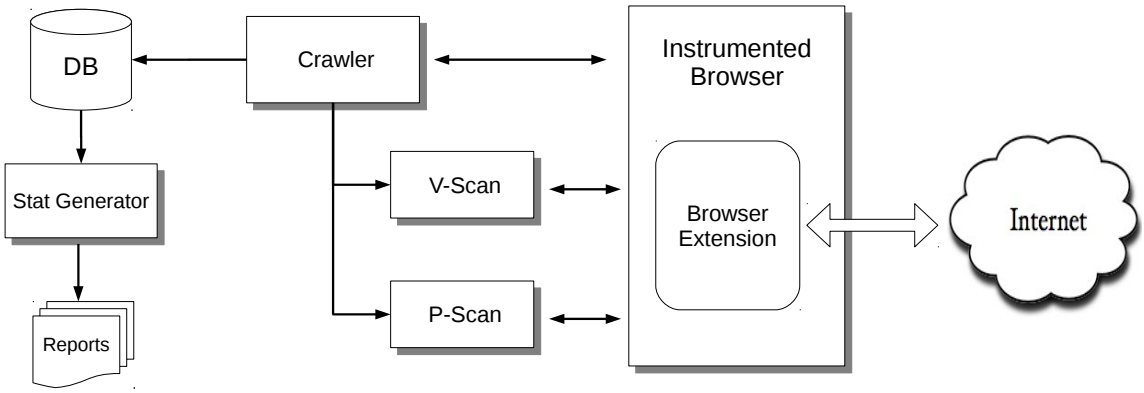


FIG. 2.2: Architecture de PAPAS

qu'ont tendance à provoquer faux positifs dans les deux plugins.

Nous avons ensuite analysé les pages contenant des FRAMEs inter-domaines. Dans cet ensemble, nous avons identifié un certain nombre des cas intéressants. Nous avons décidé de diviser ces cas dans deux catégories: les borderlines (pages difficiles à classer comme clickjacking) et les vrais positifs (tentatives réelles de clickjacking).

2.2 Pollution de Paramètres HTTP (HPP)

Dans cette section, nous introduisons la première approche automatisée pour la découverte de vulnérabilités HPP dans les applications web. Grâce à notre prototype, appelé *PAPAS (PArameter Pollution Analysis System)*, nous avons effectué une analyse à grande échelle de plus de 5.000 populaires sites web.

PAPAS se compose de quatre principaux composantes: un navigateur, un robot et deux scanners. La première composante, le navigateur instrumenté, est responsable de aller chercher les pages web, de rendre leur contenu, et d'extraire toutes les liens et les formulaires contenues dans la page. La deuxième composante est un robot qui communique avec le navigateur à travers un canal bidirectionnel. Ce canal est utilisé par le robot pour informer le navigateur sur l'URL qui doit être visités, et sur les formulaires qui doivent être soumis. Le canal est également utilisé pour récupérer les informations recueillies à partir du navigateur.

Chaque fois que le robot visite une page, il passe les informations extraites aux deux scanners afin qu'ils puissent être analysé. Le scanner de priorité (P-Scan) est chargé de déterminer comment la page se comporte quand il reçoit deux paramètres avec le même nom. Le scanner de vulnérabilité (V-Scan) teste une page pour déterminer s'est vulnérable aux attaques HPP; il injecte un paramètre de test dans l'un des paramètres existants et il

analyse la page générée. Les deux scanners communiquent avec le navigateur instrumenté afin d'exécuter leurs tests.

Toutes les informations collectées sont stockées dans une base de données et en suit analysés. L'architecture générale du système est représentée en figure 2.2.

Mise en ouvre Le navigation instrumenté est implémenté avec un plugin pour Firefox et le reste du système est développé en Python. Tout les composants communiquent permi eux via sockets TCP/IPs.

Le plugin a été développé en utilisant la technologie offerte par l'environnement de développement de Mozilla: un mélange de Javascript et XML User Interface Language (XUL). Nous utilisons XPConnect pour accéder aux composants XPCOM de Firefox. Ces composants sont utilisés pour invoquer les requêtes GET et POST et pour communiquer avec les composants d'analyse.

PAPAS offre trois modes de fonctionnement: le *mode rapide*, le *mode étendu* et le *mode assisté*. Le mode rapide vise à tester rapidement un site Web jusqu'à des vulnérabilités potentielles sont découverts. Quand une alerte est générée, l'analyse se poursuit, mais la composante V-Scan n'est plus invoquée (pour améliorer la vitesse du scan). En mode étendu, l'ensemble du site est testé de manière exhaustive et tous les problèmes potentiels et les injections sont enregistrés. En fin, le mode assisté permet au scanner d'être utilisé de manière interactive. Autrement dit, les pages ou les paramètres spécifiques peuvent être testées par la préséance des vulnérabilités HPP. Le mode assisté peut être utilisé par professionnels de la sécurité pour mener une évaluation semi-automatique d'un site web ou pour tester pages HTML qui nécessitent d'authentification.

PAPAS est également personnalisable et tout les paramètres comme la profondeur, le nombre d'injections, le temps d'attente entre deux requêtes, ainsi que le timeout sont configurables par l'analyste. Nous avons créé une version en ligne de PAPAS qui permet aux développeurs et mainteneurs d'applications web de scanner leur propre site. L'URL de ce service est `http://papas.iseclab.org`.

Résultats Nous avons utilisé PAPAS pour analyser automatiquement une liste des 5.000 sites web recueillis dans la base de données publique d'Alexa [29]. Le but de nos expériences a été de analyser rapidement le plus grand nombre des sites Web pour mesurer la prévalence des vulnérabilités HPP sur Internet. Pour optimiser la vitesse des tests, nous avons considéré seulement ces liens qui contiennent au moins un paramètre. Nous avons limité l'analyse aux cinq instances par page (une page avec une chaîne de requête différente est considérée comme une nouvelle instance). Le timeout global a été fixée à 15 minutes par site et le navigateur a été personnalisé pour charger rapidement les pages et fonctionner sans aucune interaction d'utilisateur. Par ailleurs, nous avons désactivé les popups, le chargement d'images, et tout les plugins du contenu actif comme Flash ou Silverlight. Un composant externe a été configuré pour surveiller et redémarrer le navigateur au cas où il ne répondait plus.

En 13 jours d'expériences, nous avons scanné 5.016 sites Web, correspondant à un total de 149.806 pages. Pour chaque page, PAPAS a généré une quantité variable de requêtes, en fonction du nombre des paramètres détectés. Les sites ont été distribués sur plus de 97 pays et sur différentes catégories d'Alexa. Le tableau 2.3 résume les 15 catégories contenant le plus grand nombre d'applications testées.

Pour chaque site, le P-Scan a testé chaque page afin d'évaluer l'ordre dans lequel les paramètres GET sont examinés par l'application lorsque deux occurrences d'un même

Catégories	# de applications testées	Catégories	# de applications testées
Internet	698	Government	132
News	599	Social Networking	117
Shopping	460	Video	114
Games	300	Financial	110
Sports	256	Organization	106
Health	235	University	91
Science	222	Others	1401
Travel	175		

Tab. 2.3: Les TOP15 catégories de sites analysés

Priorité	# de sites	
Dernière	2,237	(44.60%)
Première	946	(18.86%)
Union	381	(7.60%)
Incohérent	1,251	(24.94%)
Inconnu	201	(4.00%)
Total	5,016	(100.00%)
Erreurs dans le DB	238	(4.74%)

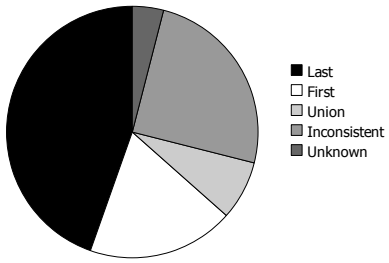


Fig. 2.3: Priorité des paramètres lorsque deux occurrences d'un même paramètre sont spécifiés

paramètre sont spécifiés (figure 2.3). La première colonne indique le type de priorité: *dernière* et *première* indiquent que toutes les les pages analysées de l'application considèrent le dernier (ou le premier) valeur. *Union* indique que les deux paramètres sont combinés pour former une seule valeur, généralement par concaténation simplement, avec espace ou virgule. La priorité est *incohérent* quand certaines pages favorisent le premier paramètre et d'autres le dernier. Cet état, ce qui représente un total de 25% des applications analysées, est généralement une conséquence du fait que le site a été développé en utilisant une combinaison de technologies hétérogènes (par exemple PHP et Perl).

Enfin, le scanner a découvert que 238 applications (prés de 5%) ont soulevé une erreur SQL lorsque deux paramètres dupliqués ont été spécifiés. Notez que l'utilisation de deux paramètres avec le même nom est une pratique courante dans le développement des applications web, et la plupart des langages de programmation fournissent spéciaux fonctionnalités pour accéder aux valeurs multiples. Néanmoins, nous ont été surpris de noter comme nombreuses sites (par exemple banques, sites gouvernementales ou éducatifs) échouent à traiter correctement leurs paramètres.

PAPAS a découvert que 1.499 sites (29,88% du total) contenaient au moins une page vulnérable au injection de paramètres HTTP. Qui est, l'outil a été en mesure d'injecter automatiquement un paramètre (encodé) dans l'un ses paramètres, et de vérifier que sa version décodée a été incluse dans l'un des URLs (liens ou formulaires) de la page résultante. Cependant, le fait qu'il est possible d'injecter un paramètre ne révéle pas d'informations sur

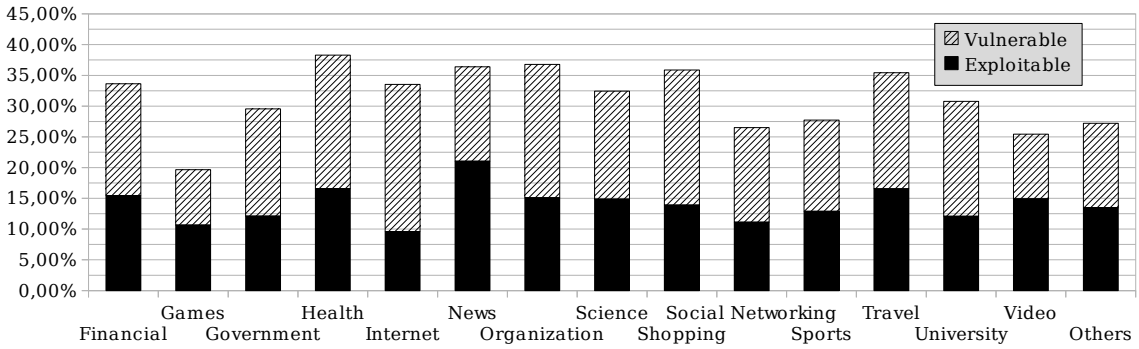


FIG. 2.4: Taux de vulnérabilité pour chaque catégorie

les conséquences de l'injection (nous analysons en détail ces conséquences en section 6.4). La figure 2.4 résume les résultats par catégorie.

2.3 Risques liés au Elastic Compute Cloud

Dans cette section, nous résumons le système automatisé, appelé *SatanCloud*, que nous avons conçu et mis en ouvre pour explorer les risques de sécurité associées à l'utilisation des serveurs virtuels (appelé AMIs) fournis par les fournisseurs de cloud (par exemple Amazon).

L'architecture de notre système est mis en évidence en figure 2.5 et se compose de trois éléments principaux.

Le *robot* est responsable de l'instanciation de l'AMI et la récupération de ces credentials de login (Amazon n'est pas responsable des credentials configurés dans ses images publiques). Après que un AMI a été correctement instancié par le robot, il est testé par deux scanners différents. Le *remote scanner* recueille la liste des ports ouverts [1] en utilisant l'util `NMap`, et télécharge la page d'index d'une éventuelle application web installée. Le *local scanner* est responsable de le téléchargement et l'exécution d'un ensemble de tests. Ces tests sont emballés dans une archive auto-extractible (la suite de testage) qui vient téléchargé dans l'AMI et exécuté à distance avec de privilèges administratifs. En outre, le local scanner analyse le système pour detecter vulnérabilités connues en utilisant l'outil Nessus [125]. Pour les AMIs exécutant Microsoft Windows, les scripts d'automatisation des tâches est compliquée par les limitées fonctionnalités d'administration à distance offrent par l'environnement Windows. Dans ce cas, nous avons monté le disque et transféré les données en utilisant les sous-système `SMB/NetBIOS`. Nous avons ensuite utilisé l'outil `psexec` [120] pour exécuter les commandes à distance et invoquer les tests.

La suite de testage transféré par le local scanner comprend 24 tests qui sont regroupés en 4 catégories: général, réseau, confidentialité et sécurité. La liste complète de tests est résumée en tableau 2.4.

La catégorie *général* recueille les informations généraux sur le système (par exemple la version de Windows), la liste des processus, l'état du système de fichiers (par exemple les partitions montées), la liste des paquets installés et la liste des modules chargés. Cette

[1] Parce que Amazon ne permet pas des scans de ports externes, nous avons d'abord créé un réseau privé virtuel avec l'AMI et puis scanné le machine à travers ce tunnel.

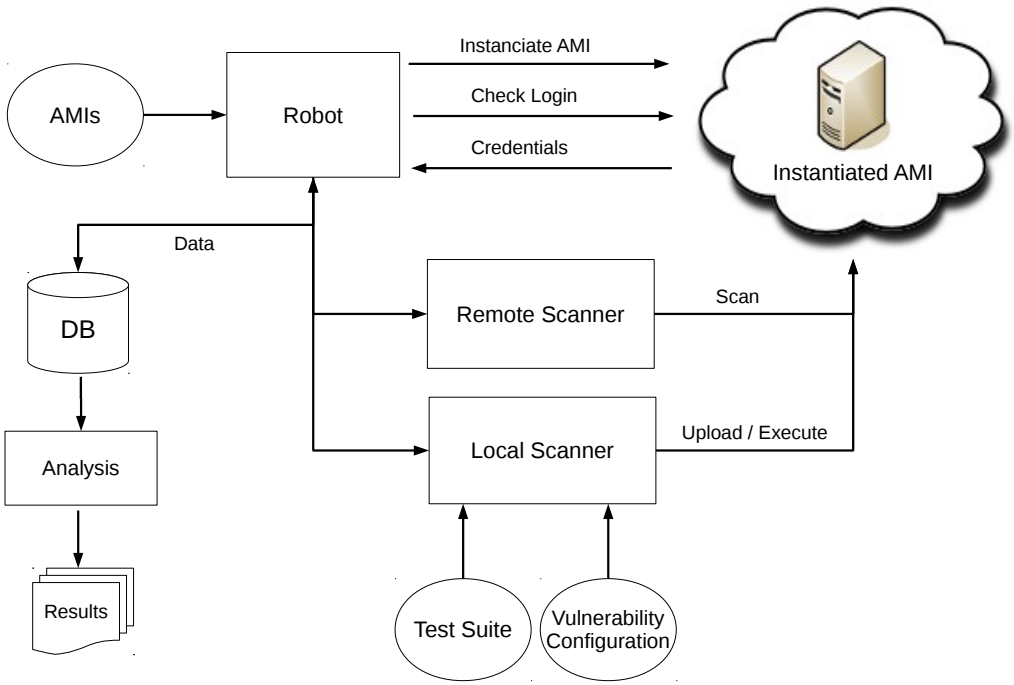


FIG. 2.5: Architecture de SatanCloud

catégorie contient également des scripts qui permettent d'enregistrer une copie de fichiers intéressants, comme les emails (par exemple `/var/mail`), les logs (par exemple `/var/log` et `%USER\Local Settings`) et les applications web installées (par exemple `/var/www` et `HKEY_LOCAL_MACHINE\SOFTWARE`).

Les tests de *confidentialité* se concentrent sur la recherche des informations sensibles qui peuvent avoir été oubliées par l'utilisateur qui a publié l'AMI. Celles-la incluent, par exemple, les clés privées pas protégées (sans mot de passe), les fichiers historiques (par exemple de shell) et le contenu des répertoires sauvés par les tests généraux. Une autre tâche importante de cette suite de tests est de scanner le système de fichiers pour récupérer le contenu du fichiers supprimés (ou undeleted files, réf. section 7.4.3).

La catégorie *réseau* récupère les information de réseau, tels que les répertoires partagés et la liste des sockets ouverts. Ces-ci peuvent être utilisée pour vérifier si l'image établit des connexions suspectes.

Enfin, la suite *test de sécurité* contient un certain nombre des outils de sécurité pour Windows et Linux. Certains de ces outils recherchent la preuve de rootkits connus, de chevaux de troie et de backdoors comme `Chkrootkit, RootkitHunter` et `RootkitRevealer`. Autres tests contrôlent l'existence de processus ou fischers cachées au utilisateur (`PsTools/PsList` et `unhide`). Cette catégorie comprend également un antivirus (`ClamAV`) pour vérifier la présence de malware dans l'image.

En fin, la suite test de sécurité recherche des credentials qui pourraient avoir été oublié sur le système (par exemple des mots de passe pour bases de données et login, ou des clés SSH publiques). En fait, ces credentials pourraient potentiellement être utilisés comme backdoor pour s'authentifier et connecter à la machine à distance. Nous avons également analysé la configuration de `sudo` pour vérifier si ces credentials permettraient l'exécution de commandes avec privilèges administratifs.

Test	Type	Note	SE
System Information	Général	-	Windows + Linux
Logs/eMails/WWW Archive	Général	-	Linux
Processes and File-System	Général	-	Windows + Linux
Loaded Modules	Général	lsmod	Linux
Installed Packages	Général	-	Linux
General Network Information	Réseau	Interfaces, routes	Windows + Linux
Listening and Established Sockets	Réseau	-	Windows + Linux
Network Shares	Réseau	Enabled Shares	Windows + Linux
History Files	Confidentialité	Common Shells + Browsers	Windows + Linux
AWS/SSH Private Keys	Confidentialité	Loss of sensitive info	Linux
Undeleted Data	Confidentialité	(Only on X AMIs)	Linux
Last logins	Confidentialité	-	Linux
SQL Credentials	Confidentialité/Sécurité	MySQL and PostgresSQL	Linux
Password Credentials	Confidentialité/Sécurité	Enabled Logins	Windows + Linux
SSH Public Keys	Sécurité	Backdoor access	Linux
Chkrootkit	Sécurité	Rootkit	Linux
RootkitHunter	Sécurité	Rootkit	Linux
RootkitRevealer	Sécurité	Rootkit	Windows
Lynis Auditing Tool	Sécurité	General Security Issues	Linux
Clam AV	Sécurité	Antivirus	Windows + Linux
Unhide	Sécurité	Processes/Sockets Hiding	Linux
PsList	Sécurité	Processes Hiding	Windows
Sudoers Configuration	Sécurité	-	Linux

Tab. 2.4: Les tests inclus dans la suite de testage

Résultats Sur une période de cinq mois, entre Novembre 2010 et Mai 2011, nous avons utilisé notre système automatisé pour instancier et analyser toutes les AMIs publiques de Amazon et disponibles dans les quatre régions de Europe, Asie, États-Unis Est et Ouest. Au total, le catalogue de ces régions contenait 8.448 AMIs Linux et 1.202 Windows.

Grâce à nos expériences nous avons identifié trois principales menaces liées, respectivement, à assurer l'image contre des attaques externes, à assurer l'image contre des fournisseurs malveillants, et à valider l'image pour empêcher l'extraction et l'abuse des informations privées (par exemple emmagasinés sur le disque virtuel par le fournisseur). Pour une discussion complète des nos résultats, s'il vous plaît de consulter la section 7.4.

Deuxième partie

These

Chapitre 3

Introduction

In the early 90's ARPANET had just been decommissioned and the newborn Internet network was a collection of about 500,000 hosts [48]. In 1991 Tim Berners-Lee published his project called the World Wide Web that he had developed at CERN in Geneva. The idea behind the World Wide Web project was to use a special program, called browser, to access and render the content of a hypertext document by using a client-server architecture. Two years later, the World Wide Web project was released into the public domain to provide a collaborative information system independent of the type of hardware and software platform, and physical location [42]. Browsers with graphical capabilities such as ViolaWWW and Mosaic rapidly became the standard to access the documents provided through the World Wide Web service. These documents were organized in websites and delivered over the Internet. These first websites were simple and small, and the functionalities providing were limited by the cost of the technology. Multimedia content such as images and video were rarely employed because storage was slow to access and expensive. For example, in 1989 a hard-disk would have cost an average of 36$ per megabyte [59].

Twenty years later the scenario has radically changed. Figure 3.1 shows how the Internet has grown over time to reach approximately 800 million hosts. The number of Internet users surpassed two billion in early 2011, doubled in the last five years, and approached a third of the global population [130]. At the same time, the capacity of hard disks has grown linearly under the influence of the market whose demand for data storage is never satisfied [103] (Fig. 3.2). Now storage is cheaper than ever, e.g. less than 10 cents per GB, and computing power is available at commodity price.

As a consequence, while in the past the Internet was mainly used to offer content that was organized around simple websites in form of hypertext documents, now the Internet provides both content and services (e.g. chat, e-mail, web) as well as outsourcing computation and applications (e.g. cloud computing).

The Web is one of the main ways in which people use the Internet today. Three popular and important search engines such as Google, Bing and Yahoo are estimated to index at least 13 billion unique web pages [49]. Websites are reachable using standard personal computers, tablet PCs, mobile phones or Internet points, and public hotspots including airports, train stations, bars, or town squares which often provide free wireless access to the Internet. Phone operators offer low-cost contracts for accessing the Web by using mobile networks everywhere and 24/7. Any sort of service from booking a flight to online banking is now made available over the Web in a comfortable way.

Complex applications that before were installed on the user's computers are now de-

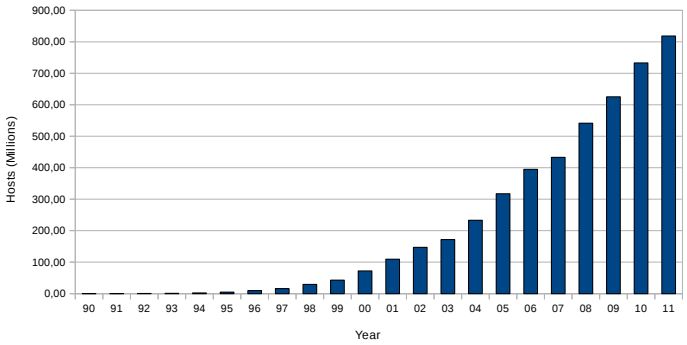


FIG. 3.1: Number of Internet hosts by year

ployed over the Internet in the form of web applications. Web applications have evolved from a simple collection of static HTML documents to complex, full-fledged applications containing hundreds of functionalities and dynamically generated pages. The combined use of client and server-side scripting allows developers to provide highly sophisticated user interfaces with the look-and-feel and functionalities that were previously reserved to traditional desktop applications. At the same time, many web applications have evolved into mesh-ups and aggregation sites that are dynamically constructed by combining together content and functionalities provided by different sources. Today even simple applications may use multi-tier architectures and provide sophisticated user interfaces with rich multimedia content. They may involve an enormous number of lines of code, combining multiple heterogeneous languages and hundreds of network-connected components.

Unfortunately, because of their high popularity and a user base that consists of millions of users, web applications have also become prime targets for attackers. At the same time, the number of flaws discovered in web applications (e.g. cross-site scripting and SQL injection) have constantly increased over time (Fig. 3.3), as Christey and Martin have reported in their study conducted over the MITRE's CVE database [46].

In July 2001, Symantec [132] identified an average of 6,797 websites per day hosting malware and other potentially unwanted programs including spyware and adware. According to the IBM Managed Security Services [126], 2010 witnessed the largest number of vulnerability disclosures in history (8,562), equal to a 27 percent increase over 2009. 49 percent of these vulnerabilities concerned web applications, showing that attackers are moving their attention toward online applications. Attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet [76].

When we analyze the behaviors of these attacks, it is possible to observe that they are increasing in quantity and in complexity as well. For example, if traditional flaws such as SQL injection and cross-site scripting may be used to steal sensitive information from application databases and to launch authentic-looking phishing attacks, many web applications are now being exploited to convert trusted websites into malicious servers serving content that contains client-side exploits. According to SANS [76], most website owners fail to scan their application for common flaws. In contrast, from the attacker's point of view, custom tools, designed to target specific web application vulnerabilities simplify the discovery and infection of several thousand of websites.

Beside providing content in the form of web pages accessible via a browser, the Internet now offers outsourcing services for remote computation or storage under the name of

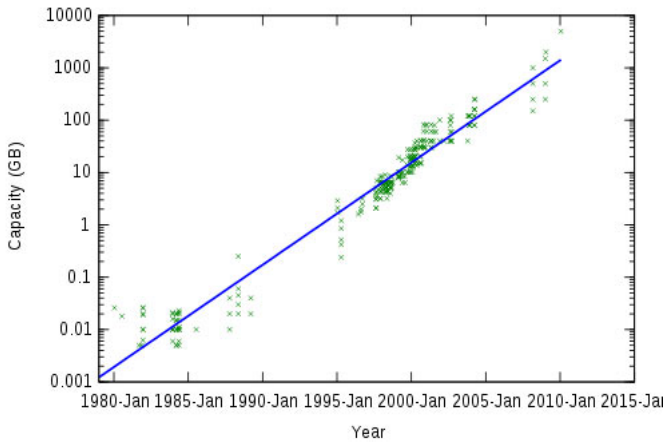


FIG. 3.2: History of hard-disks capacity

cloud computing. Cloud Computing is quickly changing the way organizations deal with IT infrastructure and are providing online services. Companies which cannot afford a large infrastructure to create and maintain online services with ease can quickly rent a server on the cloud and manage it via application programming interfaces. Resources are made available online as a service from cloud computing providers that rent them to organizations that need computing power or storage. Servers can be quickly launched, accessed and shut down on-demand, offering the user a greater level of flexibility compared to traditional server rooms. Amazon, probably the most used cloud computing provider, expected a grow of 375% in 2009 and 121% in the following year. On the Amazon Web Service Blog [33], Jeff Barr reported that, as of the end of the second quarter of 2011, Amazon Simple Storage Service (S3) stores about 450 billion objects and processes up to 290,000 requests per second at peak times. This is equivalent to 64 objects for each person on planet earth.

This increment in popularity that cloud computing providers, such as Amazon, have expected is a direct consequence of the evolution of the Web. In fact, different web providers that offer hosting services (e.g. email accounting and file storage) have already moved their infrastructures to the Cloud. The huge amount of data that they process daily calls for a reduction of the storage costs. One of these providers is DropBox, a well-known web-based file hosting service, that uses the Amazon's cloud to store personal data uploaded by its customers [74].

Malicious users are not indifferent to the evolution of the Internet. Attackers, often driven by a flourishing underground economy, are constantly looking for bugs, misconfigurations and novel techniques to access protected and authorized systems, to steal private information, or to deliver malicious content. Even if buffer overflows, format strings or SQL injections (among many) are still exploited, new alternative attack vectors that leverage unconventional channels on a large scale (e.g. cloud computing networks) have been discovered. Recently Amazon warned its customers about the presence of compromised images in its Amazon Elastic Computer Cloud (EC2) service [113]. The problem that Amazon identified with these images is that some of them contain a valid SSH key, turning the SSH service into a potential backdoor. Amazon suggested that any server instance running an infected image should be considered compromised, and that the services running on those instances should be migrated to a new, clean installation.

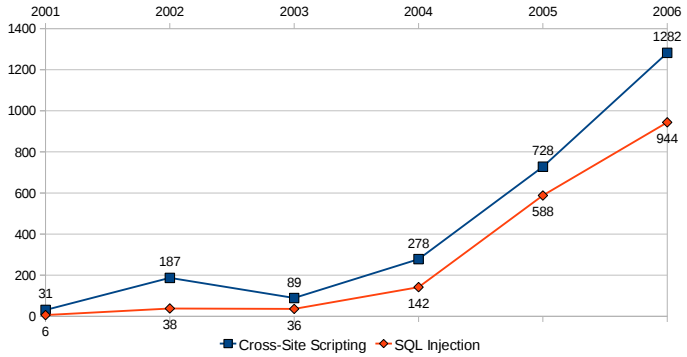


FIG. 3.3: Number of XSS and SQLi flaws by year (MITRE)

What Internet systems like the Web and Cloud Computing have in common is that they are deployed on a large scale, they are complex, and they are exposed to new security risks. Novel methodologies are required to analyze and discover bugs and vulnerabilities in these complex systems, and conventional techniques must be adapted to cope with the large scale infrastructures in which applications and services are deployed. In fact, traditional techniques used to evaluate the security of classic applications installed on the user side are not anymore adequate for evaluating the security of these online systems. One reason, for instance, is that the source code of an online service is normally not accessible and neither static code analysis techniques ([83, 131, 145]) nor fine-grained dynamic analysis tools ([50, 129]) cannot be applied. Moreover, the analysis of systems deployed on a large scale is complicated by network constraints like timing issues. Finally, while critical applications like operating systems or network services (e.g. SSH) are developed by security-aware programmers, both web and cloud applications' source code is often written by developers with very little or no security training.

Since existing techniques cannot easily scale to large scale installations, not much research has been conducted in measuring the extension of emerging Internet threats. Still, measurement research, when it is conducted diligently and accurately, can have a great impact to improve security as well. The scientific community considers a method *scientific* when the phenomena that is studied can be measured, and the measure becomes the way to quantify the results of an experiment or to judge the entity of a problem. For example, in the fight against SPAM campaigns, it is only by conducting a measurement of how many of the delivered e-mails contain unwanted content, that someone can evaluate the efficiency of a SPAM solution or the evolution of the SPAM problem over a long period. Measurement research is helpful to reveal which classes of problems are increasing and on which solutions it makes sense to invest. Measurements, as well as research surveys, are useful to researchers of the same fields to build a common understanding and knowledge of the problems.

3.1 Contributions

In this thesis we propose three novel techniques to measure Internet threats on a large scale. We identified three classes of problems that concern Internet systems which have recently

experienced a surge in popularity and we tried to estimate how prevalent and relevant are these problems for Internet users. In fact, if we consider the Web, while traditional threats like XSS and SQLi have been extensively studied [83, 131, 145] and described (e.g by the OWASP Top 10 project [109]), for other emerging classes of problems, namely Clickjacking and HTTP Parameter Pollution, it is still unclear how significant they are for the security of Internet users. Although the problems we selected were previously known, we are the first to conduct a systematic study that aims at defining how prevalent are these problems over the Internet. Our goal is to understand if the types of problems we selected are known by developers, if they occur more in certain situations, and how many systems are affected by them. To answer these questions, we designed and implemented automated tools to analyze on a large scale thousands to millions of possible vulnerable targets.

The first problem we studied, Clickjacking, exploits some rendering properties of modern browsers to trick users into initiating money transfers, clicking on banner ads that are part of an advertising click fraud, posting blog or forum messages, or, in general, to perform any action that can be triggered by a mouse click. For example, different web-based malware have been reported recently in the wild to use Clickjacking to spread over social networks (e.g. Facebook or Twitter). The exploit works by presenting people with friend profiles that recommend links with attractive titles (e.g. "LOL This girl gets OWNED after a POLICE OFFICER reads her STATUS MESSAGE" or "Don't click"). Those who click on the link, follow-up into retweeting the link or adding the link to his list of "Likes".

The idea behind a Clickjacking attack is simple: A malicious page is constructed with the aim of tricking users into clicking on an element of a different page that is only barely, or not at all noticeable. Thus, the victim's click causes unintentional actions in the context of a legitimate website. Since it is the victim who actually, but unknowingly, clicks on the element of the legitimate page, the action looks "safe" from the browser's point of view; that means the same origin policy is not violated.

Testing for Clickjacking attempts is not trivial. One main challenge is to simulate the behavior of a human user that interacts with the content of the web page to test. Existing solutions such as Selenium [7] and Watir [8] simulate the mouse actions from inside the browser, sending events to the element that should be clicked. Although this approach is convenient for testing the functional requirements of web applications (such as the correctness of links and form references), it is not suitable for our purposes. The reason is that we do not know on which element the user intended to click (this, in fact, is the premise for a Clickjacking attack). Hence, we did not wish to "simulate" a user click inside the browser, but to control the mouse at the window level, and to actually move it over the interesting element, and click the left button on it. By doing this, we can also be sure that every JavaScript code in the page are executed exactly in the same way as they would if the user was controlling the mouse herself.

The second challenge is to efficiently render pages of modern applications that often rely on dynamic content (e.g. Javascript) to generate their pages. Similar to other scanners, it would have been possible to directly retrieve web pages without rendering them in a real browser. However, such techniques have the drawback that they cannot efficiently deal with dynamic content that is often found on Web 2.0 sites. By using a real browser to render the pages we visit, we are able to analyze the page as it is supposed to appear to the user after the dynamic content has been generated.

We implemented our testing and detection approach in an automated system called *ClickIDS* that we used to conduct an empirical study on over one million unique web

pages to estimate the prevalence of Clickjacking attacks on the Internet.

We then focused on another threat for the Web called HTTP Parameter Pollution that, since its first announcement in 2009 [110], had not received much attention. By proposing the first automated approach for the discovery of HTTP Parameter Pollution vulnerabilities in web applications, we looked at 5,000 popular websites and we tried to measure how many of them may be vulnerable to HPP attacks.

HPP attacks consist of injecting encoded query string delimiters into other existing parameters. If a web application does not properly sanitize the user input, a malicious user can compromise the logic of the application to perform either client-side or server-side attacks. One consequence of HPP attacks is that the attacker can potentially override existing hard-coded HTTP parameters to modify the behavior of an application, bypass input validation checkpoints, and access and possibly exploit variables that may be out of direct reach.

HPP flaws can be abused in different ways but their detection is not easy. One problem is that modern web applications are very complex, and often include dynamic content that may vary even when the page is accessed with exactly the same parameters. Publicity banners, RSS feeds, real-time statistics, gadgets, and suggestion boxes are only a few examples of the dynamic content that can be present in a page and that may change each time the page is accessed. Another challenge is that some parameters do not affect the application logic, but are used to store the URL of the web page. Hence, performing an injection in these parameters is equivalent to modifying their values to point to a different URL. Even though this technique is syntactically very similar to an HPP vulnerability, it is not a proper injection case.

In our prototype implementation called *PAPAS*, we integrated heuristics to make our tool suitable for large scale analysis and we presented the first automated approach for the detection of HPP vulnerabilities in web applications. Our approach consists of a component to inject parameters into web applications and a set of tests and heuristics to determine if the pages that are generated contain HPP vulnerabilities. In order to show the feasibility of our approach, we used PAPAS to conduct a large scale analysis of more than 5,000 popular websites.

We finally looked at emerging threats affecting the Cloud on a large scale. Cloud computing has changed the view on IT as a pre-paid asset to a pay-as-you-go service. Several companies such as Amazon Elastic Compute Cloud (EC2) [19], Rackspace [21], IBM SmartCloud [24], Joyent Smart Data Center [26] or Terremark vCloud [22] offer access to virtualized servers in their data centers on an hourly basis. Servers can be quickly launched and shut down via application programming interfaces, offering the user a greater flexibility compared to traditional server rooms. This paradigm shift is changing the existing IT infrastructures of organizations, allowing smaller companies that cannot afford a large infrastructure to create and maintain online services with ease.

Unfortunately, while the trust model between the cloud user and the cloud provider is well-defined (i.e., the user can assume that cloud providers such as Amazon and Microsoft are not malicious), the trust relationship between the provider of the virtual image and the cloud user is not as clear.

We explored the general security risks associated with the use of virtual server images from the public catalogs of cloud service providers. Over several months, we designed and ran security tests on public AMIs that aimed to identify security vulnerabilities, problems,

and risks for cloud users as well as the cloud providers. We analyzed on a large scale over five thousands public images provided by Amazon, and we identified three main threats related, respectively, to: 1) secure the image against external attacks, 2) secure the image against a malicious image provider, and 3) sanitize the image to prevent users from extracting and abusing private information left on the disk by the image provider.

In this thesis, we conceived a set of novel techniques for the measurement of Internet threats on a large scale. What these techniques have in common is that they employ sophisticated crawling and testing functionalities to process and analyze efficiently a large number of possible vulnerable targets. We used our solutions to conduct large scale studies of three classes of problems that affect important and increasingly popular Internet systems - e.g. the Web and the Cloud. We believe that our solutions will remain valuable in the near future to analyze similar classes of problems. For example, for the analysis of web vulnerabilities, we have introduced a browser-centric system to cope with the modern websites that often rely on dynamic content (e.g. Javascript) to generate their pages. By using a real browser to render the pages we visit, we are able to analyze them as they are supposed to appear to the user after the dynamic content has been generated. This technique can easily be adapted and reused in the analysis of similar classes of problems, such as attacks that exploit the new HTML5's features (e.g. drag&drop).

3.2 Summary

The contribution of this dissertation can be summarized in the following points:

- We advance the state of the art in large scale testing and measurement of Internet threats.
- We research into three novel classes of security problems that affect Internet systems which experienced a fast surge in popularity, e.g. web applications and cloud computing services.
- We introduce the first, large scale attempt to estimate the prevalence and relevance of these problems over the Internet. Our study is conducted on a large scale over thousands to millions of possible vulnerable targets.
- We describe the challenges we faced to test real applications and services on a large scale in an efficient manner.
- Finally, when we were able to obtain contact information, we informed the affected providers about the problems we discovered. Some of them acknowledged our issues and implemented the countermeasures we proposed to them.

3.3 Organization

The rest of the dissertation is organized as follows:

- In Chapter 4 we present the research works that are related to this thesis.

- In Chapter 5 we propose a novel solution for the automated and efficient detection of Clickjacking attacks. We describe *ClickIDS*, the system that we designed, implemented and deployed to analyze over a million unique web pages. We published this research in the proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (AsiaCCS 2010)

- In Chapter 6 we introduce the first automated approach for the discovery of HTTP Parameter Pollution vulnerabilities in web applications. Using our prototype implementation called *PAPAS (PArameter Pollution Analysis System)*, we conducted a large scale analysis of more than 5,000 popular websites. This research has been published, and received the Best Paper Award in the 18th Annual Network and Distributed System Security Symposium (NDSS 2011).

- Chapter 7 explores the general security risks associated with the use of virtual machine images from the public catalogs of cloud service providers (e.g. Amazon). We describe the design and implementation of an automated system called *SatanCloud* that we used to analyze the security of 5,000 public images provided by the Amazon's EC2 service. We published this research in the proceedings of the 11th edition of the Computer Security track at the 27th ACM Symposium on Applied Computing (SEC@SAC 2012).

- Finally, in Chapter 8 we conclude and discuss the limitations and possible further improvements to our contributions.

Chapitre 4

Related Work

4.1 Large-Scale Internet Measurement

A large number of papers have presented methods for measuring how the Internet has evolved over the years. Already 10 years ago, Cho and Garcia-Molina [45] have selected and analyzed over 720,000 web pages from 270 sites with the intent of understanding how often a web page changes and what its lifespan is. The experiment has been conducted using the Stanford WebBase crawler, a system designed to create and maintain large web repositories, which was initially used for the Google search engine [38]. Cho's dissertation [44] analyzes the challenges in building a web crawler that can retrieve high quality pages quickly, while maintaining the retrieved pages fresh. Fetterly et al. [54] expanded on Cho and Garcia-Molina's study, both in terms of coverage and in terms of sensitivity to change. The authors crawled about 150 million web pages once every week, over a span of 11 weeks, by recording the changing features such as the page's checksum, the number of words and the length.

Measuring the Web is important for research, but is not straightforward. One of the main difficulties one faces are the dynamic mechanisms used by modern web applications to render the content of their pages. For example, a server-side application that creates a page after the request for the page is received from the browser, or a page that includes code that executes on the browser to retrieve content from remote servers. Lawrence and Giles [91] estimated that close to 80% of the content on the Web is dynamically-generated, and that this number is continuously increasing. Recent papers [92, 43, 34, 99] have focused on proposing crawlers that are either more efficient in coverage or faster than traditional ones. For example, Raghavan et al. [114] introduced a framework called Hidden Web Exposer (HiWE) to crawl and extract information from pages that are built using dynamic technologies (what they called the Hidden Web).

Beside the Web, there are studies that have measured the Internet from a networking perspective. Paxon discussed and introduced an Internet infrastructure called NIMI (National Internet Measurement Infrastructure) that uses a collection of cooperative platforms to measure network properties (paths and clouds) by exchanging test traffic among themselves [111]. Shavitt at al. [127] proposed DIMES a distributed measurement infrastructure for the Internet that is based on the deployment of thousands of light weight measurement agents around the globe. Other research projects that have studied the Internet's growing topology from a network perspective are [90, 63, 32].

In the next section, we present the research works that have conducted measurements of Internet threats and that are related to this thesis.

4.2 Measurement of Internet Threats

One popular and well-studied Internet threat are drive-by-downloads. In a drive-by-download scenario, the attacker configures a website to serve a malicious code (e.g. Javascript) that exploits a vulnerability in the visitor's browser to download malware or any other sort of malicious software in the victim's PC. In fact, due to the complexity of modern browsers, a large amount of client-side vulnerabilities, ranging from insecure interfaces of third party extensions to buffer overflows and memory corruptions, are constantly discovered by attackers. One first example of drive-by-download attack occurred in 2004 when different critical businesses sites (e.g. banks, providers and insurance companies) were hosting a malicious code that was exploiting the vulnerable browser of their visitors to install key loggers and trojan software in the visitors computers, with the intent of capturing sensitive information such as social security numbers, credit card numbers, usernames, passwords, and encrypted financial communications [12].

Provos et al. [112] conducted a large-scale study of drive-by-download infections over the Internet with the intent of estimating the prevalence of web-based malware. Over a 10 month period, they analyzed about 66 million URLs by deploying a large number of honeypot instances with unpatched versions of Internet Explorer. The authors identified more than 3 million malicious URLs hosted on more than 180 thousand landing sites. By comparing these URLs with their Google queries, they found out that on average, 1.3% of the queries are serving a URL with drive-by-download content.

Another empirical study on drive-by-download attacks has been done by Wang et al. [141]. The authors developed an automated system called Strider HoneyMonkey Exploit Detection System, which consists of web browsers running on operating systems with different patch levels. The idea is that, by observing the successful infections, it is possible to learn the vulnerability that was exploited. Using their system, the authors discovered and collected different zero-day exploits that targets unpatched browser vulnerabilities. A zero-day can be detected if a fully patched system is infected.

The Internet can serve as container and provider for malicious software - e.g. spyware or malware - that is downloaded into unaware users' desktops. Moshchuk et al. [104] performed a large-scale, longitudinal study of the Web, sampling both executables and conventional Web pages for malicious objects. The authors, by crawling about 40 million URLs, found executable files in approximately 19% of the crawled websites and spyware-infected executables in about 4% of the sites. Their empirical study, by confirming that 1 out of 20 of the executable files they downloaded contain spyware, shows that using Internet to download unverified software exposes their users to software infections.

More recently, Chinese researchers Zhuge et al. [150] measured that out of 145,000 commonly visited Chinese websites, about 2,000 of them (1.5%) contained some kind of malicious content, e.g. malware or drive-by-download software. The researchers also examined the relationship between the time when the vulnerabilities were advertised on the underground black market and their usage on the Web.

The Web is not the only channel in which malicious software is delivered over Internet. Kalafut et al. [85] have examined the prevalence of malware in peer-to-peer networks. The authors instrumented two open source P2P clients - Limewire and OpenFT - to download about 90,000 program files over a month and to scan them for the presence of known malware. Their results show that 68% of the downloaded content in Limewire contains malware.

Two other works have measured and studied the spreading of malware within a Intranet

environment (e.g. a university campus). Goebel et al. [61] collected information about 13,4 million exploits from an eight-weeks traffic dump of 16,000 internal IPs, while Saroiu et al. [121] quantified the presence of four widespread spyware (Gator, Cydoor, SaveNow, and eZula) among the 40,000 hosts of their university.

While a large number of researchers have measured the prevalence of malicious software over the Internet, for example in vulnerable websites being exploited and converted into malicious servers serving malware or spyware, some others have studied how many web applications are affected by vulnerabilities like cross-site scripting or SQL injection.

One study [46] has been conduced by Christey et al. in 2007 on the vulnerabilities that had been publicly disclosed and indexed by the MITRE's CVE database [102]. The authors adopted a manual classification of CVE entries using the CWE classification system. The results show that the total number of publicly reported web application vulnerabilities have overtaken traditional buffer overflows, with cross-site scripting scoring the first place and SQL injection the second. Neuhaus and Zimmermann have applied topic models on the CVEs' description texts to find prevalent vulnerability types and new trends semi-automatically [106]. Their analysis is based on 39,393 unique security bulletins published in the CVE database up to 2009, inclusive.

More recently Scholte et al. [122] proposed an automated system to process CVE data from the National Vulnerability Database (NVD) [108], which includes more comprehensive information such as the name, the version, and the vendor of the affected application together with the impact and severity of the vulnerability according to the Common Vulnerability Scoring System (CVSS) standard [98]. The authors, by analyzing and focusing their study on top web vulnerabilities like cross-site scripting and SQL injection, try to understand if these attacks become more sophisticated over time, if popular applications are more vulnerable than others, and what is the average lifetime of an unpatched flaw.

The Web Application Security Statistics Project [62] has built a statistics of vulnerabilities on about 12,000 deployed web applications. The project has collected information from companies running web application security assessment activities such as penetration testing and security auditing. Unlike CVE or NVD statistics that provide valuable insight into vulnerabilities discovered in open source and commercial applications, this project focuses on custom web applications. Moreover, the vulnerabilities have been collected regardless of the methodology used to identify them. Finally, Grossman et al. published a white paper [64] that gives a statistical picture gleaned from five years of vulnerability assessment results taken from about 3,000 websites across 400 organizations under the web vulnerability management system of their company.

Vogt et al. [138] gave an indirect insight on the prevalence of cross-site scripting vulnerabilities by evaluating their tool against a million unique web pages. The authors introduced a solution to stop cross-site scripting attacks on client-side by tracking the flow of sensitive information inside the browser with a technique known as taint analysis. By crawling and visiting a million Internet sites with their implemented solution, the authors have indirectly measured how many of them could have been exploited via a persistent (stored) XSS flaw.

Yue et al. [147] crawled 6,805 unique websites from 15 categories (e.g. business, shopping, computer) and analyzed their Javascript code for insecure development practices that could lead to different kind of browser-based security attacks. Richards et al. [118] analyzed more in details the usage of the eval() function in over 10,000 websites.

These papers conducted measurements of Internet threats, for example, by studying

the number of websites that host and offer malicious software, or that are affected by well-known problems such as cross-site scripting and SQL injections. However, attackers are constantly looking for novel vulnerabilities to bypass protection mechanisms, or to deliver malicious content. Clickjacking and HTTP Parameter Pollutions are two of these emerging vulnerabilities, and for both threats, it is unclear how significant and relevant they are for the security of Internet users. Testing web applications on a large-scale is not straightforward, and new techniques are needed - e.g. modern web applications often rely on dynamic content (e.g. Javascript) to generate their pages. The next section presents existing techniques for testing web applications for vulnerabilities and describes any research related to Clickjacking and HPP.

4.3 Web Vulnerabilities

There are two main approaches [58] to test software applications for the presence of bugs and vulnerabilities: white-box testing and black-box testing. In white-box testing, the source code of an application is analyzed to find flaws. In contrast, in black-box testing, input is fed into a running application and the generated output is analyzed for unexpected behavior that may indicate errors. Black-box testing tools (e.g. [28, 40, 86, 140, 69]) are the most popular when analyzing online applications, where often the source code is not made available. These tools mimic external attacks from hackers by providing the application with malicious strings that can possibly trigger the vulnerability (e.g. XSS's <script>alert(document.cookie);</script> or SQLi's 1' or '1' = '1).

Some of these tools (e.g. [72, 28, 69]) claim to be generic enough to identify a wide range of vulnerabilities in web applications. However, two recent studies ([35, 52]) have shown that scanning solutions that claim to be generic have serious limitations, and that they are not as comprehensive in practice as they pretend to be. Doupé et al. [52] presented an evaluation of different commercial and open-source black-box solutions that they tested against a custom vulnerable web application.

SecuBat [87] by Kals et al. automatically detects SQL injection and cross-site scripting vulnerabilities on web pages. SecuBat crawls the web, and launches attacks against any HTML forms it encounters. By analyzing the server response, successful attacks can be detected. A similar approach is followed by Huang et al. [70]. In their work, the authors performed black-box testing of web applications to detect possible SQL injection and XSS flaws. As the success of such security scanners relies on the test input that is provided to the tested web applications, Wassermann et al. [143] proposed a system to automatically generate such inputs.

Concerning white-box solutions, Jovanovic et al. [84] introduced Pixy to automatically detect web vulnerabilities in PHP based applications through static analysis. Pixy is able to identify flaws that lead to SQL injection, cross-site scripting, or command injection vulnerabilities. Pixy uses taint-analysis to trace the flow of tainted data inside the application. Tainted data denotes data that originates from potentially malicious users and thus, can cause security problems at vulnerable points in the program (called sensitive sinks).

Huang et al. [71] also performed static source code analysis to automatically add runtime protection mechanisms to web applications. Similarly, Wassermann et al. [142] introduced a static string analysis-based approach to automatically detect injection vulnerabilities in web applications. Detecting SQL injection vulnerabilities by statically analyzing a web application's source code is performed by Xie et al. in [146]. Egele et al. [53]

infered the data types and possible value sets of input parameters to web applications by applying static analysis. This information can be leveraged to fine-tune application level firewalls and help protect web applications from injection attacks.

By combining dynamic data tainting with static analysis, Vogts et al. [139] created a system that effectively detects websites that perform cross-site scripting attacks. In contrast, Balzarotti et al. [31] leveraged static and dynamic analysis techniques to automatically validate sanitization in web applications.

Software engineering researchers suggested new methodologies and tools for assess the quality of web applications. Ricca and Tonella in [117] proposed an UML model that help to understand the static structure of web applications and to exploit semi-automatic white-box testing. Huang et al. [70] described and deployed in real-world applications a number of software-testing techniques which address frequent coding faults that lay to unwanted vulnerabilities. Their work has produced the vulnerability assessment tool *WAVES*.

With respect to scanning, there also exist network-level tools such as NMap [75]. Tools like NMap can determine the availability of hosts and accessible services. However, they cannot detect higher-level application vulnerabilities. Note that there also exists a large body of more general vulnerability detection and security assessment tools (e.g. Nikto [4], and Nessus [135]). Such tools typically rely on a repository of known vulnerabilities and test for the existence of these flaws. In comparison, our approach aims to discover previously unknown vulnerabilities in online web applications.

Clickjacking The first report of a possible negative impact of transparent IFRAMEs is a bug report for the Mozilla Firefox browser from 2002 [105]. In this blog post, Ruderman suggested that the use of fully transparent IFRAMEs is a bad security practice that should be avoided. IFRAMEs should be transparent only when the content, e.g. the text, is set with a transparent background. However, the term clickjacking, referring to a web attack in which a transparent IFRAME is used to trick a user into performing unintended malicious actions (e.g. clicking on banner ads that are part of an advertising click fraud), was coined by Hansen and Grossman much later in 2008 [65]. While Hansen and Grossman elaborated on the involved techniques, we are the first to conduct an empirical study on this topic.

Concerning the existing solutions for protecting Internet users from clickjacking attacks, Maone's NoScript [95] is probably the most known. NoScript is a Firefox add-on that provides protection against common security vulnerabilities such as cross-site scripting. It also features a URL access-control mechanism that filters browser-side executable contents such as Java, Adobe Flash, and Microsoft Silverlight. In October 2008, an anti-clickjacking feature was integrated into NoScript. This feature protects users against transparent IFRAME-based attacks. Starting from version 1.8.2, the protection has been extended to cover also partially obstructed and disguised elements. The implemented technique, denoted ClearClick, resembles one proposed by Zalewski [149], and is based on the analysis of the click's neighborhood region. An alert is triggered when a mouse click is detected in a region where elements from different origins overlap. In our research, we developed a new detection technique called ClickIDS that complements the ClearClick defense provided by the NoScript plug-in. We combined them in an automated, web application testing system, that we used to estimate the prevalence of clickjacking attacks on the Internet by automatically testing more than a million web pages that are likely to

contain malicious content and to be visited by Internet users.

A number of techniques to mitigate the clickjacking problem have been discussed on security-related blogs [149]. One approach proposes to extend the HTTP protocol with an optional, proprietary X-FRAME-OPTIONS header. This header, if evaluated by the browser, prevents the content to be rendered in a frame in cross-domain situations. A second approach consists to enhance the CSS or HTML languages to allow a page to display different content when loaded inside a FRAME. These strategies can be a valid solution to protect users against the clickjacking attack, but are far away from being adopted by developers: From our analysis of over more than 11,000 unique and popular URLs, only one was using the X-FRAME-OPTIONS header.

HPP HTTP Parameter Pollution is a recent web vulnerability which has been originally described in 2009 by Di Paola et al. [110]. The two researchers did an excellent work in showing how HPP flaws can be exploited to compromise the logic of a vulnerable application and to launch client-side or server-side attacks. However, they have neither introduced a methodology to detect HPP issues, nor tried to estimate how many web applications are eventually vulnerable to HPP attacks.

Web applications are commonly tested for vulnerabilities using black-box tools. To the best of our knowledge, only one of these black-box scanners, Cenzic Hailstorm [41], claims to support HPP detection. However, a study of its marketing material revealed that the tool only looks for behavioral differences when HTTP parameters are duplicated (i.e., not a sufficient test by itself to detect HPP). Unfortunately, we were not able to obtain more information about the inner-workings of the tool as Cenzic did not respond to our request for an evaluation version.

The injection technique we use is similar to other black-box approaches such as SecuBat [86] that aim to discover SQL injection, or reflected cross site scripting vulnerabilities. However, note that conceptually, detecting cross site scripting or SQL injection is different from detecting HPP. In fact, our approach required the development of a set of tests and heuristics to be able to deal with dynamic content that is often found on webpages today (content that is not an issue when testing for XSS or SQL injection). Hence, compared to existing work in literature, our approach for detecting HPP, and the prototype we present in this dissertation are unique.

With respect to white-box testing of web applications, a large number of static source code analysis tools that aim to identify vulnerabilities have been proposed. These approaches typically employ taint tracking to help discover if tainted user input reaches a critical function without being validated. We believe that static code analysis would be useful and would help developers identify HPP vulnerabilities. However, to be able to use static code analysis, it is still necessary for the developers to understand the concept of HPP.

4.4 Cloud Computing Threats

Both academic and industrial researchers have started to study the security problems related to the use of cloud services.

There are several organizations that released general security guidance on the usage of cloud computing (e.g. [16, 15, 23]). Amazon Web Services, in addition to the security advices already mentioned, released a paper describing the security processes put in place,

focusing more specifically on the management of public images [18]. Varia [137] gave an overview of the security processes in the cloud infrastructure of Amazon Web Services.

The problem statement of security on cloud computing infrastructures has been widely explored. Garfinkel and Rosenblum [57] studied the problem statement of using virtual images and especially the security problems of using third party virtual images. Glott and al. [60] presented a good overview of the problem of sharing images as well. The above papers expose a set of best practices and problem statements but did not perform any practical assessment tests.

On the other hand, some authors [133] focused on solutions that would mitigate parts of the security problems we identified. For example, solutions that tackle the problem of rootkits detection on multi-tenant clouds [47]. The authors described a secure-introspection technique at the virtualization layer in order to identify the guest OS and implement rootkit detection while running outside the guest OS. Ibrahim et al. [73] analyzed the problem of cloud virtual infrastructures and drew the requirements for virtualization-aware techniques. Another approach to address the problem consists in specifying contracts between cloud providers and virtual machine users [97]. In this case, the authors introduced extensions to the Open Virtual Machine Format (OVF) to specify the requirements of machine images in order to be safely executed by a cloud computing provider. Wei et al [144] proposed a technique to share and use third party images in a secure way. The authors described an image management system that controls the access to machines, tracks their provenance and integrity, and finally assesses their security through scanning. Thus, these papers focused on solutions and did not implement or perform a wide range of tests on an existing cloud infrastructure.

More specific to Amazon EC2, a novel approach was proposed by Bleikerts et al. [37]. The paper analyses the security of an infrastructure (a set of connected virtual machines) deployed on Amazon EC2 through graph theory techniques. The goal is to provide security resilience metrics at an infrastructure level rather than a virtual image level. Thus, this study aims at designing better cloud infrastructures by identifying problems of security groups configuration (network firewall rules).

Other works focused on the placement algorithm of Amazon EC2 instances [119], and showed how to exploit it to achieve co-residence with a targeted instance. This is the first step towards the exploitation of side channel attacks but these attacks were only outlined. Slaviero et al. [68] showed that cloud users are not careful when choosing AMIs. By publishing a public malicious AMI, they showed that this AMI was instantiated several times and statistics about the usage of the AMIs were collected. Moreover, they showed that it was possible to circumvent the payment mechanism of paid AMIs by modifying the AMI manifest file.

Finally, Bugiel et al. [39] have recently published a paper in which they describe the different risks associated with renting machine images from the Amazon's EC2 catalogue. When compared with their work, our study is more comprehensive and conducted on a larger scale. In fact, we selected and analyzed in an automated fashion over five thousands public images provided by Amazon in its four distinct data centers, while Bugiel only considered the 1,225 images that are located in the two data center of Europe and US-East. We discovered and discussed a wider number of novel security issues by testing every image for known malware samples or vulnerabilities. Finally, during our experiments, we actively collaborated with the Amazon Security Team to have the problems acknowledged and fixed. Though most of these papers highlighted trust and security problems associated to the use of third party images, to the best of our knowledge we are the first to preset a

large-scale, comprehensive study of the security and privacy of existing images.

4.5 Summary

In this chapter, we presented the existing work in measuring Internet threats on a large scale.

We started by introducing the main studies of the Internet evolution, with a main focus on papers related to the web. We then reviewed papers that conducted measurements of Internet threats. A good number of papers have studied the prevalence of websites that host attacking code (i.e., a drive-by-download) or that provide malicious software such as malware or spyware. Some authors have crawled and analyzed web applications, while others have derived this number by analyzing network traffic or peer-to-peer systems such as Limewire.

More relevant to this thesis are those studies that have tried to understand how many online applications are affected by web vulnerabilities such as cross-site scripting or SQL injection. However, to the best of our knowledge, none of these works have looked at Clickjacking or HTTP Parameter Pollution, and no empirical studies have been previously done to define the prevalence of these novel classes of problems.

We concluded the related work section with works that analyzed the security of Internet Cloud services (e.g. Infrastructure-as-a-Service providers). Cloud Computing is a novel research area and not many papers have been published in this direction. For example, together with Bugiel [39], we are the first to describe the security risks associated with renting machine images from a cloud provider's catalogue. When compared to Bugiel's paper, our study is more comprehensive and conducted on a larger-scale (i.e., by analyzing in an automated fashion over 5,000 public images provided by the Amazon's EC2 service).

Chapitre 5

Clickjacking

This chapter proposes a novel solution for the automated and efficient detection of clickjacking attacks. Clickjacking is a malicious technique of tricking users into clicking on page elements that are only barely, or not at all, visible. By stealing the victim's clicks, clickjacking can force the user to perform unintended actions that are advantageous for the attacker (e.g., a click fraud or spamming).

We describe ClickIDS, the system that we designed and implemented to conduct a measurement study of clickjacking over a million unique web page. Our large-scale experiment suggests that clickjacking has not yet been largely adopted by attackers on the Internet.

5.1 Introduction

The *same origin policy*, first introduced in Netscape Navigator 2.0 [107], is still the keystone around which the entire security of cross-domain applications is built. The main idea is to implement a set of mechanisms in the browser that enforce a strict separation between different sources. This separation is achieved by preventing the interaction between pages that are from different origins (where the origin of a page is usually defined as a combination of the domain name, the application layer protocol, and the TCP port number). The same origin policy, hence, can guarantee that cookies and JavaScript code from different websites can safely co-exist in the user's browser. Unfortunately, attackers are constantly looking for exceptions, browser bugs, or corner cases to circumvent the same origin checks with the aim of stealing or modifying sensitive user information. One of these techniques, previously known as *UI Redress* [148], has recently gained increasing attention under the new, appealing name of *clickjacking*. Since Robert Hansen and Jeremiah Grossman announced a talk on the topic at OWASP AppSec 2008 [81], there has been a flood of news, discussions, and demonstrations on clickjacking.

The idea behind a clickjacking attack is simple: A malicious page is constructed such that it tricks users into clicking on an element of a different page that is only barely, or not at all noticeable. Thus, the victim's click causes unintentional actions in the context of a legitimate website. Since it is the victim who actually, but unknowingly, clicks on the element of the legitimate page, the action looks "safe" from the browser's point of view; that is, the same origin policy is not violated. Clickjacking attacks have been reported to be usable in practice to trick users into initiating money transfers, clicking on banner ads that are part of an advertising click fraud, posting blog or forum messages, or, in general, to perform any action that can be triggered by a mouse click.

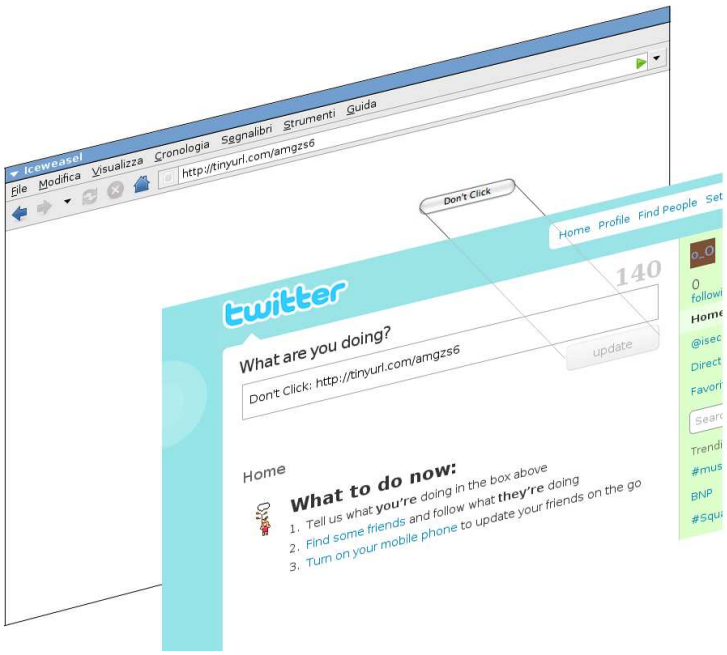


FIG. 5.1: Clickjacking attack against Twitter: The page rendering showing the two frames.

Beside several proof-of-concept clickjacking examples that have been posted on security-related blogs, it is not clear to what extent clickjacking is used by attackers in practice. To the best of our knowledge, there has only been a single, large-scale real-world clickjacking attack, where the attack was used to spread a message on the Twitter network [94]. We describe this attack in more detail in Section 5.2.

In this chapter, we present a novel approach to detect clickjacking attempts. By using a real browser, we designed and developed an automated system that is able to analyze web pages for clickjacking attacks. Our solution can be adopted by security experts to automatically test a large number of websites for clickjacking. Moreover, the clickjacking plug-in we developed can be integrated into a standard browser configuration in order to protect normal users from clickjacking during their daily Internet use.

To the best of our knowledge, we are the first to conduct a large-scale study of the clickjacking problem, and to propose a novel system for the automated testing, and detection of the attack. Our work provides a first insight into the current prevalence of clickjacking attempts on the Internet.

5.2 Clickjacking

Despite extensive discussions and reports, clickjacking still lacks a formal and precise definition. Informally, it is a technique to lure the victim into clicking on a certain element of a page, while her intention is to interact with the content of a different site. That is, even though the victim is under the impression of clicking on a seemingly harmless page, she is actually clicking on an element of the attacker's choice. The typical scenario, as described by Grossman and Hansen [66], involves two different websites: A target site T, and a malicious site M.

T is a website accessible to the victim and important for the attacker. Such sites

include, for example, online-banking portals, auction sites, and web mail services. The goal of the attacker is to lure the victim into unsuspectingly clicking on elements of the target page T.

M, on the other hand, is under control of the attacker. Commonly, this page is created in a way so that a transparent IFRAME containing T overlays the content of M. Since the victim is not aware of the invisible IFRAME, by correctly aligning T over M, an attacker can lure the victim into clicking elements in T, while she is under the impression of clicking on M. A successful clickjacking attack, for example, might result in the victim deleting all messages from her web mail inbox, or generating artificial clicks on advertisement banners.

Figure 5.1 shows a real-world clickjacking attack that has been used to propagate a message among Twitter users [94]. In this attack, the malicious page embeds `Twitter.com` on a transparent IFRAME. The status-message field is initialized with the URL of the malicious page itself. To provoke the click, which is necessary to publish the entry, the malicious page displays a button labeled "*Don't Click*." This button is aligned with the invisible "*Update*" button of Twitter. Once the user performs the click, the status message (i.e., a link to the malicious page itself) is posted to her Twitter profile. The attacking HTML code is the following:

```
<IFRAME style={
              width: 550px; height: 228px;
              top: -170px; left: -400px;
              position: absolute; z-index: 2;
              opacity: 0; filter: alpha(opacity=0);
              }
        scrolling="no"
        src="http://twitter.com/home?status=
           Don't Click: http://tinyurl.com/amgzs6">
</IFRAME>

<BUTTON style={
              width: 120px; top: 10px; left: 10px;
              position: absolute; z-index: 1;
              }>
        Don't Click
</BUTTON>
```

While clickjacking attacks can be performed in plain HTML, the use of JavaScript can be used to create more sophisticated attacks. For instance, JavaScript allows the attacker to dynamically align the framed content with the user's mouse cursor, thus making it possible to perform attacks that require multiple clicks.

Note that manipulating the frame's opacity level (e.g., making it transparent) is not the only way to mount a clickjacking attack. A click can also be "stolen" by covering the frame containing the victim page with opaque elements, and then leaving a small hole aligned with the target element on the underlying page. Another possible approach consists of resizing and/or moving the IFRAME in front of the mouse just before the user performs a click.

Unlike other common web vulnerabilities such as cross-site scripting and SQL injection, clickjacking is not a consequence of a bug in a web application (e.g., a failure to properly sanitize the user input). In contrast, it is a consequence of a misuse of some HTML/CSS features (e.g., the ability to create transparent IFRAMEs), combined with the way in which the browser allows the user to interact with invisible, or barely visible, elements.

A number of techniques to mitigate the clickjacking problem have been discussed on

security-related blogs [149]. One approach proposes to extend the HTTP protocol with an optional, proprietary X-FRAME-OPTIONS header. This header, if evaluated by the browser, prevents the content to be rendered in a frame in cross-domain situations. A similar approach was proposed to enhance the CSS or HTML languages to allow a page to display different content when loaded inside a frame.

Some of the mentioned defenses are already implemented by browser vendors. For example, Microsoft's Internet Explorer 8 honors the X-FRAME-OPTIONS HTTP header and replaces a page whose header is set to `deny` with a warning message [100]. Additionally, the NoScript plug-in for Firefox [95] also evaluates this header and behaves accordingly [96].

In the meanwhile, web developers who do not wish their content to be displayed as frames in other pages have been adopting so-called *frame-busting* techniques. An example of frame-busting is the following JavaScript snippet:

```
<script type="text/javascript">
 if ( top.location.hostname != self.location.hostname )
      top.location.replace(self.location.href);
</script>
```

The code compares the origin of the content with the currently displayed resource in the browser and, upon a mismatch, it redirects the browser, thus, "busting" the frame.

Interestingly, an attacker who specifies the IFRAME's attribute `security="restricted"` [101] can force the Internet Explorer to treat the frame's content in the security context of *restricted sites*, where, by default, active scripting is turned off. Therefore, if the embedded page does not make use of JavaScript beyond frame-busting, this protection can be thwarted by an attacker.

An alternative solution, not relying on JavaScript, requires the user to re-authenticate (e.g., by re-typing the password or by solving a CAPTCHA) in order to perform any sensitive actions. However, frequent re-authentications degrade the user experience, and thus, cannot be used extensively.

Finally, a possible way to mitigate the problem consists of detecting and stopping clickjacking attempts in the browser. The *ClearClick* extension, recently introduced into the NoScript plug-in, offers some degree of protection. To this end, ClearClick attempts to detect if a mouse click event reaches an invisible, or partially obstructed element. The click event is put on hold, and the user is informed about the true origin of the clicked element. Only if the user agrees, the event propagation continues as usual. Note that NoScript's ClearClick exhibited a large number of false positives in our experiments (i.e., see Section 5.4).

5.3 Detection Approach

In this section, we present our approach to simulate user clicks on all the elements of the page under analysis, and to detect the consequences of these clicks in terms of clickjacking attacks. Our technique relies on a real browser to load and render a web page. When the page has been rendered, we extract the coordinates of all the clickable elements. In addition, we programmatically control the mouse and the keyboard to properly scroll the web page and click on each of those elements.

Figure 5.2 shows the architecture of our system, which consists of two main parts: A

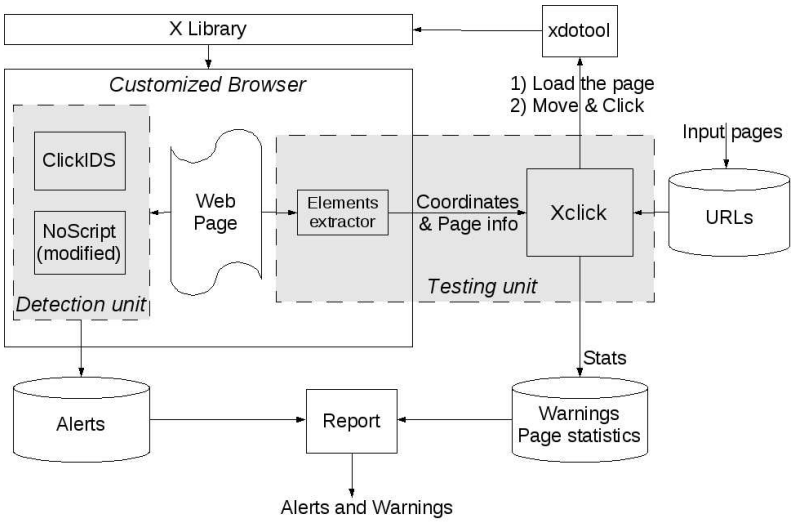


FIG. 5.2: System architecture

testing unit is in charge of performing the clicks, and a detection unit is responsible for identifying possible clickjacking attempts on the web page under analysis.

The detection unit combines two browser plug-ins that operate in parallel to analyze the automated clicks. The first plug-in consists of code that we developed in order to detect overlapping clickable elements. To complement this solution, we also adopted the NoScript tool, that has recently introduced an anti-clickjacking feature. Our experimental results (see Section 5.4) show that the combination of the two different detection techniques greatly reduces the number of false positives.

The testing unit contains a plug-in that extracts the coordinates of the clickable elements rendered on the page, and a browser-independent component that moves the mouse to the coordinates, and simulates the user's clicks. In addition, the testing unit is responsible for navigating the browser by typing into the address bar the address of the web page to visit.

In the following, we explain the two units in more detail.

5.3.1 Detection unit

This unit is responsible for detecting and logging any clickjacking attacks that are contained in the web page under analysis.

The detection is handled by two browser plug-ins. The first component is a solution that we developed to detect when multiple clickable elements co-exist and overlay in the region of the page where the user has clicked. We call our detection solution *ClickIDS*. The second plug-in is the modified version of the NoScript open-source tool that saves the generated alerts into a database instead of displaying pop-ups to the user. In the following, we describe ClickIDS and NoScript in more detail.

ClickIDS ClickIDS is the browser plug-in that we implemented. It intercepts the mouse click events, checks the interactions with the elements of a web page, and detects clickjacking attacks.

The basic idea behind ClickIDS is simple. A suspicious behavior is reported when two

or more clickable elements of different pages overlap at the coordinates of the mouse click. As clickable elements, we consider links (or, more precisely, the area enclosed between HTML `<A>` tags), buttons, and form inputs fields such as checkboxes, radio buttons, menu, and text fields. In addition, we also take into account Adobe Flash content, embedded in HTML with `<EMBED>` tags and associated with the application-type *x-shockwave-flash*.

We motivate the consideration of Flash content in two ways. First, when clickjacking was first reported in October 2008, it gained interest fast mainly because a clickjacking exploitation technique against the Adobe Flash Player Setting Manager would have permitted to modify the web-cam and microphone security settings [136]. Basically this exploit allowed an attacker to remotely turn the user's computer into an eavesdropping device. Second, for some advanced attacks to be successful, an attacker would need to steal multiple user-clicks, and therefore, would prefer to overlay the clickjacked site with flash content (e.g., a game) that persuades the user to perform several clicks.

When our plug-in is loaded, it registers to the document event *load*. Each time a new page is loaded, the page-handler routine is executed. This routine registers the loaded page and attaches a second click-handler to it. At this point, every click of the user in the context of the web page is intercepted, and handled by the click-handler routine.

If the clicked element is clickable (according to our previous definition), we register the current mouse coordinates. Then, we scan the main page and the contained FRAMEs and IFRAMEs to check if they contain clickable elements at the same position. If there exists at least one element that overlays the clicked one, we generate an alert. ClickIDS is more precise in identifying attacks based on overlapping elements. However, unlike NoScript, it is not able to detect attacks based on partially obstructed pages. Nevertheless, the combination of the two different techniques can effectively reduce the number of false positives generated by the tools individually.

Note that we also developed a third component that we call *Stopper*, which drops mouse events after all the registered listeners have been successfully executed. This prevents the browser from actually opening a new page, submitting a form, or downloading a file in response to the mouse clicks.

NoScript NoScript is a Firefox add-on that provides protection against common security vulnerabilities such as cross-site scripting. It also features a URL access-control mechanism that filters browser-side executable contents such as Java, Adobe Flash, and Microsoft Silverlight.

In October 2008, an anti-clickjacking feature was integrated into NoScript. This feature protects users against transparent IFRAME-based attacks. Starting from version 1.8.2, the protection has been extended to cover also partially obstructed and disguised elements. The implemented technique, denoted ClearClick, resembles one proposed by Zalewski [149], and is based on the analysis of the click's neighborhood region. An alert is triggered when a mouse click is detected in a region where elements from different origins overlap.

For our prototype, we modified NoScript version 1.9.0.5 and we replace the visual alerts that are normally generated for the user with a logging capability toward an external SQL database. Our customized version produces an entry for each clickjacking attempt, containing a reference to the website that has generated the alert, the URL of the (I)FRAME that has been clickjacked, and the element of the page that has been clicked (tag name, type, href, and coordinates).

5.3.2 Testing unit

The testing unit simulates the behavior of a human user that interacts with the content of a web page. It is responsible for instructing the browser to visit a certain URL, and then to iteratively click on the clickable elements contained on the page.

We designed our testing unit to overcome the limitations of existing systems for web application testing. Solutions such as Selenium [7] and Watir [8] simulate the mouse actions from inside the browser, sending events to the element that should be clicked. Although this approach is convenient for testing the functional requirements of web applications (such as the correctness of links and form references), it is not suitable for our purposes. The reason is that we do not know on which element the user intended to click on (this is, in fact, the premise for a clickjacking attack). Hence, we did not wish to "simulate" a user click inside the browser, but to control the mouse at the window level, and to actually move it over the interesting element, and click the left button on it. By doing this, we can also be sure that every JavaScript code in the page (such as the ones registered to `OnMouseOver` or `OnMouseUp` events) are executed exactly in the same way as they would if the user was controlling the mouse herself.

Our tool utilizes *xdotool* [10], a wrapper around the X11 testing library, to move the mouse on the screen and to generate keyboard and mouse events. The testing unit can place the mouse cursor at the screen coordinates where the web page's clickable elements are rendered. Since the clicks are generated from the graphical interface itself, from the browser's point of view, they are identical to those of a real user.

The main component of the testing unit is the *Xclick* script. It receives the list of URLs to visit, and it feeds them, one by one, to the browser. Once the page is successfully loaded, the script positions the mouse over each element, and clicks on them. If the element coordinates are outside the browser window, Xclick properly scrolls down the page to show the required content. In addition, Xclick properly manages special elements such as form drop down boxes which can be rolled up pressing the escape button. For large elements (e.g., images and flash contents), it is more difficult to predict the exact position where the clickjacking may occur. Therefore, Xclick performs multiple clicks at fixed intervals to cover the entire element area, in such a way to raise any possible clickjacking attacks. Finally, to improve the reliability of the system, Xclick is able to detect and close any windows, popups, or browser tabs that are opened by the web page as a consequence of the mouse clicks. A pseudocode of the script is here detailed:

```
start browser
for url in input:
   check the browser functionalities, else:
      restart it
   feed the browser with the url and instruct it
                                to load the page
   wait for the page to be loaded
   if a timeout occurs:
      continue
   check the elements extractor's logfile, else:
      continue
   parse the logfile for the list_of_elements and
                              the page statistics
   record the page statistics in the database

   for element in list_of_elements:
      if element > 50x50px:
         crop it (multi click)
      if element.coordinates are in the next page:
```

```
      scroll the browser page
   check the element.coordinates validity else:
      continue
   move the mouse on the element.coordinates
   click
   if element.type == select:
      press 'esc' to close the menu
```

The coordinates of the web page's clickable elements are received from the *element extractor*, a custom extension that we installed in the browser. This component is registered to the page-open event such that each time a page is loaded, a callback function is called to parse the page's DOM, and to extract all the information about the clickable elements. The plug-in also extracts information concerning all the FRAMEs and IFRAMEs included in the visited page, including their URL and opacity values. The opacity is a CSS3 property that specifies the transparency of an HTML element to a value varying from 0.0 (fully transparent) to 1.0 (completely opaque).

5.3.3 Limitations

The main limitation of our current implementation to detect clickjacking attempts is that the testing unit interacts only with the clickable elements of the page. In general, this is not a requirement for mounting a clickjacking attack because, at least in theory, it is possible for an attacker to build a page in which a transparent IFRAME containing the target site is placed on top of an area containing normal text.

In order to cope with this additional set of attacks, we combine the alerts produced by our plug-ins with the warnings generated by the Xclick tool for the web pages that contain cross-domain transparent IFRAMEs. In particular, our approach generates a final report containing both the *alert messages* for the pages where a clickjacking attempt is detected, and the *warning messages* that are raised when no attacks are detected, but the page contains transparent IFRAMEs that partially overlap the rest of the page content. As explained in the Section 5.4, by analyzing the warning messages, it is possible to detect the clickjacking attacks that do not make use of clickable elements.

5.4 Evaluation

To test the effectiveness of our prototype tool in detecting clickjacking attacks, we first created five different test pages, based on the examples published on the Internet, that contained clickjacking attacks,. In all cases, the system correctly raised an alert message to report the attack.

Having initially validated our approach on these test pages, we set out to test the effectiveness of our system in identifying real-world websites containing similar, previously-unknown clickjacking attacks. We combined different sources to obtain an initial list of URLs that is representative of what an average user may encounter in her everyday web browsing experience. More precisely, we included the top 1000 most popular websites published by Alexa [1], over 20,000 profiles of the MySpace [3] social network, and the results of ad-hoc queries on popular search engines. In particular, we queried Google and Yahoo with various combinations of terms such as "porn," "free download," "warez," "online game," "ringtones," and "torrents." We ran each query in different languages including English, German, French, Italian, and Turkish.

	Value	Rate
Visited Pages	1,065,482	100%
Unreachable or Empty	86,799	8.15%
Valid Pages	978,683	91.85%
With IFRAMEs	368,963	37.70%
With FRAMES	32,296	3.30%
Transparent (I)FRAMEs	1,557	0.16%
Clickable Elements	143,701,194	146.83 el./page
Speed Performance	71 days	15,006 pages/day

Tab. 5.1: Statistics on the visited pages

To increase the chances of finding attacks, we also included sources that were more likely to contain malicious content. For this purpose, we included domains from `malwaredomains.com` [2], lists of phishing URLs published by PhishTank [5], and domains that were queried by malware samples analyzed by the Anubis [77] online malware analysis tool.

Combining all these sources, we generated an initial seed list of around 70,000 URLs. Our crawler then visited these URLs, and continued to crawl through the links embedded in the pages. Overall, we visited 1,065,482 pages on 830,000 unique domains.

For our crawling experiments, we installed our tool on ten virtual machines executed in parallel on VMWare Server 3. Since a clickjacking attack is likely to exploit a transparent IFRAME, to speedup the analysis, we decided to dedicate half of the machines to click only on pages containing transparent IFRAMEs. Each machine was running Debian Linux Squeeze and Mozilla Firefox 3, equipped with our detection and testing plug-ins. The browser was customized for being suitable for running in the background, and enabling automated access to its interface. We also disabled any user interfaces that required user interaction, blocked pop-ups and video content, and disabled document caching.

5.5 Results

We ran our experiments for about two months, visiting a total of 1,065,482 unique web pages. We analyzed those pages in "online mode" and we performed an average of 15,000 pages per day. Around 7% of the pages did not contain any clickable element – usually a sign that a page has been taken down, or it is still under construction. The remaining pages contained a total of 143.7 million clickable elements (i.e., an average of 146.8 elements per page).

37.3% of the visited pages contained at least one IFRAME, while only 3.3% of the pages included a FRAME. However, only 930 pages contained completely transparent IFRAMEs, and 627 pages contained IFRAMEs that were partially transparent. This suggests that while IFRAMEs are commonly used in a large fraction of Internet sites, the use of transparency is still quite rare, accounting for only 0.16% of the visited pages. Table 5.1 summarizes these statistics.

Table 5.2 shows the number of pages on which our tool generated an alert. The results indicate that the two plug-ins raised a total of 672 (137 for ClickIDS and 535 for NoScript) alerts. That is, on average, one alert was raised every 1,470 pages. This value drops down to a mere 6 alerts (one every 163,000 pages) if we only consider the cases where both plug-ins reported a clickjacking attack. Note that NoScript was responsible for most of

	Total	True Positives	Borderlines	False Positives
ClickIDS	137	2	5	130
NoScript	535	2	31	502
Both	6	2	0	4

Tab. 5.2: Results

the alerts, and, interestingly, 97% of these alerts were raised on websites containing no transparent elements at all.

To better understand which alerts corresponded to real attacks, and which ones were false positives, we manually analyzed all alerts by visiting the corresponding web pages. The results of our analysis are reported in the last three columns of Table 5.2.

Around 5% of the alerts raised during our experiments involved a frame pointing to the same domain of the main page. Since it is very unlikely that websites would try to trick the user into clicking on a hidden element of the site itself, we marked all these messages as being false positives. However, we decided to manually visit some of these pages to have an insight into what kind of conditions tend to cause a false positive in the two plug-ins.

We then carefully analyzed the pages containing cross-domain frames. In this set, we identified a number of interesting cases that, even though not corresponding to real attacks, matched our definition of clickjacking. We decided to divide these cases in two categories: The true positives contain real clickjacking attempts, while the borderline cases contain pages that were difficult to classify as being clickjacking.

5.5.1 False positives

Most of the false alarms were generated by pop-ups that dynamically appear in response to particular events, or by banners that are placed on top of a scrollable page. In both cases, the content of the advertisement was visible to the user, but it confuses both NoScript (because the area around the mouse click is not the one that NoScript is expecting), and ClickIDS (because the banner can contain clickable elements that overlap other clickable elements on the main page). For similar reasons, the use of dynamic drop down menus can sometimes confuse both plug-ins.

NoScript also reported an alert when a page contained a transparent IFRAME positioned completely outside of the page margins. A manual examination of some of these cases revealed that they corresponded, most of the time, to compromised websites where an attacker included a hidden IFRAME pointing to a malicious page that attempts to infect the user's computer with malware. Even though these were obvious attacks, no attempts were done to intercept, or steal the user's clicks. Another common scenario that induced NoScript to generate false alarms are sites that contain IFRAMEs overlapping the page content in proximity, but not on top of, a clicked element. While ClickIDS did not report these pages, it raised several false alarms due to harmless overlapping of clickable elements even though the page and IFRAME contents were perfectly visible to the user.

Nevertheless, note that by combining together the two techniques (i.e., ClickIDS and NoScript), only four false positive messages were generated.

5.5.2 True positive and borderline cases

In our experiments, we were able to identify two real-world clickjacking attacks. The first one used the transparent IFRAME technique to trick the user into clicking on a concealed advertisement banner in order to generate revenue for the attacker (i.e., click fraud). Both plug-ins raised an alert for this attack. The second attack contained an instance of the Twitter attack we already discussed in Section 5.2. We were able to detect this second case by analyzing the warnings generated by our system for the web pages that contain cross-domain transparent IFRAMEs. In fact, by the time we visited the page, Twitter had already implemented an anti-clickjacking defense (i.e., A javascript frame-busting code now substitutes the framed page with empty content).

Even though the pages containing these clickjacking attacks turned out to be examples posted on security-related websites, they were true positives and we detected them automatically. Hence, our system was able to detect these pages by automated analysis of real-world Internet pages.

Moreover, we also found a number of interesting cases that are difficult to accurately classify as either being real attacks, or false positives.

A first example of these borderline cases occurred when an IFRAME was encapsulated in a link tag. In this case, a cross-domain IFRAME was included in a page as link content (i.e., between `<A>` tags). We found that on certain browsers, such as the Internet Explorer, the user can interact with the framed page normally, but when she clicks somewhere on the content that is not a clickable element, the click is caught by the encapsulating link. The result is a kind of "reversed" clickjacking in the sense that the user believes that she is clicking on the framed page, but is instead clicking on a link in the main page. Even though it matches our attack definition, this setup cannot be used to deceive the user into interacting with a different web page. It is unclear to us why the developers of the site chose to use this technique, but we believe that it might have a usability purpose – no matter where the user would click on the page, she would be directed to a single URL chosen by the developer.

Another interesting case that we observed in our experiments occurs when the page to be included into an IFRAME is not available anymore, or has been heavily modified since the page was created. If the IFRAME is set with the CSS attributes `allowtransparency:true` and `background-color:`
`transparent`, the content of the IFRAME is visible to the user, but the area that does not contain anything (e.g., the page background) is not. The obvious intention of the page authors was to display some content from another page (e.g., a small horizontal bar containing some news messages), but since the destination page was not found, and therefore returned a mostly empty page, the IFRAME was rendered as a transparent bar. If the area overlaps with clickable elements, the user could end up clicking on the transparent empty layer (containing a page from a different domain) instead of the main page elements.

5.5.3 False negatives

In order to estimate the false negative rate of our tool, we analyzed all pages for which warning messages were raised (i.e., the pages containing cross-domain transparent IFRAMEs, but in which no attack was reported by our plug-ins). Most of the 140 pages for which our tool raised a warning were pages that included the Blogger [14] navigation bar on the top of the page. This bar is implemented as a transparent IFRAME that is automatically set to be opaque when the mouse is moved to the top of the page. In this case, the

transparency is used as a means to easily control the appearance and disappearance of the navigation bar.

5.6 Pages implementing protection techniques

In our study, we conducted the following experiment to assess the prevalence of web sites that implement the so-called frame-busting technique (see Section 5.2).

First, we prepared a web page that accepts a single parameter denoting a URL that should be embedded in an IFRAME. Once the page and all contents (i.e., the IFRAME) finished loading and rendering, we verified that the IFRAME was still present. Pages that perform frame-busting would substitute the whole content in the browser window, thus removing the IFRAME. To automate this experiment, we implemented a Firefox extension that takes a list of URLs to be visited. Once a page is loaded, the extension waits for a few seconds and then verifies the presence of the IFRAME. If the IFRAME is not part of the document's DOM-tree anymore, we conclude that the embedded page performed frame-busting.

Simultaneously, we analyzed the HTTP headers of the visited websites. The optional X-FRAME-OPTIONS header is intended to control whether a resource can be embedded in a frame or not. While it is known that Internet Explorer 8 and the NoScript plug-in for Firefox honor this header, we also wanted to find out how common the use of this header is among the sites on the Internet.

We constructed a list of popular URLs to visit by downloading the top 200 entries of each of the 17 Alexa categories [30]. Furthermore, we added the top 10,000 pages from the Alexa rankings to the list. Due to many pages that are present in both lists, we visited a total of 11,005 unique URLs. 1,967 pages did not finish rendering within a 30 seconds timeout, and were, thus, not evaluated.

Our experiment revealed that out of the remaining 9,038 pages, 352 websites (3,8%) already implement frame-busting techniques to prevent being loaded in a frame. Furthermore, only one of the visited pages[1] was using the X-FRAME-OPTIONS header.

5.7 Summary

Clickjacking is a recent web attacks that have been widely discussed on the Web. However, it is unclear to what extent clickjacking is being used by attackers in the wild, and how significant the attack is for the security of Internet users.

In this Chapter, we presented our system that is able to automatically detect clickjacking attempts on web pages. We validated our tool and we conducted empirical experiments to estimate the prevalence of such attacks on the Internet by automatically testing more than a million web pages that are likely to contain malicious content and to be visited by Internet users. By distributing the analysis on multiple virtual machines we were able to scan up to 15,000 web pages per day. Furthermore, we developed a new detection technique, called *ClickIDS*, that complements the ClearClick defense provided by the NoScript plug-in [95]. We integrated all components into an automated, web application testing system.

Of the web pages we visited, we could confirm two proof-of-concept instances of clickjacking attacks used for click fraud and message spamming. Even though the pages

[1] http://flashgot.net/

containing these clickjacking attacks have been posted as examples on security-related websites, we found them automatically. Furthermore, in our analysis, we also detected several other interesting cases that we call "borderline attacks". Such attacks are difficult to accurately classify as either being real attacks, or false positives.

Our findings suggested that clickjacking is not the preferred attack vector adopted by attackers on the Internet. In fact, after we had finished our study, Jeremiah Grossman posted in his blog that he only expects clickjacking to become a real problem in 5 to 6 years from now [82].

Chapitre 6

HTTP Parameter Pollution

This chapter presents the first automated approach for the discovery of HTTP Parameter Pollution vulnerabilities. HPP bugs are the result of a wrong sanitization of the user inputs for parameter delimiters, and they can be potentially exploited to compromise the logic of the application and to perform either client-side or server-side attacks.

Using our prototype implementation called PAPAS, we conducted a large-scale study on more than 5,000 popular websites. Our empirical results show that about a third of the websites we tested contain vulnerable parameters, and that 46.8% of the vulnerabilities we discovered can be exploited. The fact we were able to find vulnerabilities in many high-profile, well-known websites suggests that many developers are not aware of the HPP problem.

6.1 Problem Statement

HTTP Parameter Pollution attacks (HPP) have only recently been presented and discussed [110], and have not received much attention so far. An HPP vulnerability allows an attacker to inject a parameter inside the URLs generated by a web application. The consequences of the attack depend on the application's logic, and may vary from a simple annoyance to a complete corruption of the application's behavior. Because this class of web vulnerability is not widely known and well-understood yet, in this section, we first explain and discuss the problem.

Even though injecting a new parameter can sometimes be enough to exploit an application, the attacker is usually more interested in *overriding* the value of an already existing parameter. This can be achieved by "masking" the old parameter by introducing a new one with the same name. For this to be possible, it is necessary for the web application to "misbehave" in the presence of duplicated parameters, a problem that is often erroneously confused with the HPP vulnerability itself. However, since parameter pollution attacks often rely on duplicated parameters in practice, we decided to study the parameter duplication behavior of applications, and measure it in our experiments.

6.1.1 Parameter Precedence in Web Applications

During the interaction with a web application, the client often needs to provide input to the program that generates the requested web page (e.g., a PHP or a Perl script). The HTTP protocol [55] allows the user's browser to transfer information inside the URI itself (i.e., GET parameters), in the HTTP headers (e.g., in the Cookie field), or inside the

Technology/Server	Tested Method	Parameter Precedence
ASP/IIS	`Request.QueryString("par")`	All (comma-delimited string)
PHP/Apache	`$_GET["par"]`	Last
JSP/Tomcat	`Request.getParameter("par")`	First
Perl(CGI)/Apache	`Param("par")`	First
Python/Apache	`getvalue("par")`	All (List)

Tab. 6.1: Parameter precedence in the presence of multiple parameters with the same name

request body (i.e., POST parameters). The adopted technique depends on the application and on the type and amount of data that has to be transferred.

For the sake of simplicity, in the following, we focus on GET parameters. However, note that HPP attacks can be launched against any other input vector that may contain parameters controlled by the user.

RFC 3986 [36] specifies that the query component (or query string) of a URI is the part between the "`?`" character and the end of the URI (or the character "`#`"). The query string is passed unmodified to the application, and consists of one or more `field=value` pairs, separated by either an ampersand or a semicolon character.

For example, the URI `http://host/path/somepage.pl?name=john&age=32` invokes the `verify.pl` script, passing the values `john` for the `name` parameter and the value 32 for the `age` parameter. To avoid conflicts, any special characters (such as the question mark) inside a parameter value must be encoded in its `%FF` hexadecimal form.

This standard technique for passing parameters is straightforward and is generally well-understood by web developers. However, the way in which the query string is processed to extract the single values depends on the application, the technology, and the development language that is used.

For example, consider a web page that contains a check-box that allows the user to select one or more options in a form. In a typical implementation, all the check-box items share the same name, and, therefore, the browser will send a separate homonym parameter for each item selected by the user. To support this functionality, most of the programming languages used to develop web applications provide methods for retrieving the complete list of values associated with a certain parameter. For example, the JSP `getParameterValues` method groups all the values together, and returns them as a list of strings. For the languages that do not support this functionality, the developer has to manually parse the query string to extract each single value.

However, the problem arises when the developer expects to receive a single item and, therefore, invokes methods (such as `getParameter` in JSP) that only return a single value. In this case, if more than one parameter with the same name is present in the query string, the one that is returned can either be the first, the last, or a combination of all the values. Since there is no standard behavior in this situation, the exact result depends on the combination of the programming language that is used, and the web server that is being deployed. Table 6.1 shows examples of the parameter precedence adopted by different web technologies.

Note that the fact that only one value is returned is not a vulnerability per se. However, if the developer is not aware of the problem, the presence of duplicated parameters can produce an anomalous behavior in the application that can be potentially exploited by an

attacker in combination with other attacks. In fact, as we explain in the next section, this is often used in conjunction with HPP vulnerabilities to override hard-coded parameter values in the application's links.

6.1.2 Parameter Pollution

An HTTP Parameter Pollution (HPP) attack occurs when a malicious parameter P_{inj}, preceded by an encoded query string delimiter, is injected into an existing parameter P_{host}. If P_{host} is not properly sanitized by the application and its value is later decoded and used to generate a URL A, the attacker is able to add one or more new parameters to A.

The typical client-side scenario consists of persuading a victim to visit a malicious URL that exploits the HPP vulnerability. For example, consider a web application that allows users to cast their vote on a number of different elections. The application, written in JSP, receives a single parameter, called `poll_id`, that uniquely identifies the election the user is participating in. Based on the value of the parameter, the application generates a page that includes one link for each candidate. For example, the following snippet shows an election page with two candidates where the user could cast her vote by clicking on the desired link:

```
Url: http://host/election.jsp?poll_id=4568

Link1: <a href="vote.jsp?poll_id=4568&candidate=white">
        Vote for Mr. White</a>
Link2: <a href="vote.jsp?poll_id=4568&candidate=green">
        Vote for Mrs. Green</a>
```

Suppose that Mallory, a Mrs. Green supporter, is interested in subverting the result of the online election. By analyzing the webpage, he realizes that the application does not properly sanitize the `poll_id` parameter. Hence, Mallory can use the HPP vulnerability to inject another parameter of his choice. He then creates and sends to Alice the following malicious Url:

```
http://host/election.jsp?poll_id=4568%26candidate%3Dgreen
```

Note how Mallory "polluted" the `poll_id` parameter by injecting into it the `candidate=green` pair. By clicking on the link, Alice is redirected to the original election website where she can cast her vote for the election. However, since the `poll_id` parameter is URL-decoded and used by the application to construct the links, when Alice visits the page, the malicious `candidate` value is injected into the URLs[1]:

```
http://host/election.jsp?poll_id=4568%26candidate%3Dgreen

Link 1: <a href=vote.jsp?poll_id=4568&candidate=green
        &candidate=white>Vote for Mr. White</a>
Link 2: <a href=vote.jsp?poll_id=4568&candidate=green
        &candidate=green>Vote for Mrs. Green</a>
```

No matter which link Alice clicks on, the application (in this case the `vote.jsp` script) will receive two `candidate` parameters. Furthermore, the first parameter will *always* be

[1] URLs in the page snippets have the injected string emphasized by using a red, underlining font.

set to `green`.

In the scenario we discussed, it is likely that the developer of the voting application expected to receive only one candidate name, and, therefore, relied on the provided basic Java functionality to retrieve a single parameter. As a consequence, as shown in Table 6.1, only the first value (i.e., `green`) is returned to the program, and the second value (i.e., the one carrying the Alice's actual vote) is discarded.

In summary, in the example we presented, since the voting application is vulnerable to HPP, it is possible for an attacker to forge a malicious link that, once visited, tampers with the content of the page, and returns only links that force a vote for Mrs. Green.

Cross-Channel Pollution HPP attacks can also be used to override parameters between different input channels. A good security practice when developing a web application is to accept parameters only from the input channel (e.g., GET, POST, or Cookies) where they are supposed to be supplied. That is, an application that receives data from a POST request should not accept the same parameters if they are provided inside the URL. In fact, if this safety rule is ignored, an attacker could exploit an HPP flaw to inject arbitrary parameter-value pairs into a channel A to override the legitimate parameters that are normally provided in another channel B. Obviously, for this to be possible, a necessary condition is that the web technology gives precedence to A with respect to B.

For example, in the next snippet, the attacker, by injecting the parameter `id=6` in the vulnerable parameter, force the client to build a GET request that can (possibly) override the value of the POSTed `id=1`.

```
Url: foo?vulnerable-parameter=foo%26id%3D6

Form:
<form action=buy?vulnerable-parameter=foo&id=6>
 <input type="text" name="id" />
 <input type="submit" value="Submit" />
</form>

Request:
POST  /buy?vulnerable_parameter=foo&id=6
Host: site.com

id=1
```

HPP to bypass CSRF tokens One interesting use of HPP attacks is to bypass the protection mechanism used to prevent cross-site request forgery. A cross-site request forgery (CRSF) is a confused deputy type of attack [67] that works by including a malicious link in a page (usually in an image tag) that points to a website in which the victim is supposed to be authenticated. The attacker places parameters into the link that are required to initiate an unauthorized action. When the victim visits the attack page, the target application receives the malicious request. Since the request comes from a legitimate user and includes the cookie associated with a valid session, the request is likely to be processed.

A common technique to protect web applications against CSRF attacks consists of using a secret request token (e.g., see [78, 88]). A unique token is generated by the application and inserted in all the sensitive links URLs. When the application receives a request, it verifies that it contains the valid token before authorizing the action. Hence,

since the attacker cannot predict the value of the token, she cannot forge the malicious URL to initiate the action.

A parameter pollution vulnerability can be used to inject parameters inside the existing links generated by the application (that, therefore, include a valid secret token). With these injected parameters, it may be possible for the attacker to initiate a malicious action and bypass CSRF protection.

A CSRF bypassing attack using HPP was demonstrated in 2009 against Yahoo Mail [51]. The parameter injection permitted to bypass the token protections adopted by Yahoo to protect sensitive operations, allowing the attacker to delete all the mails of a user.

The following example demonstrates a simplified version of the Yahoo attack:

```
Url:
showFolder?fid=Inbox&order=down&tt=24&pSize=25&startMid=0
%2526cmd=fmgt.emptytrash%26DEL=1%26DelFID=Inbox%26
cmd=fmgt.delete

Link:
showMessage?sort=date&order=down&startMid=0
%26cmd%3Dfmgt.emptytrash&DEL=1&DelFID=Inbox&
cmd=fmgt.delete&.rand=1076957714
```

In the example, the link to display the mail message is protected by a secret token that is stored in the `.rand` parameter. This token prevents an attacker from including the link inside another page to launch a CSRF attack. However, by exploiting an HPP vulnerability, the attacker can still inject the malicious parameters (i.e., deleting all the mails of a user and emptying the trash can) into the legitimate page. The injection string is a concatenation of the two commands, where the second command needs to be URL-encoded twice in order to force the application to clean the trash can only after the deletion of the mails.

Bypass WAFs input validation checks An attacker can exploit HPP flaws to launch traditional web attacks (e.g. XSS, SQL Injection) and by-pass web application firewalls (WAFs). Different technologies, such as ASP, concatenate the multiple values of a parameter, when it is twice repeated.

For example, the two query strings `var=foo&var=bar` and `var=foo,bar` are equivalent, but the second one is the result of a server-side concatenation of the parameters. Listing 6.1.2 shows how an attacker can setup a SQL Injection by splitting his query into multiple parameters with the same name. Lavakumar recently presented ([89]) an advance version of this example where the "commas" introduced by the concatenation are stripped out with inline comments (only on Microsoft SQL Server).

```
Standard: show_user.aspx?id=5;select+1,2,3+from+users+where+id=1--
Over HPP: show_user.aspx?id=5;select+1&id=2&id=3+from+users+where+id=1--

Standard: show_user.aspx?id=5+union+select+*+from+users--
Over HPP: show_user.aspx?id=5/*&id=*/union/*&id=*/select+*/*&id=*/from+users--
```

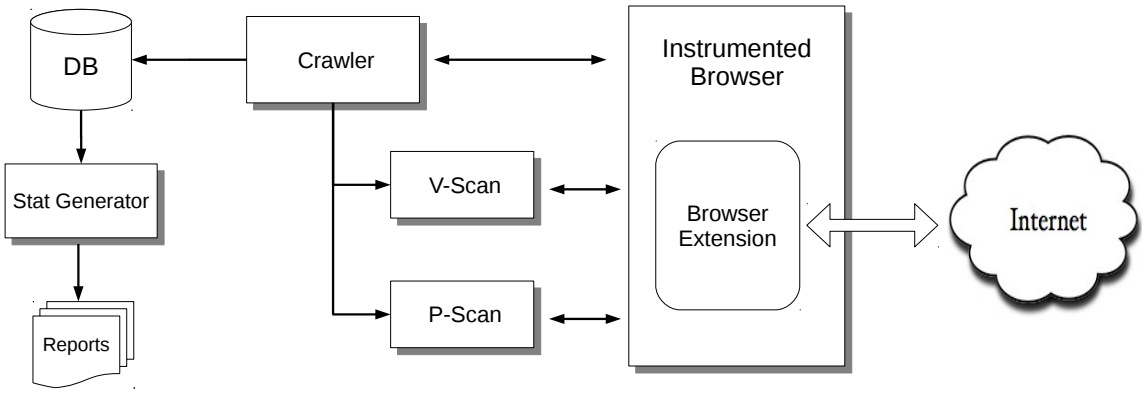


FIG. 6.1: Architecture of PAPAS

6.2 Automated HPP Detection

Our PArameter Pollution Analysis System (PAPAS) to automatically detect HPP vulnerabilities in websites consists of four main components: A browser, a crawler, and two scanners.

The first component is an instrumented browser that is responsible for fetching the webpages, rendering the content, and extracting all the links and form URLs contained in the page.

The second component is a crawler that communicates with the browser through a bidirectional channel. This channel is used by the crawler to inform the browser on the URLs that need to be visited, and on the forms that need to be submitted. Furthermore, the channel is also used to retrieve the collected information from the browser.

Every time the crawler visits a page, it passes the extracted information to the two scanners so that it can be analyzed. The parameter Precedence Scanner (P-Scan) is responsible for determining how the page behaves when it receives two parameters with the same name. The Vulnerability Scanner (V-Scan), in contrast, is responsible for testing the page to determine if it is vulnerable to HPP attacks. V-Scan does this by attempting to inject a new parameter inside one of the existing ones and analyzing the output. The two scanners also communicate with the instrumented browser in order to execute the tests.

All the collected information is stored in a database that is later analyzed by a statistics component that groups together information about the analyzed pages, and generates a report for the vulnerable URLs.

The general architecture of the system is summarized in Figure 6.1. In the following, we describe the approach that is used to detect HPP vulnerabilities and each component in more detail.

6.2.1 Browser and Crawler Components

Whenever the crawler issues a command such as the visiting of a new webpage, the instrumented browser in PAPAS first waits until the target page is loaded. After the browser is finished parsing the DOM, executing the client-side scripts, and loading additional resources, a browser extension (i.e., plugin) extracts the content, the list of links, and the forms in the page.

In order to increase the depth that a website can be scanned with, the instrumented browser in PAPAS uses a number of simple heuristics to automatically fill forms (similarly to previously proposed scanning solutions such as [86]). For example, random alphanumeric values of 8 characters are inserted into `password` fields and a default e-mail address is inserted into fields with the name `email`, `e-mail`, or `mail`.

For sites where the authentication or the provided inputs fail (e.g., because of the use of CAPTCHAs), the crawler can be assisted by manually logging into the application using the browser, and then specifying a regular expression to be used to prevent the crawler from visiting the log-out page (e.g., by excluding links that include the `cmd=logout` parameter).

6.2.2 P-Scan: Analysis of the Parameter Precedence

The P-Scan component analyzes a page to determine the precedence of parameters if multiple occurrences of the same parameter are injected into an application. For URLs that contain several parameters, each one is analyzed until the page's precedence has been determined or all available parameters have been tested.

The algorithm we use to test the precedence of parameters starts by taking the first parameter of the URL (in the form `par1=val1`), and generates a new parameter value `val2` that is similar to the existing one. The idea is to generate a value that would be accepted as being valid by the application. For example, a parameter that represents a page number cannot be replaced with a string. Hence, a number is cloned into a consecutive number, and a string is cloned into a same-length string with the first two characters modified.

In a second step, the scanner asks the browser to generate two new requests. The first request contains only the newly generated value `val2`. In contrast, the second request contains two copies of the parameter, one with the original value `val1`, and one with the value `val2`.

Suppose, for example, that a page accepts two parameters *par*1 and *par*2. In the first iteration, the first parameter is tested for the precedence behavior. That is, a new value *new_val* is generated and two requests are issued. In sum, the parameter precedence test is run on that pages that are the results of the three following requests:

```
Page0 - Original Url: application.php?
                        par1=val1&par2=val2
Page1 - Request 1:    application.php?
                        par1=new_val&par2=val2
Page2 - Request 2:    application.php?
                        par1=val1&par1=new_val&par2=val2
```

A naive approach to determine the parameter precedence would be to simply compare the three pages returned by the previous requests: If `Page1 == Page2`, then the second (last) parameter would have precedence over the first. If, however, `Page2 == Page0`, the application is giving precedence to the first parameter over the second.

Unfortunately, this straightforward approach does not work well in practice. Modern web applications are very complex, and often include dynamic content that may still vary even when the page is accessed with exactly the same parameters. Publicity banners, RSS feeds, real-time statistics, gadgets, and suggestion boxes are only a few examples of the dynamic content that can be present in a page and that may change each time the page is accessed.

The P-Scan component resolves the dynamic content problem in two stages. First, it

pre-processes the page and tries to eliminate all dynamic content that does not depend on the values of the application parameters. That is, P-Scan removes HTML comments, images, embedded contents, interactive objects (e.g., Java applets), CSS stylesheets, cross-domain iFrames, and client-side scripts. It also uses regular expressions to identify and remove "timers" that are often used to report how long it takes to generate the page that is being accessed. In a similar way, all the date and time strings on the page are removed.

The last part of the sanitization step consists of removing all the URLs that reference the page itself. The problem is that as it is very common for form actions to submit data to the same page, when the parameters of a page are modified, the self-referencing URLs also change accordingly. Hence, to cope with this problem, we also eliminate these URLs.

After the pages have been stripped off their dynamic components, P-Scan compares them to determine the precedence of the parameters. Let `P0', P1'`, and `P2'` be the sanitized versions of `Page0`, `Page1`, and `Page2`. The comparison procedure consists of five different tests that are applied until one of the tests succeeds:

I. Identity Test - The identity test checks whether the parameter under analysis has any impact on the content of the page. In fact, it is very common for query strings to contain many parameters that only affect the internal state, or some "invisible" logic of the application. Hence, if `P0' == P1' == P2'`, the parameter is considered to be ineffective.

II. Base Test - The base test is based on the assumption that the dynamic component stripping process is able to perfectly remove all dynamic components from the page that is under analysis. If this is the case, the second (last) parameter has precedence over the first if `P1'==P2'`. The situation is the opposite if `P2' == P0'`. Note that despite our efforts to improve the dynamic content stripping process as much as possible, in practice, it is rarely the case that the compared pages match perfectly.

III. Join Test - The join test checks the pages for indications that show that the two values of the homonym parameters are somehow combined together by the application. For example, it searches `P2'` for two values that are separated by commas, spaces, or that are contained in the same HTML tag. If there is a positive match, the algorithm concludes that the application is merging the values of the parameters.

IV. Fuzzy Test - The fuzzy test is designed to cope with pages whose dynamic components have not been perfectly sanitized. The test aims to handle identical pages that may show minor differences because of embedded dynamic parts. The test is based on confidence intervals. We compute two values, S_{21} and S_{20}, that represent how similar `P2'` is to the pages `P1'` and `P0'` respectively. The similarity algorithm we use is based on the Ratcliff/Obershelp pattern recognition algorithm, (also known as *gestalt pattern matching* [115]), and returns a number between 0 (i.e, completely different) to 1 (i.e., perfect match). The parameter precedence detection algorithm that we use in the fuzzy test works as follows:

```
if ABS(S21-S20) > DISCRIMINATION_THRESHOLD:
  if (S21 > S20) and (S21 > SIMILARITY_THRESHOLD):
    Precedence = last
  else (S20 > S21) and (S20 > SIMILARITY_THRESHOLD):
    Precedence = first
  else:
    Unknown precedence
```

```
else:
  Unknown precedence
```

To draw a conclusion, the algorithm first checks if the two similarity values are different enough (i.e., the values show a difference that is greater than a certain *discrimination threshold*). If this is the case, the closer match (if the similarity is over a minimum *similarity threshold*) determines the parameter precedence. In other words, if the page with the duplicated parameters is very similar to the original page, there is a strong probability that the web application is only using the first parameter, and ignoring the second. However, if the similarity is closer to the page with the artificially injected parameter, there is a strong probability that the application is only accepting the second parameter.

The two threshold values have been determined by running the algorithm on one hundred random webpages that failed to pass the base test, and for which we manually determined the precedence of parameters. The two experimental thresholds (set respectively to 0.05 and 0.75) were chosen to maximize the accuracy of the detection, while minimizing the error rate.

V. Error Test - The error test checks if the application crashes, or returns an "internal" error when an identical parameter is injected multiple times. Such an error usually happens when the application does not expect to receive multiple parameters with the same name. Hence, it receives an array (or a list) of parameters instead of a single value. An error occurs if the value is later used in a function that expects a well-defined type (such as a number or a string). In this test, we search the page under analysis for strings that are associated with common error messages or exceptions. In particular, we adopted all the regular expressions that the *SqlMap project* [56] uses to identify database errors in MySQL, PostgreSQL, MS SQL Server, Microsoft Access, Oracle, DB2, and SQLite.

If none of these five tests succeed, the parameter is discarded from the analysis. This could be, for example, because of content that is generated randomly on the server-side. The parameter precedence detection algorithm is then run again on the next available parameter.

6.2.3 V-Scan: Testing for HPP vulnerabilities

In this section, we describe how the V-Scan component tests for the presence of HTTP Parameter Pollution vulnerabilities in web applications.

For every page that V-Scan receives from the crawler, it tries to inject a URL-encoded version of an innocuous parameter into each existing parameter of the query string. Then, for each injection, the scanner verifies the presence of the parameter in links, action fields and hidden fields of forms in the answer page.

For example, in a typical scenario, V-Scan injects the pair "`%26foo%3Dbar`" into the parameter "`par1=val1`" and then checks if the "`&foo=bar`" string is included inside the URLs of links or forms in the answer page.

Note that we do not check for the presence of the vulnerable parameter itself (e.g., by looking for the string "`par1=val1&foo=bar`"). This is because web applications sometimes use a different name for the same parameter in the URL and in the page content. Therefore, the parameter "`par1`" may appear under a different name inside the page.

In more detail, V-Scan starts by extracting the list $P_{URL} = [P_{U1}, P_{U2}, \ldots P_{Un}]$ of the parameters that are present in the page URL, and the list $P_{Body} = [P_{B1}, P_{B2}, \ldots P_{Bm}]$ of the parameters that are present in links or forms contained in the page body.
It then computes the following three sets:

- $P_A = P_{URL} \cap P_{Body}$ is the set of parameters that appear unmodified in the URL and in the links or forms of the page.
- $P_B = p \mid p \in P_{URL} \ \wedge \ p \notin P_{Body}$ contains the URL parameters that do not appear in the page. Some of these parameters may appear in the page under a different name.
- $P_C = p \mid p \notin P_{URL} \ \wedge \ p \in P_{Body}$ is the set of parameters that appear somewhere in the page, but that are not present in the URL.

First, V-Scan starts by injecting the new parameter in the P_A set. We observed that in practice, in the majority of the cases, the application copies the parameter to the page body and maintains the same name. Hence, there is a high probability that a vulnerability will be identified at this stage. However, if this test does not discover any vulnerability, then the scanner moves on to the second set (P_B). In the second test, the scanner tests for the (less likely) case in which the vulnerable parameter is renamed by the application. Finally, in the final test, V-Scan takes the parameters in the P_C group, attempts to add these to the URL, and use them as a vector to inject the malicious pair. This is because webpages usually accept a very large number of parameters, not all of which are normally specified in the URL. For example, imagine a case in which we observe that one of the links in the page contains a parameter "`language=en`". Suppose, however, that this parameter is not present in the page URL. In the final test, V-Scan would attempt to build a query string like "`par1=var1&language=en%26foo%3Dbar`".

Note that the last test V-Scan applies can be executed on pages with an empty query string (but with parameterized links/forms), while the first two require pages that already contain a query string.

In our prototype implementation, the V-Scan component encodes the attacker pair using the standard URL encoding schema[2]. Our experiments show that this is sufficient for discovering HPP flaws in many applications. However, there is room for improvement as in some cases, the attacker might need to use different types of encodings to be able to trigger a bug. For example, this was the case of the HPP attack against Yahoo (previously described in Section 6.1) where the attacker had to double URL-encode the "cleaning of the trash can" action.

Handling special cases In our experiments, we identified two special cases in which, even though our vulnerability scanner reported an alert, the page was not actually vulnerable to parameter pollution.

In the first case, one of the URL parameters (or part of it) is used as the *entire* target of a link. For example:

```
Url:  index.php?v1=p1&uri=apps%2Femail.jsp%3Fvar1%3Dpar1
                                        %26foo%3Dbar
Link: apps/email.jsp?var1=par1&foo=bar
```

[2]URL Encoding Reference, `http://www.w3schools.com/TAGS/ref_urlencode.asp`

A parameter is used to store the URL of the target page. Hence, performing an injection in that parameter is equivalent to modifying its value to point to a different URL. Even though this technique is syntactically very similar to an HPP vulnerability, it is not a proper injection case. Therefore, we decided to consider this case as a false positive of the tool.

The second case that generates false alarms is the opposite of the first case. In some pages, the entire URL of the page becomes a parameter in one of the links. This can frequently be observed in pages that support printing or sharing functionalities. For example, imagine an application that contains a link to report a problem to the website's administrator. The link contains a parameter `page` that references the URL of the page responsible for the problem:

```
Url: search.html?session_id=jKAmSZx5%26foo%3Dbar&q=shoes

Link: service_request.html?page=search%2ehtml%3f
      session_id%3djKAmSZx5&foo=bar&q=shoes
```

Note that by changing the URL of the page, we also change the `page` parameter contained in the link. Clearly, this is not an HPP vulnerability.

Since the two previous implementation techniques are quite common in web applications, PAPAS would erroneously report these sites as being vulnerable to HPP. To eliminate such alarms and to make PAPAS suitable for large-scale analysis, we integrated heuristics into the V-Scan component to cross-check and verify that the vulnerabilities that are identified do not correspond to these two common techniques that are used in practice.

In our prototype implementation, in order to eliminate these false alarms, V-Scan checks that the parameter in which the injection is performed does not start with a scheme specifier string (e.g., `http://`). Then, it verifies that the parameter as a whole is not used as the target for a link. Furthermore, it also checks that the entire URL is not copied as a parameter inside a link. Finally, our vulnerability analysis component double-checks each vulnerability by injecting the new parameter *without* url-encoding the separator (i.e., by injecting `&foo=bar` instead of `%26foo%3Dbar`). If the result is the same, we know that the query string is simply copied inside another URL. While such input handling is possibly a dangerous design decision on the side of the developer, there is a high probability that it is intentional so we ignore it and do not report it by default. However, such checks can be deactivated anytime if the analyst would like to perform a more in-depth analysis of the website.

6.3 Implementation

The browser component of PAPAS is implemented as a Firefox extension, while the rest of the system is written in Python. The components communicate over TCP/IP sockets.

Similar to other scanners, it would have been possible to directly retrieve web pages without rendering them in a real browser. However, such techniques have the drawback that they cannot efficiently deal with dynamic content that is often found on Web pages (e.g., Javascript). By using a real browser to render the pages we visit, we are able to analyze the page as it is supposed to appear to the user after the dynamic content has

been generated. Also, note that unlike detecting cross site scripting or SQL injections, the ability to deal with dynamic content is a necessary prerequisite to be able to test for HPP vulnerabilities using a black-box approach.

The browser extension has been developed using the standard technology offered by the Mozilla development environment: a mix of Javascript and XML User Interface Language (XUL). We use XPConnect to access Firefox's XPCOM components. These components are used for invoking GET and POST requests and for communicating with the scanning component.

PAPAS supports three different operational modes: *fast mode*, *extensive mode* and *assisted mode*. The fast mode aims to rapidly test a site until potential vulnerabilities are discovered. Whenever an alert is generated, the analysis continues, but the V-Scan component is not invoked to improve the scanning speed. In the extensive mode, the entire website is tested exhaustively and all potential problems and injections are logged. The assisted mode allows the scanner to be used in an interactive way. That is, the crawler pauses and specific pages can be tested for parameter precedence and HPP vulnerabilities. The assisted mode can be used by security professionals to conduct a semi-automated assessment of a web application, or to test websites that require a particular user authentication.

PAPAS is also customizable and settings such as scanning depths, numbers of injections that are performed, waiting times between requests, and page loading timeouts are all configurable by the analyst.

Online Service We created an online version of PAPAS, shown in Fig. 6.2, that allows developers and maintainers of web applications to scan their own site. A user can submit the URL of her site for being tested. Our automated system will analyze the web application, and generate a nice HTML-formatted report when the scan is completed (Fig. 6.4). We implemented a challenge-response mechanism based on tokens to proof the site ownership of our users. The URL of this service is `http://papas.iseclab.org`.

6.3.1 Limitations

Our current implementation of PAPAS has several limitations. First, PAPAS does not support the crawling of links embedded in active content such as Flash, and therefore, is not able to visit websites that rely on active content technologies to navigate among the pages.

Second, currently, PAPAS focuses only on HPP vulnerabilities that can be exploited via client-side attacks (e.g., analogous to reflected XSS attacks) where the user needs to click on a link prepared by the attacker. Some HPP vulnerabilities can also be used to exploit server-side components (when the malicious parameter value is not included in a link but it is decoded and passed to a back-end component). However, testing for server-side attacks is more difficult than testing for client-side attacks as comparing requests and answers is not sufficient (i.e., similar to the difficulty of detecting stored SQL-injection vulnerabilities via black-box scanning). We leave the detection of server-side attacks to future work.

Listing 6.3.1 shows an example of a server-side attack and illustrates the difficulty of detecting such vulnerabilities by a scanner. In the example, an application prints the list of employees working in a given department. In this specific case, the "engineering" department is passed as a parameter with which a database select statement is constructed.

Fig. 6.2: PAPAS Online Service, Homepage

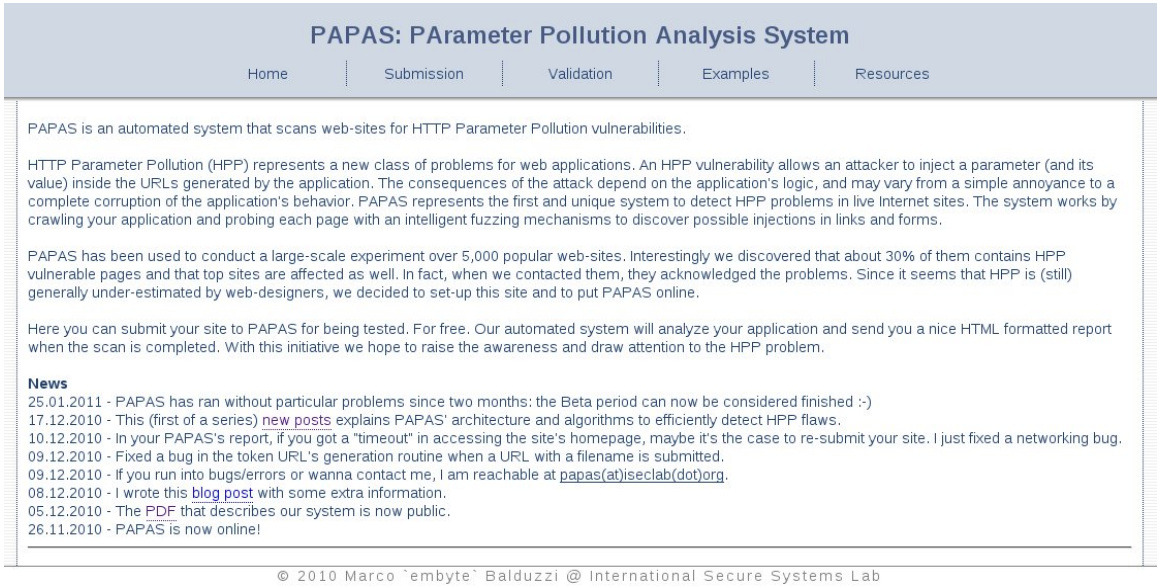
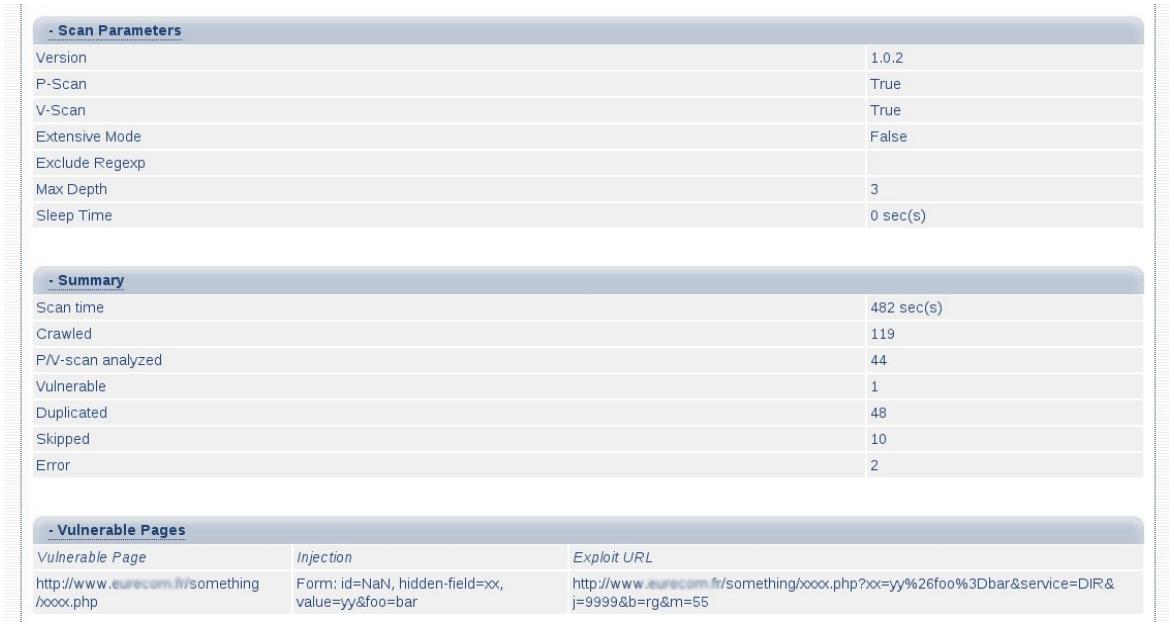


Fig. 6.3: PAPAS Online Service, Token Verification

Fig. 6.4: PAPAS Online Service, Report



- Scan Parameters

Version	1.0.2
P-Scan	True
V-Scan	True
Extensive Mode	False
Exclude Regexp	
Max Depth	3
Sleep Time	0 sec(s)

- Summary

Scan time	482 sec(s)
Crawled	119
P/V-scan analyzed	44
Vulnerable	1
Duplicated	48
Skipped	10
Error	2

- Vulnerable Pages

Vulnerable Page	*Injection*	*Exploit URL*
http://www.eurecom.fr/something/xxxx.php	Form: id=NaN, hidden-field=xx, value=yy&foo=bar	http://www.eurecom.fr/something/xxxx.php?xx=yy%26foo%3Dbar&service=DIR&j=9999&b=rg&m=55

Categories	# of Tested Applications	Categories	# of Tested Applications
Internet	698	Government	132
News	599	Social Networking	117
Shopping	460	Video	114
Games	300	Financial	110
Sports	256	Organization	106
Health	235	University	91
Science	222	Others	1401
Travel	175		

TAB. 6.2: TOP15 categories of the analyzed sites

In the attack, the attacker injects a new parameter `what=passwd` and is able to construct a new select filter in the database (note that this is possible because ASP concatenates two values with a comma).

```
Normal requests:
 URL      : printEmploys?department=engineering
 Back-end: dbconnect.asp?what=users&department=engineering
 Database: select users from table where department=engineering

HPP injected requests:
 URL      : printEmploys?department=engineering%26what%3Dpasswd
 Back-end: dbconnect.asp?what=users&department=engineering&what=passwd
 Database: select users,passwd from table where department=engineering
```

6.4 Evaluation

We evaluated our detection technique by running two experiments. In the first experiment, we used PAPAS to automatically scan a list of popular websites with the aim of measuring the prevalence of HPP vulnerabilities in the wild. We then selected a limited number of vulnerable sites and, in a second experiment, performed a more in-depth analysis of the detected vulnerabilities to gain a better understanding of the possible consequences of the vulnerabilities our tool automatically identified.

In the first experiment, we collected 5,000 unique URLs from the public database of Alexa. In particular, we extracted the top ranked sites from each of the *Alexa's categories* [29]. Each website was considered only once – even if it was present in multiple distinct categories, or with different top-level domain names such as `google.com` and `google.fr`.

The aim of our experiments was to quickly scan as many websites as possible. Our basic premise was that it would be likely that the application would contain parameter injection vulnerabilities on many pages and on a large number of parameters if the developers of the site were not aware of the HPP threat and had failed to properly sanitize the user input.

To maximize the speed of the tests, we configured the crawler to start from the homepage and visit the sub-pages up to a distance of three (i.e., three clicks away from the

Parameter Precedence	WebSites	
Last	2,237	(44.60%)
First	946	(18.86%)
Union	381	(7.60%)
Inconsistent	1,251	(24.94%)
Unknown	201	(4.00%)
Total	5,016	(100.00%)
Database Errors	238	(4.74%)

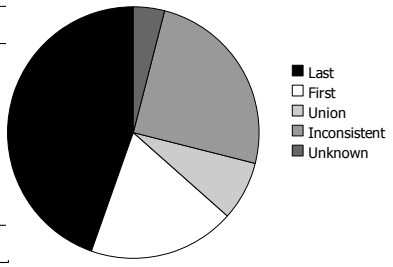


FIG. 6.5: Precedence when the same parameter occurs multiple time

website's entry point). For the tests, we only considered links that contained at least one parameter. In addition, we limited the analysis to 5 instances per page (i.e., a page with the same URL, but a different query string was considered a new instance). The global timeout was set to 15 minutes per site and the browser was customized to quickly load and render the pages, and run without any user interaction. Furthermore, we disabled pop-ups, image loading, and any plug-ins for active content technologies such as Flash, or Silverlight. An external watchdog was also configured to monitor and restart the browser in case it became unresponsive.

In 13 days of experiments, we successfully scanned 5,016 websites, corresponding to a total of 149,806 unique pages. For each page, our tool generated a variable amount of queries, depending on the number of detected parameters. The websites we tested were distributed over 97 countries and hundreds of different Alexa categories. Table 6.2 summarizes the 15 categories containing the higher number of tested applications.

Parameter Precedence For each website, the P-Scan component tested every page to evaluate the order in which the GET parameters were considered by the application when two occurrences of the same parameter were specified. The results were then grouped together in a per-site summary, as shown in Figure 6.5. The first column reports the type of parameter precedence. *Last* and *First* indicate that all the analyzed pages of the application uniformly considered the last or the first specified value. *Union* indicates that the two parameters were combined together to form a single value, usually by simply concatenating the two strings with a space or a comma. In contrast, the parameter precedence is set to *inconsistent* when different pages of the website present mismatching precedences (i.e., some pages favor the first parameter's value, others favor the last). The *inconsistent* state, accounting for a total of 25% of the analyzed applications, is usually a consequence of the fact that the website has been developed using a combination of heterogeneous technologies. For example, the main implementation language of the website may be PHP, but a few Perl scripts may still be responsible for serving certain pages.

Even though the lack of a uniform behavior can be suspicious, it is neither a sign, nor a consequence of a vulnerable application. In fact, each parameter precedence behavior (even the *inconsistent* case) is perfectly safe if the application's developers are aware of the HPP threat and know how to handle a parameter's value in the proper way. Unfortunately, as shown in the rest of the section, the results of our experiments suggest that many developers are not aware of HPP.

Figure 6.5 shows that for 4% of the websites we analyzed, our scanner was not been able to automatically detect the parameter precedence. This is usually due to two main reasons. The first reason is that the parameters do not affect (or only minimally affect) the rendered page. Therefore, the result of the page comparison does not reach the discrimination threshold. The second reason is the opposite of the first. That is, the page shows too many differences even after the removal of the dynamic content, and the result of the comparison falls below the similarity threshold (see Section 6.2.2 for the full algorithm and an explanation of the threshold values).

The scanner found 238 applications that raised an SQL error when they were tested with duplicated parameters. Quite surprisingly, almost 5% of the most popular websites on the Internet failed to properly handle the user input, and returned an "internal" error page when a perfectly-legal parameter was repeated twice. Note that providing two parameters with the same name is a common practice in many applications, and most of the programming languages provide special functionalities to access multiple values. Therefore, this test was not intended to be an attack against the applications, but only a check to verify which parameter's value was given the precedence. Nevertheless, we were surprised to note error messages from the websites of many major companies, banks and government institutions, educational sites, and other popular websites.

HPP Vulnerabilities PAPAS discovered that 1499 websites (29.88% of the total we analyzed) contained at least one page vulnerable to HTTP Parameter Injection. That is, the tool was able to automatically inject an encoded parameter inside one of the existing parameters, and was then able to verify that its URL-decoded version was included in one of the URLs (links or forms) of the resulting page.

However, the fact that it is possible to inject a parameter does not reveal information about the significance and the consequences of the injection. Therefore, we attempted to verify the number of *exploitable* applications (i.e., the subset of vulnerable websites in which the injected parameter could potentially be used to modify the behavior of the application).

We started by splitting the vulnerable set into two separate groups. In 872 websites (17.39%), the injection was on a link or a form's action field. In the remaining 627 cases (12.5%), the injection was on a form's hidden field.

For the first group, our tool verified if the parameter injection vulnerability could be used to override the value of one of the existing parameters in the application. This is possible only if the parameter precedence of the page is consistent with the position of the injected value. For example, if the malicious parameter is always added to the end of the URL and the first value has parameter precedence, it is impossible to override any existing parameter.

When the parameter precedence is not favorable, a vulnerable application can still be exploitable by injecting a new parameter (that differs from all the ones already present in the URL) that is accepted by the target page.

For example, consider a page `target.pl` that accepts an `action` parameter. Suppose that, on the same page, we find a page `poor.pl` vulnerable to HPP:

```
Url:  poor.pl?par1=val1%26action%3Dreset
Link: target.pl?x=y&w=z&par1=val1&action=reset
```

Since in Perl the parameter precedence is on the *first* value, it is impossible to over-

ride the `x` and `w` parameters. However, as shown in the example, the attacker can still exploit the application by injecting the `action` parameter that she knows is accepted by the `target.pl` script. Note that while the parameter overriding test was completely automated, this type of injection required a manual supervision to verify the effects of the injected parameter on the web application.

The final result was that at least 702 out of the 872 applications of the first group were exploitable. For the remaining 170 pages, we were not able, through a parameter injection, to affect the behavior of the application.

For the applications in the second group, the impact of the vulnerability is more difficult to estimate in an automated fashion. In fact, since modern browsers automatically encode all the form fields, the injected parameter will still be sent in a url-encoded form, thus making an attack ineffective.

In such a case, it may still be possible to exploit the application using a two-step attack where the malicious value is injected into the vulnerable field, it is propagated in the form submission, and it is (possibly) decoded and used in a later stage. In addition, the vulnerability could also be exploited to perform a server-side attack, as explained in Section 6.3.1. However, using a black-box approach, it is very difficult to automatically test the exploitability of multi-step or server-side vulnerabilities. Furthermore, server-side testing might have had ethical implications (see Section 6.4.2 for discussion). Therefore, we did not perform any further analysis in this direction.
To conclude, we were able to confirm that in (at least) 702 out of the 1499 vulnerable websites (i.e., 46.8%) that PAPAS identified, it would have been possible to exploit the HPP vulnerability to override one of the hard-coded parameters, or to inject another malicious parameter that would affect the behavior of the application.

Figure 6.6 shows the fraction of vulnerable and exploitable applications grouped by the different Alexa categories. The results are equally divided, suggesting that important financial and health institutions do not seem to be more security-aware and immune to HPP than leisure sites for sporting and gaming.

False Positives In our vulnerability detection experiments, the false positives rate was 1.12% (10 applications). All the false alarms were due to parameters that were used by the application as an entire target for one of the links. The heuristic we implemented to detect these cases (explained in Section 6.2.3) failed because the applications applied a transformation to the parameter before using it as a link's URL.

Note that, to maximize efficiency, our results were obtained by crawling each website at a maximum depth of three pages. In our experiments, we observed that 11% of the vulnerable pages were directly linked from the homepage, while the remaining 89% were equally distributed between the distance of 2 and 3. This trend suggests that it is very probable that many more vulnerabilities could have been found by exploring the sites in more depth.

6.4.1 Examples of Discovered Vulnerabilities

Our final experiments consisted of the further analysis of some of the vulnerable websites that we identified. Our aim was to gain an insight into the real consequences of the HPP vulnerabilities we discovered.

The analysis we performed was assisted by the V-Scan component. When invoked in

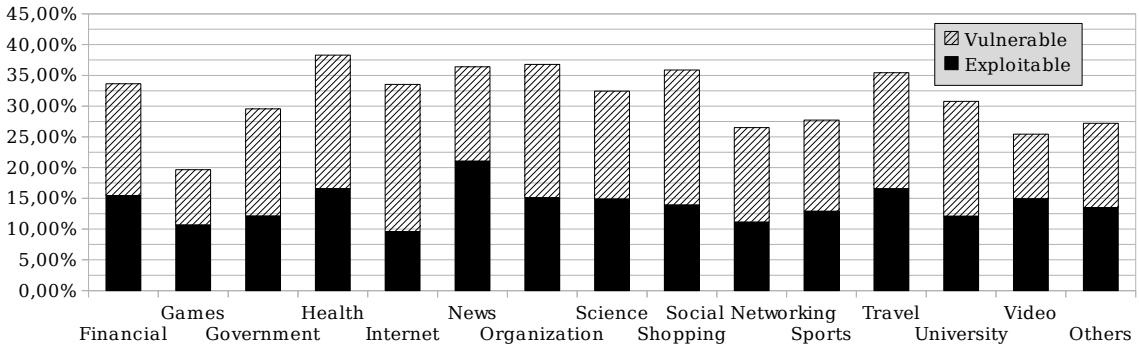


FIG. 6.6: Vulnerability rate for category

extensive mode, V-Scan was able to explore in detail the web application, enumerating all the vulnerable parameters. For some of the websites, we also registered an account and configured the scanner to test the authenticated part of the website.

HPP vulnerabilities can be abused to run a wide range of different attacks. In the rest of this section, we discuss the different classes of problems we identified in our analysis with the help of real-world examples.

The problems we identified affected many important and well-known websites. Since, at the time of writing, we have not yet received confirmation that all of the vulnerabilities have been fixed, we have anonymized the description of the following real-word cases.

Sharing Components Facebook, Twitter, Digg and other social networking sites offer a *share component* to easily share the content of a webpage over a user profile. Many news portals nowadays integrate these components with the intent of facilitating the distribution of their news.

By reviewing the vulnerability logs of the tested applications, we noticed that different sites allowed a parameter injection on the links referencing the share component of a note social network. In all those cases, a vulnerable parameter would allow an attacker to alter the request sent to the social network and to trick the victim into sharing a page chosen by the attacker. For example, it was possible for an attacker to exploit these vulnerabilities to corrupt a shared link by overwriting the reference with the URL of a drive-by-download website.

In technical terms, the problem was due to the fact that it was possible to inject an extra *url-to-share* parameter that could overwrite the value of the parameter used by the application. For example:

```
Url:
<site>/shareurl.htm?PG=<default url>&zItl=<description>
              %26url-to-share%3Dhttp://www.malicious.com
Link:
http://www.social-network.com/sharer.php?
      url-to-share=<default url>&t=<description>&
      url-to-share=http://www.malicious.com
```

Even though the problem lies with the websites that use the share component, the social network facilitated the exploitation by accepting multiple instances of the same parameter, and always considering the latest value (i.e., the one on the right).

We notified the security team of the affected provider and proposed a simple solution based on the filtering of all incoming sharing requests that include duplicate parameters. The team promptly acknowledged the issue and informed us that they were willing to put in place our countermeasure.

CSRF via HPP Injection Many applications use hidden parameters to store a URL that is later used to redirect the users to an appropriate page. For example, social networks commonly use this feature to redirect new users to a page where they can look up a friend's profile.

In some of these sites, we observed that it was possible for an attacker to inject a new *redirect* parameter inside the registration or the login page so that it could override the hard-coded parameter's value. On one social-network website, we were able to inject a custom URL that had the effect of automatically sending friend requests after the login. In another site, by injecting the malicious pair into the registration form, an attacker could perform different actions on the authenticated area.

This problem is a CSRF attack that is carried out via an HPP injection. The advantages compared to a normal CSRF is that the attack URL is injected into the real login/registration page. Moreover, the user does not have to be already logged into the target website because the action is automatically executed when the user logs into the application. However, just like in normal CSRF, this attack can be prevented by using security tokens.

Shopping Carts We discovered different HPP vulnerabilities in online shopping websites that allow the attacker to tamper with the user interaction with the shopping cart component.

For example, in several shopping websites, we were able to force the application to select a particular product to be added into the user's cart. That is, when the victim checks out and would like to pay for the merchandise, she is actually paying for a product that is different from the ones she actually selected. On an Italian shopping portal, for example, it was even possible to override the ID of the product in such a way that the browser was still showing the image and the description of the original product, even when the victim was actually buying a different one.

Financial Institutions We ran PAPAS against the authenticated and non-authenticated areas of some financial websites and the tool automatically detected several HPP vulnerabilities that were potentially exploitable. Since the links involved sensitive operations (such as increasing account limits and manipulating credit card operations), we immediately stopped our experiments and promptly informed the security departments of the involved companies. The problems were acknowledged and are currently being fixed.

Tampering with Query Results In most cases, the HPP vulnerabilities that we discovered in our experiments allow the attacker to tamper with the data provided by the vulnerable website, and to present to the victim some information chosen by the attacker.

On several popular news portals, we managed to modify the news search results to hide certain news, to show the news of a certain day with another date, or to filter the news of a specific source/author. An attacker can exploit these vulnerabilities to promote some particular news, or conceal news that can hurt his person/image, or even subvert the information by replacing an article with an older one.

Also some multimedia websites were vulnerable to HPP attacks. In several popular sites, an attacker could override the video links and make them point to a link of his choice (e.g., a drive-by download site), or alter the results of a query to inject malicious multimedia materials. In one case, we were able to automatically register a user to a specific streaming event.

Similar problems also affected several popular search engines. We noticed that it would have been possible to tamper with the results of the search functionality by adding special keywords, or by manipulating the order in which the results are shown. We also noticed that on some search engines, it was possible to replace the content of the commercial suggestion boxes with links to sites owned by the attacker.

6.4.2 Ethical Considerations

Crawling and automatically testing a large number of applications may be considered an ethically sensitive issue. Clearly, one question that arises is if it is ethically acceptable and justifiable to test for vulnerabilities in popular websites.

Analogous to the real-world experiments conducted by Jakobsson et al. in [79, 80], we believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real-world. Unfortunately, criminals do not have any second thoughts about discovering vulnerabilities in the wild. As researchers, we believe that our experiments helped many websites to improve their security. Furthermore, we were able to raise some awareness about HPP problems in the community.

Also, note that:

- PAPAS only performed client-side checks. Similar client-side vulnerability experiments have been peformed before in other studies (e.g., for detecting cross site scripting, SQL injections, and CSRF in the wild [86, 116]). Furthermore, we did not perform any server-side vulnerability analysis because such experiments had the potential to cause harm.
- We only provided the applications with innocuous parameters that we knew that the applications were already accepting, and did not use any malicious code as input.
- PAPAS was not powerful enough to influence the performance of any website we investigated, and the scan activities was limited to 15 minutes to further reduce the generated traffic.
- We informed the concerned sites of any critical vulnerabilities that we discovered.
- None of the security groups of the websites that we interacted with complained to us when we informed them that we were researchers, and that we had discovered vulnerabilities on their site with a tool that we were testing. On the contrary, many people were thankful to us that we were informing them about vulnerabilities in their code, and helping them make their site more secure.

6.5 Summary

Web applications are not what they used to be ten years ago. Popular web applications have now become more dynamic, interactive, complex, and often contain a large number

of multimedia components. Unfortunately, as the popularity of a technology increases, it also becomes a target for criminals. As a result, most attacks today are launched against web applications. Vulnerabilities such as cross site scripting, SQL injection, and cross site request forgery are well-known and have been intensively studied by the research community. Many solutions have been proposed, and tools have been released. However, a new class of injection vulnerabilities called HTTP Parameter Pollution (HPP) that was first presented at the OWASP conference [110] in 2009 has not received as much attention. If a web application does not properly sanitize the user input for parameter delimiters, using an HPP vulnerability, an attacker can compromise the logic of the application to perform client-side or server-side attacks.

In this chapter, we presented the first automated approach for the discovery of HPP vulnerabilities in web applications. Our prototype implementation called PArameter Pollution Analysis System (PAPAS) is able to crawl websites and discover HPP vulnerabilities by parameter injection. In order to determine the feasibility of our approach and to assess the prevalence of HPP vulnerabilities on the Internet today, we analyzed more than 5,000 popular websites on a large scale. Our results show that about 30% of the sites we analyzed contain vulnerable parameters and that at least 14% of them can be exploited using HPP. A large number of well-known, high-profile websites such as search engines, ecommerce and online banking applications were affected by HPP vulnerabilities. We informed the sites for which we could obtain contact information, and some of these sites wrote back to us and confirmed our findings. We hope that this research will help raise awareness and draw attention to the HPP problem.

Chapitre 7

Elastic Compute Cloud Risks

This chapter explores the general security risks associated with using virtual server images from the public catalogs of cloud service providers such as Amazon EC2.

We describe the design and implementation of an automated system called SatanCloud that we used to conduct a large-scale study over more than five thousands images provided by Amazon in four of its data centers (U.S. East/West, Europe and Asia).

Our measurements demonstrate that both the users and the providers of server images may be vulnerable to security risks such as unauthorized access, malware infections, and loss of sensitive information.

7.1 Introduction

Cloud computing has changed the view on IT as a pre-paid asset to a pay-as-you-go service. Several companies such as *Amazon Elastic Compute Cloud* [19], *Rackspace* [21], *IBM SmartCloud* [24], *Joyent Smart Data Center* [26] or *Terremark vCloud* [22] are offering access to virtualized servers in their data centers on an hourly basis. Servers can be quickly launched and shut down via application programming interfaces, offering the user a greater flexibility compared to traditional server rooms. This paradigm shift is changing the existing IT infrastructures of organizations, allowing smaller companies that cannot afford a large infrastructure to create and maintain online services with ease.

A popular approach in cloud-based services is to allow users to create and share virtual images with other users. For example, a user who has created a legacy Linux Debian Sarge image may decide to make this image public so that other users can easily reuse it. In addition to user-shared images, the cloud service provider may also provide customized public images based on common needs of their customers (e.g., an Ubuntu web server image that has been pre-configured with MySQL, PHP and an Apache). This allows the customers to simply instantiate and start new servers, without the hassle of installing new software themselves.

Unfortunately, while the trust model between the cloud user and the cloud provider is well-defined (i.e., the user can assume that cloud providers such as Amazon and Microsoft are not malicious), the trust relationship between the provider of the virtual image and the cloud user is not as clear.

In this chapter, we investigate the security risks related with the use of public images from the Amazon EC2 service. Over several months, we designed and ran security tests on public AMIs that aimed to identify security vulnerabilities, problems, and risks for cloud

users as well as the cloud providers. We instantiated and analyzed over five thousands public images provided by Amazon in its data centers located in Virginia (US-East), Northern California (US-West), Ireland (EU-West) and Singapore (AP-Southwest). We analyzed both Linux and Windows-based images, checking for a wide-range of security problems such as the prevalence of malware, the quantity of sensitive data left on such images, and the privacy risks of sharing an image on the cloud.

We identified three main threats related, respectively, to: 1) secure the image against external attacks, 2) secure the image against a malicious image provider, and 3) sanitize the image to prevent users from extracting and abusing private information left on the disk by the image provider. For example, in our experiments we identified many images in which a user can use standard tools to *undelete* files from the filesystem and recover important documents including credentials and private keys. Public cloud server images are highly useful for organizations but, if users are not properly trained, the risk associated in using these images can be quite high. The fact that these machines come pre-installed and pre-configured may communicate the wrong message, i.e., that they can provide an easy-to-use "shortcut" for user that do not have the skills to configure and setup a complex server. Reality is quite different, and this work shows how many different aspects must be considered to make sure that a virtual image can be operated securely.

For this reason, during our study we had continuous contact with the Amazon Web Services Security Team. Even though Amazon is not responsible of what users put into their images, the team has been prompt in addressing the security risks identified and described in this chapter. Meanwhile, it has published public bulletins and tutorials to train users on how to use Amazon Machine Images (AMIs) in a secure way [124, 123]. A more detailed description of the Amazon feedback is provided in Section 7.6.

7.2 Overview of Amazon EC2

The Amazon *Elastic Compute Cloud (EC2)* is an Infrastructure-as-a-Service cloud provider where users can rent virtualized servers (called *instances*) on an hourly base. In particular, each user is allowed to run any pre-installed virtual machine image (called *Amazon Machine Image*, or *AMI*) on this service. To simplify the setup of a server, Amazon offers an online catalog where users can choose between a large number of AMIs that come pre-installed with common services such as web servers, web applications, and databases. An AMI can be created from a live system, a virtual machine image, or another AMI by copying the filesystem contents to the Amazon *Simple Storage Service (S3)* in a process called *bundling*. Public images may be available for free, or may be associated with a *product code* that allows companies to bill an additional usage cost via the Amazon *DevPay* payment service. Thus, some of these public machines are provided by companies, some are freely shared by single individuals, and some are created by the Amazon team itself.

In order to start an image, the user has to select a resource configuration (differing in processing, memory, and IO performance), a set of credentials that will be used for login, a firewall configuration for inbound connections (called a *security group*), and the *region* of the data center in which the machine will be started. Amazon data centers are currently located in the US (Northern Virginia and Northern California), Europe (Ireland), and Asia (Singapore). An additional data center (Tokyo) was added after we had completed our experiments. Hence, this data center will not be discussed.

Currently, there are three different pricing models available: The first one is a fixed pricing scheme where the customer pays per instance-hour, the second one is a subscription

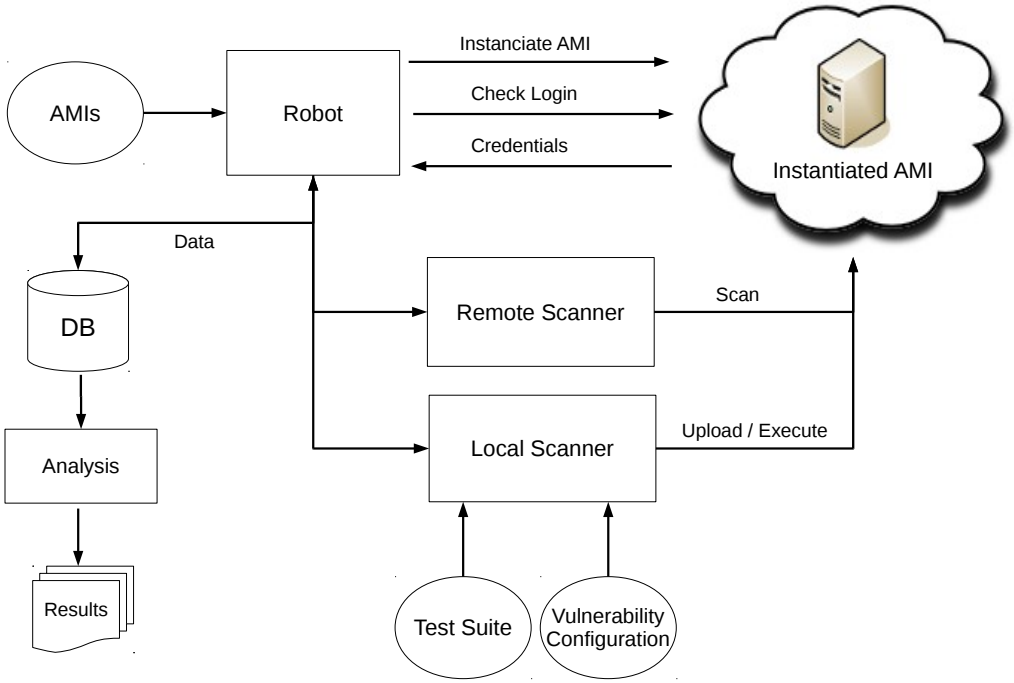


FIG. 7.1: System Architecture

model with an initial fee and lower instance-hour costs, and the third one (called *spot instances*) is a model where prices vary according to the current load of the data centers. This model lets the customer specify a maximum price he is willing to pay for an instance in addition to the other instance parameters. If the current spot price drops below this threshold, the instance is started, and if the spot price rises above the threshold, it is terminated, thus making this model suitable for interruptible tasks.

When an AMI is instantiated, its public DNS address is announced via the Amazon API, and the machine is made accessible via SSH on port 22 (Linux) or Remote Desktop on port 3389 (Windows). An important aspect of this cloud computing service is that the instance's maintenance is completely under the responsibility of the user. That is, she is the one who can be held responsible for any content provided by the machine, and she is the one who has to assure its security. This includes, for example, the usual administration tasks of maintaining the configuration in a secure state (i.e., applying patches for vulnerable software, choosing the right passwords, and firewall configuration), and only allowing secure, encrypted communication protocols.

7.3 AMI Testing Methodology

To conduct our security evaluation, we developed an automated system to instantiate and test the Amazon's AMIs. The architecture of our system is highlighted in Fig. 7.1, and consists of three main components. The *Robot* is the part of the system that is responsible for instantiating the AMIs, and fetching the corresponding login credentials. Since Amazon does not control which credentials are configured in the public images, our tool was configured to try a list of the most common user names (e.g., `root`, `ec2-user`, `ubuntu`, and `bitnami` for Linux). Despite these attempts, there are cases in which the

robot may fail to retrieve the correct login information. This is the case, for example, for AMIs whose credentials are distributed only to the image provider's customers by companies that make business by renting AMIs. Hence, these type of images are outside the scope of our evaluation.

After an AMI has been successfully instantiated by the robot, it is tested by two different scanners. The *Remote Scanner* collects the list of open ports[1] using the `NMap` tool [93], and downloads the index page of the installed web applications. In Section 7.5, we explain how an attacker can use this information as a fingerprint to identify running images. The *Local Scanner* component is responsible for uploading and running a set of tests. The test suite to be executed is packaged together in a self-extracting archive, uploaded to the AMI, and run on the machine with administrative privileges. In addition, the Local Scanner also analyzes the system for known vulnerabilities using the Nessus tool [125]. For AMIs running Microsoft Windows, the scripting of automated tasks is complicated by the limited remote administration functionalities offered by the Windows environment. In this case, we mounted the remote disk and transfered the data using the `SMB/Netbios` subsystem. We then used the `psexec` tool [120] to execute remote commands and invoke the tests.

The test suite uploaded by the Local Scanner includes 24 tests grouped in 4 categories: general, network, privacy, and security. The complete list is summarized in Table 7.1.

The *general* category contains tests that collect general information about the system (e.g. the Linux distribution name, or the Windows version), the list of running processes, the file-system status (e.g., the mounted partitions), the list of installed packages, and the list of loaded kernel modules. In addition to these basic tests, the general category also contains scripts that save a copy of interesting data, such as emails (e.g., `/var/mail`), log files (e.g., `/var/log` and `%USER\Local Settings`), and installed web applications (e.g., `/var/www` and `HKEY_LOCAL_MACHINE\SOFTWARE`).

The *privacy* test cases focus on finding any sensitive information that may have been forgotten by the user that published the AMI. This includes, for example, unprotected private keys, application history files, shell history logs, and the content of the directory saved by the general test cases. Another important task of this test suite is to scan the filesystem to retrieve the contents of undeleted files.

The *network* test suite focuses on network-related information, such as shared directories and the list of open sockets. These lists, together with the processes bound to the sockets, can be used to verify if the image is establishing suspicious connections.

Finally, the *security* test suite consists of a number of well-known audit tools for Windows and Linux. Some of these tools look for the evidence of known rootkits, Trojans and backdoors (e.g. `Chkrootkit`, `RootkitHunter` and `RootkitRevealer`), while others specifically check for processes and sockets that have been hidden from the user (`PsTools/PsList` and `unhide`). In this phase, we also run the `ClamAV` antivirus software (see Section 7.4.2) to scan for the presence of known malware samples.

These security tests also contain checks for credentials that have been left or forgotten on the system (e.g., database passwords, login passwords, and SSH public keys). As already mentioned in an Amazon report published in June 2011 [27], these credentials could potentially be used as backdoors to allows attackers to log into running AMIs. In fact, as we show in Section 7.5, it is relatively easy to scan the address space used by Amazon EC2 and map running instances to the corresponding AMI IDs. As a consequence, if an

[1] Since Amazon does not allow external portscans of EC2 machines, we first established a virtual private network connection to the AMI through SSH, and then scanned the machine through this tunnel.

Tests	Type	Details	OS
System Information	General	-	Windows + Linux
Logs/eMails/WWW Archive	General	-	Linux
Processes and File-System	General	-	Windows + Linux
Loaded Modules	General	lsmod	Linux
Installed Packages	General	-	Linux
General Network Information	Network	Interfaces, routes	Windows + Linux
Listening and Established Sockets	Network	-	Windows + Linux
Network Shares	Network	Enabled Shares	Windows + Linux
History Files	Privacy	Common Shells + Browsers	Windows + Linux
AWS/SSH Private Keys	Privacy	Loss of sensitive info	Linux
Undeleted Data	Privacy	(Only on X AMIs)	Linux
Last logins	Privacy	-	Linux
SQL Credentials	Privacy/Security	MySQL and PostgresSQL	Linux
Password Credentials	Privacy/Security	Enabled Logins	Windows + Linux
SSH Public Keys	Security	Backdoor access	Linux
Chkrootkit	Security	Rootkit	Linux
RootkitHunter	Security	Rootkit	Linux
RootkitRevealer	Security	Rootkit	Windows
Lynis Auditing Tool	Security	General Security Issues	Linux
Clam AV	Security	Antivirus	Windows + Linux
Unhide	Security	Processes/Sockets Hiding	Linux
PsList	Security	Processes Hiding	Windows
Sudoers Configuration	Security	-	Linux

TAB. 7.1: Details of the tests included in the automated AMI test suite

attacker discovers that a certain instance is running an image with forgotten credentials, she may try to authenticate and remotely log in into the machine. For this reason, as a part of the security tests, we also analyzed the Linux `sudo` configuration to verify if the leftover credentials would allow the execution of code with administrative privileges.

7.4 Results of the Large Scale Analysis

Over a period of five months, between November 2010 to May 2011, we used our automated system to instantiate and analyze all Amazon images available in the Europe, Asia, US East, and US West data centers. In total, the catalog of these data centers contained 8,448 Linux AMIs and 1,202 Windows AMIs. Note that we were successfully able to analyze in depth a total of 5,303 AMIs. In the remaining cases, a number of technical problems prevented our tool to successfully complete the analysis. For example, sometimes an AMI did not start because the corresponding manifest file was missing, or corrupted. In some cases, the running image was not responding to SSH, or Remote Desktop connections. In other cases, the Amazon API failed to launch the machine, or our robot was not able to retrieve valid login credentials. These problems were particularly common for Windows machines where, in 45% of the images, the Amazon service was not able to provide us with a valid username and password to login into the machine. Nevertheless, we believe that a successful analysis of over 5,000 different images represents a sample large enough to be representative of the security and privacy status of publicly available AMIs.

Table 7.2 shows a number of general statistics we collected from the AMIs we analyzed. Our audit process took on average 77 minutes for Windows machines, and 21 minutes for the Linux images. This large difference is due to two main reasons: first, Windows

Average #/AMI	Windows	Linux
Audit duration	77 min	21 min
Installed packages	–	416
Running Processes	32	54
Shares	3.9	0
Established sockets	2.75	2.52
Listening sockets	22	6
Users	3.8	24.8
Used disk space	1.07 GB	2.67 GB

TAB. 7.2: General Statistics

machines in the Amazon cloud take a much longer time to start, and, second, our antivirus test was configured to analyze the entire Windows file-system, while only focused the analysis on directories containing executables for the Linux machines.

In the rest of this section, we present and discuss the results of the individual test suites.

7.4.1 Software Vulnerabilities

The goal of this first phase of testing is to confirm the fact that the software running on each AMIs is often out of date and, therefore, must be immediately updated by the user after the image is instantiated.

For this purpose, we decided to run the Nessus [125], an automated vulnerability scanner, on each AMI under test. In order to improve the accuracy of the results, our testing system provided Nessus with the image login credentials, so that the tool was able to perform a more precise local scan. In addition, to further reduce the false positives, the vulnerability scanner was automatically configured to run only the tests corresponding to the actual software installed on the machine.

Nessus classifies each vulnerability with a *severity level* ranging from 0 to 3. Since we were not interested in analyzing each single vulnerability, but just in assessing the general security level of the software that was installed, we only considered vulnerabilities with the highest severity (e.g., *critical* vulnerabilities such as remote code execution).

As reported in Table 7.3, 98% of Windows AMIs and 58% of Linux AMIs contain software with critical vulnerabilities. This observation was not typically restricted to a single application but often involved multiple services: an average of 46 for Windows and 11 for Linux images (the overall distribution is reported in Figure 7.2). On a broader scale, we observed that a large number of images come with software that is more than two years old. Our findings empirically demonstrate that renting and using an AMI without any adequate security assessment poses a real security risk for users. To further prove this point, in Section 7.4.2, we describe how one of the machines we were testing was probably compromised by an Internet malware in the short time that we were running our experiments.

Table 7.3 also reports the most common vulnerabilities that affect Windows and Linux AMIs. For example, the vulnerabilities `MS10-098` and `MS10-051` affect around 92% and 80% of the tested Windows AMIs, and allows remote code execution if the user views a particular website using the Internet Explorer. Microsoft Office and the Windows'

	Windows	Linux
Tested AMIs	253	3,432
Vulnerable AMIs	249	2,005
With vulns <=2 Years	145	1,197
With vulns <=3 Years	38	364
With vulns <=4 Years	2	106
Avg. # Vuln/AMI	46	11
TOP 10 Vuln.	MS10-037, MS10-049, MS10-051, MS10-073, MS10-076, MS10-083, MS10-090, MS10-091, MS10-098, MS11-05	CVE-2009-2730, CVE-2010-0296, CVE-2010-0428, CVE-2010-0830, CVE-2010-0997, CVE-2010-1205, CVE-2010-2527, CVE-2010-2808, CVE-2010-3847, CVE-2011-0997

Tab. 7.3: Nessus Results

standard text editor Wordpad contained in 81% of the Windows AMIs allow an attacker to take control of the vulnerable machine by opening a single malicious document (i.e., vulnerability `MS10-83`). A similar vulnerability (i.e., `CVE-2010-1205`) affects Linux AMIs as well: A PNG image sent to a vulnerable host might allow a malicious user to run code remotely on the AMI. We also observed that 87 public Debian AMIs come with the now notorious SSH/OpenSSL vulnerability discovered in May 2008 (i.e., `CVE-2008-0166`) in which, since the seed of the random number generator used to generate SSH keys is predictable, any SSH key generated on the vulnerable systems needs to be considered as being compromised [13].

7.4.2 Security Risks

Malware

As part of our tests, we used ClamAV [20], an open source antivirus engine, to analyze the filesystem on the target AMI. ClamAV contains about 850,000 signatures to identify different types of known malware instances such as viruses, worms, spyware, and trojans. Since most of the existing malware targets the Windows operating systems, we analyzed the complete file-system tree of Windows AMIs, while we limited the coverage for Linux AMIs to common binary directories (e.g. `/usr/bin, /bin`, and `/sbin`). As a consequence, the scan time took an average of 40 minutes for a Windows installation, and less then a minute for a Linux one.

In our malware analysis, we discovered two infected AMIs, both Windows-based. The first machine was infected with a `Trojan-Spy` malware (variant 50112). This trojan has a wide range of capabilities, including performing key logging, monitoring processes on the computer, and stealing data from files saved on the machine. By manually analyzing this machine, we found that it was hosting different types of suspicious content such as `Trojan.Firepass`, a tool to decrypt and recover the passwords stored by Firefox. The second infected machine contained variant 173287 of the `Trojan.Agent` malware. This malware allows a malicious user to spy on the browsing habits of users, modify Internet Explorer settings, and download other malicious content.

While we were able to manually confirm the first case, we were unable to further ana-

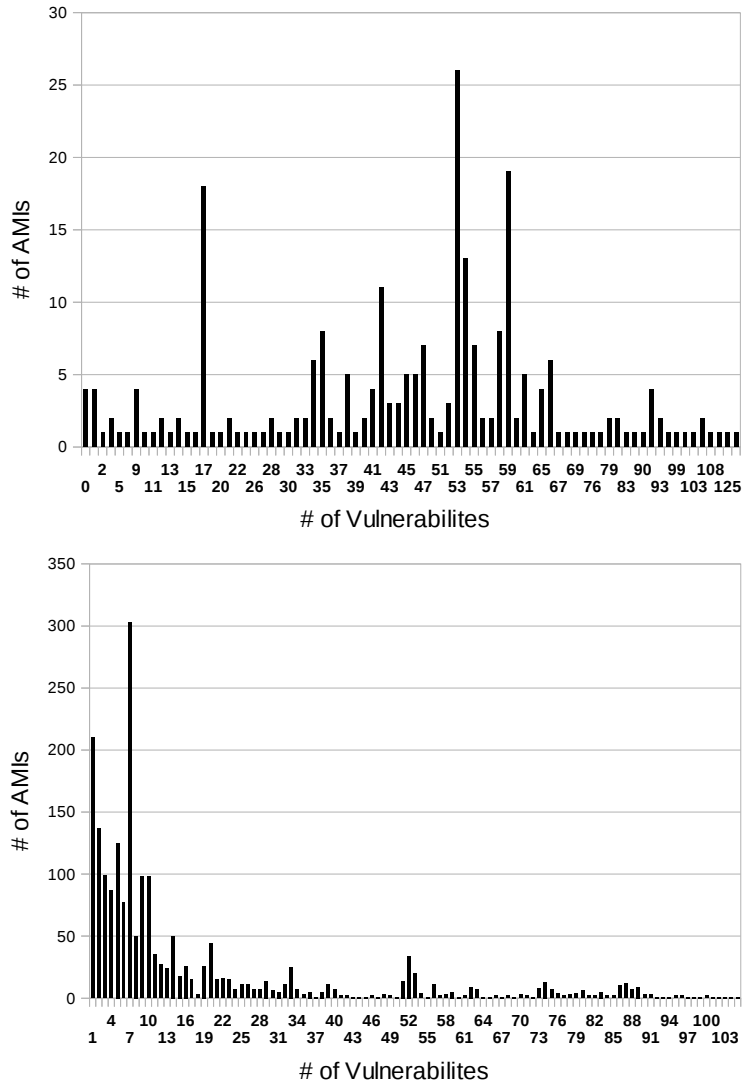


Fig. 7.2: Distribution AMIs / Vulnerabilites (Windows and Linux)

lyze the second infected machine. In fact, after we rented it again for a manual analysis a few hours after the automated test, the infected files did not existed anymore. Hence, we believe that the AMI was most probably compromised by an automatically propagating malware during the time that we were executing our tests. In fact, the software vulnerability analysis showed that different services running on the machine suffered from known, remotely exploitable, vulnerabilities.

Unsolicited connections

Unsolicited outgoing connections from an invoked instance to an external address may be an indication for a significant security problem. For example, such connections could be the evidence of some kind of backdoor, or the sign for a malware infection. Outgoing connections that are more stealthy may also be used to gather information about the AMI's usage, and collect IP target addresses that can then be used to attack the instance

through another built-in backdoor.

In our experiments, we observed several images that opened connections to various web applications within and outside of Amazon EC2. These connections were apparently checking for the availability of new versions of the installed software. Unfortunately, it is almost impossible to distinguish between a legitimate connection (e.g., a software update) and a connection that is used for malicious purposes.

Nevertheless, we noticed a number of suspicious connections on several Linux images: The Linux operating system comes with a service called *syslog* for recording various events generated by the system (e.g., the login and logout of users, the connection of hardware devices, or incoming requests toward the web server [11]). Standard installations record these kinds of events in files usually stored under the `/var/log` directory and only users with administrative privileges are allowed to access the logs generated by the syslog service. In our tests, we discovered two AMIs in which the syslog daemon was configured to send the log messages to a remote host, out of the control of the user instantiating the image. It is clear that this setup constitutes a privacy breach, since confidential information, normally stored locally under a protected directory, were sent out to a third party machine.

Backdoors and Leftover Credentials

The primary mechanism to connect to a Linux machine remotely is through the `ssh` service. When a user rents an AMI, she is required to provide the public part of the her ssh key that it is then stored by Amazon in the `authorized_keys` in the home directory. The first problem with this process is that a user who is malicious and does not remove her public key from the image before making it public could login into any running instance of the AMI. The existence of these kinds of potential backdoors is known by Amazon since the beginning of April 2011 [113].

A second problem is related to the fact that the ssh server may also permit password-based authentication, thus providing a similar backdoor functionality if the AMI provider does not remove her passwords from the machine. In addition, while leftover ssh keys only allow people with the corresponding private key (normally the AMI image creator), to obtain access to the instance, passwords provide a larger attack vector: Anybody can extract the password hashes from an AMI, and try to crack them using a password-cracking tool (e.g., John the Ripper [25]).

In other words, ssh keys were probably left on the images by mistake, and without a malicious intent. The same applies to password, with the difference that passwords can also be exploited by third parties, transforming a mistake in a serious security problem.

During our tests, we gathered these leftover credentials, and performed an analysis to verify if a remote login would be possible by checking the account information in `/etc/passwd` and `/etc/shadow`, as well as the remote access configuration of OpenSSH.

The results, summarized in Table 7.4, show that the problem of leftover credentials is significant: 21.8% of the scanned AMIs contain leftover credentials that would allow a third-party to remotely login into the machine. The table also reports the type of credentials, and lists how many of these would grant superuser privileges (either via root, sudo or su with a password).

7.4.3 Privacy Risks

The sharing of AMIs not only bears risks for the customers who rent them, but also for the user who creates and distributes the image. In fact, if the image contains sensitive

	US East	US West	Europe	Asia	Total
AMIs with leftover credentials	34.75%	8.35%	9.80%	6.32%	21.80%
With password	67	10	22	2	101
With SSH keys	794	53	86	32	965
With both	71	6	9	4	90
Superuser privileges	783	57	105	26	971
User privileges	149	12	12	12	185

Tab. 7.4: Left credentials per AMI

information, this would be available to anybody who is renting the AMI. For example, an attacker can gather SSH private keys to break into other machines, or use forgotten Amazon Web Services (AWS) keys to start instances at the image provider's cost. In addition, other data sources such as the browser and shell histories, or the database of last login attempts can be used to identify and de-anonymize the AMI's creator.

Private keys

We developed a number of tests to search the AMIs' file-system for typical filenames used to store keys (e.g., `id_dsa` and `id_rsa` for SSH keys, and `pk-[0-9A-Z]*.pem` and `cert-[0-9A-Z]*.pem` for AWS API keys). Our system was able to identify 67 Amazon API keys, and 56 private SSH keys that were forgotten. The API keys are not password protected and, therefore, can immediately be used to start images on the cloud at the expense of the key's owner. Even though it is good security practice to protect SSH keys with a passphrase, 54 out of 56 keys were not protected. Thus, these keys are easily reusable by anybody who has access to them. Although some of the keys may have been generated specifically to install and configure the AMI, it would not be a surprising discovery if some users reused their own personal key, or use the key on the AMI to access other hosts, or Amazon images.

By consulting the last login attempts (i.e., by `lastlog` or `last` commands), an attacker can easily retrieve IP addresses that likely belong to other machines owned by the same person. Our analysis showed that 22% of the analyzed AMIs contain information in at least one of the `last` login databases. The `lastb` database contains the failed login attempts, and therefore, can also be very helpful in retrieving user account passwords since passwords that are mistyped or typed too early often appear as user names in this database. There were 187 AMIs that contained a total of 66,601 entries in their `lastb` databases. Note that host names gathered from the shell history, the SSH user configuration, and the SSH server connection logs can also provide useful clues to an attacker.

Browser History

Nine AMIs contained a Firefox history file (two concerning root and seven concerning a normal user). Note that because of ethical concerns, we did not manually inspect the contents of the browser history. Rather, we used scripts to check which domains had been contacted. From the automated analysis of the history file, we discovered that one machine was used by a person to log into the portal of a Fortune 500 company. The same user then logged into his/her personal Google email account. Combining this kind of information,

Finding	# Credentials	# For Image	# For Remote Machines
Amazon RDS	4	0	4
Dynamic DNS	1	0	1
Database Monitoring	7	6	1
Mysql	58	45	13
Web Applications	3	2	1
VNC	1	1	0
Total	74	54	20

TAB. 7.5: Credentials in history files

history files can easily be used to de-anonymize, and reveal information about the image's creator.

Shell History

When we tested the AMI using our test suite, we inspected common shell history files (e.g. `~/.history`, `~/.bash_history`, `~/.sh_history`) that were left on the image when it was created. We discovered that 612 AMIs (i.e., 11.54% of the total) contained at least one single history file. We found a total of 869 files that stored interesting information (471 for root and 398 for generic users), and that contained 158,354 lines of command history. In these logs, we identified 74 different authentication credentials that were specified in the command line, and consequently recorded on file (ref. Table 7.5).

For example, the standard MySQL client allows to specify the password from the command line using the `-p` flag. A similar scenario occurs when sensitive information, such as a password or a credit card number, is transferred to a web application using an HTTP GET request. GET requests, contrary to POST submissions, are stored on the web server's logs. The credentials we discovered belong to two categories: local and remote.

The credentials in the *image* group grant an attacker access to a service/resource that is hosted on the AMI. In contrast, remote credentials enable the access to a remote target. For example, we identified remote credentials that can be used to modify (and access) the domain name information of a dynamic DNS account. A malicious user that obtains a DNS management password can easily change the DNS configuration, and redirect the traffic of the original host to his own machines. In addition, we discovered four credentials for the Amazon Relational Database Service (RDS) [134] – a web service to set up, operate, and scale a relational database in the Amazon cloud. We also found credentials for local and remote web applications for different uses (e.g. Evergreen, GlassFish, and Vertica) and for a database performance monitoring service. One machine was configured with VNC, and its password was specified from the command line. Finally, we were able to collect 13 credentials for MySQL that were used in the authentication of remote databases.

Recovery of deleted files

In the previous sections, we discussed the types of sensitive information that may be forgotten by the image provider. Unfortunately, the simple solution of deleting this information before making the image publicly available is not satisfactory from a security point of view.

In many file systems, when a user deletes a file, the space occupied by the file is marked as free, but the content of the file physically remains on the media (e.g. the hard-disk). The contents of the deleted file are definitely lost only when this marked space is overwritten by another file. Utilities such as `shred`, `wipe`, `sfill`, `scrub` and `zerofree` make data recovery difficult either by overwriting the file's contents before the file is actually unlinked, or by overwriting all the corresponding empty blocks in the filesystem (i.e., secure deletion or wiping). When these security mechanisms are not used, it is possible to use tools (e.g., `extundelete` and `Winundelete`) to attempt to recover previously deleted files.

In the context of Amazon EC2, in order to publish a custom image on the Amazon Cloud, a user has to prepare her image using a predefined procedure called *bundling*. This procedure involves three main steps: Create an image from a loopback device or a mounted filesystem, compress and encrypt the image, and finally, split it into manageable parts so that it can be uploaded to the S3 storage.

The first step of this procedure changes across different bundling methods adopted by the user (ref. Table 7.6). For example, the `ec2-bundle-image` method is used to bundle an image that was prepared in a loopback file. In this case, the tool transfers the data to the image using a block level operation (e.g. similar to the `dd` utility). In contrast, if the user wishes to bundle a running system, she can choose the `ec2-bundle-vol` tool that creates the image by recursively copying files from the live filesystem (e.g., using `rsync`). In this case, the bundle system works at the file level.

Any filesystem image created with a block-level tool will also contain blocks marked as free, and thus may contain parts of deleted files. As a result, out of the four bundling methods provided by Amazon, three are prone to a file undeletion attack.

To show that our concerns have practical security implications, we randomly selected 1,100 Linux AMIs in four different regions (US East/West, Europe and Asia). We then used the `extundelete` data recovery utility [17] to analyze the filesystem, and recover the contents of all deleted files. In our experiment, we were able to recover files for 98% of the AMIs (from a minimum of 6 to a maximum of more than 40,000 files per AMI). In total, we were able to retrieve 28.3GB of data (i.e., an average of 24MB per AMI), as shown in Table 7.8.

We collected statistics on the type (Table 7.7) of the undeleted files by remotely running the `file` command. Note that in order to protect the privacy of Amazon users, we did not access the contents of the recovered data, and we also did not transfer this data out of the vulnerable AMI. The table shows a breakdown of the types of sensitive data we were able to retrieve (e.g., PDFs, Office documents, private keys). Again, note that the Amazon AWS keys are not password-protected. That is, an attacker that gains access to these keys is then able to instantiate Amazon resources (e.g. S3 and AWS services) at the victim's expense (i.e., the costs are charged to the victim's credit card).

In our analysis, we verified if the same problem exists for Windows AMIs. We analyzed some images using the `WinUndelete` tool [128], and were able to recover deleted files in all cases. Interestingly, we were also able to undelete 8,996 files from an official image that was published by Amazon AWS itself.

7.5 Matching AMIs to Running Instances

In the previous sections, we presented a number of experiments we conducted to assess the security and privacy issues involved in the release and use of public AMIs. The results of our experiments showed that a large number of factors must be considered when making

Method	Level	Vulnerable
`ec2-bundle-vol`	File-System	No
`ec2-bundle-image`	Block	Yes
From AMI snapshot	Block	Yes
From VMWare	Block	Yes

Tab. 7.6: Tested Bundle Methods

Type	#
Home files (`/home, /root`)	33,011
Images (min. 800x600)	1,085
Microsoft Office documents	336
Amazon AWS certificates and access keys	293
SSH private keys	232
PGP/GPG private keys	151
PDF documents	141
Password file (`/etc/shadow`)	106

Tab. 7.7: Recovered data from deleted files

sure that a virtual machine image can be operated securely (e.g., services must be patched and information must be sanitized).

A number of the issues we described in the previous sections could potentially be exploited by an attacker (or a malicious image provider) to obtain unauthorized remote access to any running machine that adopted a certain vulnerable AMI. However, finding the right target is not necessarily an easy task.

For example, suppose that a malicious provider distributes an image containing his own ssh key, so that he can later login into the virtual machines as root. Unfortunately, unless he also adds some kind of mechanism to "call back home" and notify him of the IP address of every new instance, he would have to brute force all the Amazon IP space to try to find a running machine on which he can use his credentials. To avoid this problem, in this section we explore the feasibility of automatically matching a running instance back to the corresponding AMI.

"Static" testing of an images as we have done can only indicate the existence of a *potential* problem. However, it does not prove that the problem indeed exists in the running instances. Clearly, it is also possible that (although very unlikely) that all users were already aware of the risks we presented in our study and, therefore, took all the required steps to update and secure the machines they rented from Amazon.

We started our experiment by querying different public IP registries (ARIN, RIPE, and LAPNIC) to obtained a list of all IPs belonging to the Amazon EC2 service for the regions US-East, US-West, Europe and Singapore. The result was a set of sub-networks that comprises 653,401 distinct IPs that are potentially associated with running images.

For each IP, we queried the status of thirty commonly used ports (i.e., using the `NMap` tool), and compared the results with the information extracted from the AMI analysis. We only queried a limited number of ports because our aim was to be as non-intrusive as possible (i.e., see Section 7.6 for a detailed discussion of ethical considerations, precautions, and collaboration with Amazon). For the same reason, we configured NMap to only send a few packets per second to prevent any flooding, or denial of service effect.

	Minimum	Average	Maximum	Total
Files (#)	6	480	40,038	564,513
Directories (#)	8	66	2,166	78,412
Size	124KB	24MB	2.4GB	28.3GB

TAB. 7.8: Statistics of the recovered data

Technique	Instances	Perfect Match	Set of 10 Candidates	Set of 50 Candidates
SSH	130,580	2,149 (1.65%)	8,869 (6.79%)	11,762 (9.01%)
Services	203,563	7,017 (3.45%)	30,345 (14.91%)	63,512 (31.20%)
Web	125,554	5,548 (4.42%)	31,651 (25.21%)	54,918 (43.74%)

TAB. 7.9: Discovered Instances

Our scan detected 233,228 running instances. This number may not reflect the exact number of instances that were indeed running. That is, there may have been virtual machines that might have been blocking all ports.

We adopted three different approaches to match and map a running instance to a set of possible AMIs. The three methods are based on the comparison of the SSH keys, versions of services, and web-application signatures.

Table 7.9 depicts the results obtained by applying the three techniques. The first column shows the number of running instances to which a certain technique could be applied (e.g., the number of instances where we were able to grab the SSH banner). The last two columns report the number of running machines for which a certain matching approach was able to reduce the set of candidate AMIs to either 10 or 50 per matched instance. Since 50 possibilities is a number that is small enough to be easily brute-forced manually, we can conclude that it is possible to identify the AMI used in more than half of the running machines.

SSH matching Every SSH server has a host key that is used to identify itself. The public part of this key is used to verify the authenticity of the server. Therefore, this key is disclosed to the clients. In the EC2, the host key of an image needs to be regenerated upon instantiation of an AMI for two reasons: First, a host key that is shared among several machines makes these servers vulnerable to man-in-the-middle attacks (i.e., especially when the private host key is freely accessible). Second, an unaltered host key can serve as an identifier for the AMI, and may thus convey sensitive information about the software that is used in the instance.

This key regeneration operation is normally performed by the `cloud-init` script provided by Amazon. The script should normally be invoked at startup when the image is first booted. However, if the image provider either forgets or intentionally decides not to add the script to his AMI, this important initialization procedure is not performed. In such cases, it is very easy for an attacker to match the SSH keys extracted from the AMIs with the ones obtained from a simple `NMap` scan. As reported in Table 7.9, we were able to precisely identify over 2,100 AMI instances by using this method.

Service matching In the cases where the ssh-based identification failed, we attempted to compare the banners captured by `NMap` with the information extracted from the services

installed on the AMIs. In particular, we compared the service name, the service version, and (optionally) the additional information fields returned by the thirty common ports we scanned in our experiment.

The service-matching approach is not as precise as the ssh-based identification. Hence, it may produce false positives if the user has modified the running services. However, since most services installed on the AMIs were old and out of date, it is very unlikely that new services (or updated ones) will match the same banners as the one extracted from the AMIs. Therefore, a service update will likely decrease the matching rate, but unlikely generate false positives. The fact that over 7,000 machines were identified using this method seems to support the hypothesis that a large number of users often forget to update the installed software after they rent an AMI.

Web matching For our last AMI matching approach, we first collected web information from all the instances that had ports 80 and 443 (i.e., web ports) open. We then compared this information with the data we collected during the scan of the Amazon AMIs.

In the first phase, we used the WhatWeb tool [9] to extract the name and version of the installed web server, the configuration details (e.g., the OpenSSL version), and the installed interpreters (e.g., PHP, and JSP). In addition, we also attempted to detect the name and version of the web applications installed in the root document of the web server by using WhatWeb's plugins for the detection of over 900 popular web software.

In the second phase, we compared this information to the scanned AMIs, and checked for those machines that had the same web server, the same configuration, and the same versions of the language interpreters. We considered only those AMIs that matched these three criteria. Since different installations of the same operating system distribution likely share this information, we then further reduced the size of the candidate set by checking two additional items: The title of the web application (provided by the `<title>` tag), and the source files (e.g., the `index.html` document) installed in the root directory. We attempted to match the application's source code with the application name detected by the WhatWeb tool.

The last row of Table 7.9 shows that we were able to identify more than 5,000 machines by using this technique.

From our experiments, it seems that the web-based matching is, out of the three, the most efficient technique to match online instances with AMIs. Indeed, we perfectly matched 4.42% of instances and we restricted the choice to 50 AMIs for more than 40% of the instances. However, the same considerations about false positives we mentioned for the *service matching* also applies for this test.

7.6 Ethical Considerations and Amazon's Feedback

Real-world experiments involving cloud services may be considered an ethically sensitive area. Clearly, one question that arises is if it is ethically acceptable and justifiable to conduct experiments on a real cloud service. During all the experiments, we took into account the privacy of the users, the sensitivity of the data that was analyzed, and the availability of Amazon's services. Note that the first part of our experiments was conducted by automated tools running inside virtual machines we rented explicitly for our study. We did not use any sensitive data extracted from the AMI, or interact with any other server during this test. In addition, we promptly notified Amazon of any problem we found

during our experiments.

Amazon has a dedicated group dealing with the security issues of their cloud computing infrastructure: the AWS (Amazon Web Services) Security Team. We first contacted them on May 19th 2011, and provided information about the credentials that were inadvertently left on public AMIs. Amazon immediately verified and acknowledged the problem, and contacted all the affected customers as summarized by a public bulletin released on June 4th [124]. In cases where the affected customer could not be reached immediately, the security team acted on behalf of the user, and changed the status of the vulnerable AMI to private to prevent further exposure of the customer's personal credentials.

We also communicated to the AWS Security team our concerns regarding the privacy issues related to publishing of public AMIs (e.g., history files, remote logging, and leftover private keys described in Section 7.4.3). The security team reacted quickly, and realeased a tutorial [123] within five days to help customers share public images in a secure manner. Finally, we contacted again Amazon on June 24th about the possibility of recovering deleted data from public Amazon AMIs. To fix the problem, we provided them some of the countermeasures we discussed in Section 7.4.3. Their team immediately reported the issue internally and was grateful of the issue we reported to attention. By the time of writing, Amazon has already verified all the public AMIs where we have been able to recover data, and has moved on to check the status of all other public AMIs. The AWS security team is also working on providing a solution to prevent the recovery of private documents by undeletion.

The second part of our experiments included the use of a port scanner to scan running images. Even though port scanning has not been considered to be illegal per se (e.g., such as in the legal ruling in [6]), this activity may be considered an ethically sensitive issue. However, given the limited number of ports scanned (i.e, 30) and the very low volume of packets per second that we generated, we believe that our activity could not have caused any damage to the integrity and availability of Amazon's network, or the images running on it.

7.7 Summary

In this chapter, we explored the general security risks associated with virtual server images from the public catalogs of cloud service providers. We investigated in detail the security problems of public images that are available on the Amazon EC2 service by conducting measuremnts on a large-scale. Our findings demonstrate that both users and providers of public AMIs may be vulnerable to security risks such as unauthorized access, malware infections, and the loss of sensitive information. The Amazon Web Services Security Team has acknowledged our findings, and has already taken steps to address the security risks we have identified. We hope that the results of this study will be useful for other cloud service providers who offer similar services.

Chapitre 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we advanced the state of the art in large-scale testing and measurement of Internet threats. We started by identifying three classes of problems that affect Internet systems which have lately experienced a fast surge in popularity (e.g., web applications and cloud computing services). We researched into the security of web applications and we selected two novel classes of problems called Clickjacking and HTTP Parameter Pollution. We then explored the risks associated with the use of cloud computing services such as Amazon's EC2. All our studies have been conducted on a large-scale - i.e. over thousands to a million of possible vulnerable targets - and they provide a first estimation of the prevalence and relevance of these novel Internet threats. We proposed new methodologies and solutions for analyzing online applications and services on a large scale in an efficient manner.

We introduced *ClickIDS*, a system to analyze web pages for the presence of clickjacking attempts. Although Clickjacking has been widely discussed on web forums and blogs, it is unclear to what extent it is being used by attackers in the wild, and how significant the attack is for the security of Internet users. ClickIDS relies on a real browser to load and render a web page, and on a window-centric component to simulate the behavior of a human user that interacts with the content of the page. Using ClickIDS, we conducted an empirical study aimed at estimating the prevalence of Clickjacking attacks over the Internet by automatically testing a million web pages that are likely to contain malicious content and to be visited by Internet users. In our experiment we have identified only two instances of clickjacking attacks - i.e. one used for click fraud and the other for message spamming - and this suggests that clickjacking is not the preferred attack vector adopted by attackers. We published this research in the proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (AsiaCCS 2010).

We then focused on a second recent threat for the Web called HTTP Parameter Pollution (HPP). Since its first announcement in 2009, HPP had not received much attention. We proposed the first automated approach for the discovery of HPP vulnerabilities that we implemented in a tool called *PAPAS (PArameter Pollution Analysis System)* and made available online as a service. We conducted a measurement study to assess the the prevalence of HPP vulnerabilities on the Internet today by analyzing more than 5,000 popular websites with PAPAS. We showed that a third of these websites contained vulnerable parameters and that 46.8% of the vulnerabilities we discovered can be exploited. The fact we were able to find vulnerabilities in many high-profile, well-known websites suggests that

many developers are not aware of the HPP problem. This research has been published, and received the Best Paper Award in the 18th Annual Network and Distributed System Security Symposium (NDSS 2011).

Finally, we looked at cloud computing as one of the Internet paradigms that has significantly increased in importance and popularity. We explored the general security risks associated with the use of virtual server images provided by cloud computing providers (e.g. Amazon). We designed and implemented a system called *SatanCloud* for evaluating the privacy and security of Linux and Windows machine images in an automated fashion. We used SatanCloud to conduct the first large-scale study over five thousands images provided by Amazon in four of its data centers (U.S. East/West, Europe and Asia). Our measurements demonstrated that both the users and the providers of server images might be vulnerable to security risks such as unauthorized access, malware infections, and loss of sensitive information. We published this research in the proceedings of the 11th edition of the Computer Security track at the 27th ACM Symposium on Applied Computing (SEC@SAC 2012).

The measurement studies that we conducted have helped in identifying, characterizing and describing three novel security threats, their relevance and prevalence on the Internet. In our research, when we succeed in obtaining valid contact information, we always informed the providers of the affected applications and services of the problems we identified. In most of the cases, they reacted in a positive manner, and acknowledge or fixed the problems.

8.2 Future Work

In the future, we hope to identify and study other recent threats such as attacks that target smart-phones, or critical infrastructures (e.g. vulnerabilities and attacks to power plants or satellite networks).

Concerning the limitations of our work, the detection of Clickjacking attempts is limited to attacks that employ clickable page elements. In general, this is not a requirement for mounting a clickjacking attack because, at least in theory, it is possible for an attacker to build a page in which a transparent IFRAME containing the target site is placed on top of an area containing normal text. In the future, we plan to extend our *ClickIDS* detection plugin to support these use cases, for example by comparing and analyzing the rendering properties of two page elements located at the same position and with a different stack order (i.e. a different `z-index`).

Our solution for the detection of HTTP Parameter Pollution vulnerabilities has limitations as well. First, *PAPAS* does not support the crawling of links embedded in active content such as Flash, and therefore, is not able to visit websites that rely on active content technologies to navigate among the pages. A future work is to extend the crawling capabilities of black-box web scanners, such as PAPAS, to support these websites. Second, currently, PAPAS focuses only on HPP vulnerabilities that can be exploited via client-side attacks (e.g., analogous to reflected XSS attacks) where the user needs to click on a link prepared by the attacker. Some HPP vulnerabilities can also be used to exploit server side components (when the malicious parameter value is not included in a link but it is decoded and passed to a backend component). However, testing for server side attacks is more difficult than testing for client-side attacks as comparing requests and answers is not sufficient (i.e., similar to the difficulty of detecting stored SQL-injection vulnerabilities via black-box scanning). We leave the detection of server-side attacks to future work.

Finally, after we concluded our threat measurement work, we were asked several times to repeat the experiments - for example on a yearly basis - to assess the evolution of the problems over time. We leave such studies to future work.

Bibliographie

[1] Alexa top sites. http://www.alexa.com/topsites.

[2] Malware domain blocklist. http://www.malwaredomains.com/.

[3] Myspace. http://www.myspace.com.

[4] Nikto. http://www.cirt.net/nikto2.

[5] Phishtank: Join the fight against phishing. http://www.phishtank.com/.

[6] Security focus: Port scans legal, judge says. http://www.securityfocus.com/news/126.

[7] Selenium web application testing system. http://seleniumhq.org/.

[8] Watir automated webbrowsers. http://wtr.rubyforge.org/.

[9] Whatweb webapp scanner. http://www.morningstarsecurity.com/research/whatweb.

[10] xdotool. http://www.semicomplete.com/projects/xdotool/.

[11] Syslog-ng, 2000-2011. http://www.balabit.com/network-security/syslog-ng/.

[12] Pc users warned of infected web sites, 2004. http://www.washingtonpost.com/wp-dyn/articles/A5524-2004Jun25.html.

[13] New openssl packages fix predictable random number generator, 2008. http://lists.debian.org/debian-security-announce/2008/msg00152.html.

[14] http://www.blogger.com, 2009.

[15] Cloud computing risk assessment, November 2009. http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment/at_download/fullReport.

[16] Security guidance for critical areas of focus in cloud computing v2.1, December 2009. https://cloudsecurityalliance.org/csaguide.pdf.

[17] Extundelete linux data recovery tool, 2010. http://extundelete.sourceforge.net/.

[18] Amazon elastic compute cloud, May 2011. http://aws.amazon.com/security.

[19] Amazon elastic compute cloud (amazon ec2), May 2011. http://aws.amazon.com/ec2/.

[20] Clamav, July 2011. http://www.clamav.net/.

[21] Cloud computing, cloud hosting and online storage by rackspace hosting, May 2011. http://www.rackspace.com/cloud/.

[22] Enterprise cloud computing from terremark, May 2011. http://www.terremark.com/services/cloudcomputing.aspx.

[23] Guidelines on security and privacy in public cloud computing, draft special publication 800-144, January 2011. csrc.nist.gov/publications/drafts/.../Draft-SP-800-144_cloud-computing.pdf.

[24] Ibm smartcloud, May 2011. http://www.ibm.com/cloud-computing/us/en/#!iaas.

[25] John the ripper unix password cracker, April 2011. http://www.openwall.com/john/.

[26] Joyent smartdatacenter, May 2011. http://www.joyent.com/services/smartdatacenter-services/.

[27] Reminder about safely sharing and using public amis, Jun 2011. http://aws.amazon.com/security/security-bulletins/reminder-about-safely-sharing-and-using-public-amis/.

[28] Acunetix. Acunetix Web Vulnerability Scanner. http://www.acunetix.com/, 2008.

[29] Inc. Alexa Internet. Alexa - Top Sites by Category: Top. http://www.alexa.com/topsites/category.

[30] Alexa Internet, Inc. Alexa - top sites by category. http://www.alexa.com/topsites/category/Top/, 2009.

[31] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[32] P. Barford, A. Bestavros, J. Byers, and M. Crovella. On the marginal utility of network topology measurements. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 5–17. ACM, 2001.

[33] Jeff Barr. Amazon usage statistics. http://www.storagenewsletter.com/news/cloud/amazon-s3-449-billion-objects.

[34] S. Batsakis, E.G.M. Petrakis, and E. Milios. Improving the performance of focused web crawlers. *Data & Knowledge Engineering*, 68(10):1001–1013, 2009.

[35] Jason Bau, Elie Burzstein, Divij Gupta, and John. C. Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proceedings of IEEE Security and Privacy*, May 2010.

[36] T. Berners-Lee, R. Fielding, and L. Masinter. Rfc 3986, uniform resource identifier (uri): Generic syntax, 2005. http://rfc.net/rfc3986.html.

[37] S. Bleikertz, M. Schunter, C.W. Probst, D. Pendarakis, and K. Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 93–102. ACM, 2010.

[38] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[39] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.R. Sadeghi, and T. Schneider. Amazonia: When elasticity snaps back. 2011.

[40] Burp Spider. Web Application Security. http://portswigger.net/spider/, 2008.

[41] Cenzic. Cenzic Hailstormr. http://www.cenzic.com/, 2010.

[42] CERN. W3 software release into public domain. http://tenyears-www.web.cern.ch/tenyears-www/Welcome.html.

[43] Y. Chen. *A novel hybrid focused crawling algorithm to build domain-specific collections*. PhD thesis, Citeseer, 2007.

[44] J. Cho. *Crawling the Web: Discovery and maintenance of large-scale Web data*. PhD thesis, Citeseer, 2001.

[45] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the 26th international conference on very large data bases*, pages 200–209. Citeseer, 2000.

[46] Steve Christey and Robert A. Martin. Vulnerability Type Distributions in CVE, May 2007. http://cwe.mitre.org/documents/vuln-trends/index.html.

[47] M. Christodorescu, R. Sailer, D.L. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 97–102. ACM, 2009.

[48] Internet System Consortium. Internet host count history. http://www.isc.org/solutions/survey/history.

[49] Maurice de Kunder. World wide web size. http://www.worldwidewebsize.com.

[50] Valgrind Developers. Valgrind. http://valgrind.org/.

[51] Stefano di Paola and Luca Carettoni. Client side Http Parameter Pollution - Yahoo! Classic Mail Video Poc, May 2009. http://blog.mindedsecurity.com/2009/05/client-side-http-parameter-pollution.html.

[52] A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, 2010.

[53] Manuel Egele, Martin Szydlowski, Engin Kirda, and Christopher Kruegel. Using static program analysis to aid intrusion detection. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006, Proceedings*, pages 17–36, 2006.

[54] D. Fetterly, M. Manasse, M. Najork, and J.L. Wiener. A large-scale study of the evolution of web pages. *Software: Practice and Experience*, 34(2):213–237, 2004.

[55] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999. `http://www.rfc.net/rfc2616.html`.

[56] Bernardo Damele A. G. and Miroslav Stampar. sqlmap. `http://sqlmap.sourceforge.net`.

[57] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems-Volume 10*, pages 20–20. USENIX Association, 2005.

[58] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International, 1994.

[59] Nova Scotia's Electric Gleaner. Cost of hard drive storage space. `http://ns1758.ca/winch/winchest.html`.

[60] R. Glott, E. Husmann, A.R. Sadeghi, and M. Schunter. Trustworthy clouds underpinning the future internet. *The Future Internet*, pages 209–221, 2011.

[61] Jan Goebel, Thorsten Holz, and Carsten Willems. Measurement and analysis of autonomous spreading malware in a university environment. In *Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '07, pages 109–128, Berlin, Heidelberg, 2007. Springer-Verlag.

[62] S. Gordeychik, J. Grossman, M. Khera, M. Lantinga, C. Wysopal, C. Eng, S. Shah, L. Lee, C. Murray, and D. Evteev. Web application security statistics project. *The Web Application Security Consortium*.

[63] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1371–1380. IEEE, 2000.

[64] J. Grossman. Whitehat website security statistics report. *WhiteHat Security*, 2011.

[65] Robert Hansen. Clickjacking details. `http://ha.ckers.org/blog/20081007/clickjacking-details/`, 2008.

[66] Robert Hansen and Jeremiah Grossman. Clickjacking. `http://www.sectheory.com/clickjacking.htm`, 09 2008.

[67] Norman Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.

[68] Marco Slaviero Haroon Meer, Nick Arvanitis. Clobbering the cloud, part 4 of 5, 2009. http://www.sensepost.com/labs/conferences/clobbering_the_cloud/amazon.

[69] HP Fortify. HP WebInspect. https://www.fortify.com/products/web_inspect.html.

[70] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2003. ACM.

[71] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[72] IBM Software. IBM Rational AppScan. www.ibm.com/software/awdtools/appscan/.

[73] A.S. Ibrahim, J. Hamlyn-Harris, and J. Grundy. Emerging security challenges of cloud virtual infrastructure. In *Proceedings of the APSEC 2010 Cloud Workshop*, November 2010.

[74] DropBox Inc. Where are my files stored? https://www.dropbox.com/help/7.

[75] Insecure.org. NMap Network Scanner. http://www.insecure.org/nmap/, 2010.

[76] SANS Institute. Top Cyber Security Risks, September 2009. http://www.sans.org/top-cyber-security-risks/summary.php.

[77] International Secure Systems Lab. http://anubis.iseclab.org, 2009.

[78] Adam Barth Collin Jackson and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *15th ACM Conference on Computer and Communications Security*, 2007.

[79] M. Jakobsson, P. Finn, and N. Johnson. Why and How to Perform Fraud Experiments. *Security & Privacy, IEEE*, 6(2):66–68, March-April 2008.

[80] Markus Jakobsson and Jacob Ratkiewicz. Designing ethical phishing experiments: a study of (ROT13) rOnl query features. In *15th International Conference on World Wide Web (WWW)*, 2006.

[81] Jeremiah Grossman. (Cancelled) Clickjacking - OWASP AppSec Talk. http://jeremiahgrossman.blogspot.com/2009/06/clickjacking-2017.html, September 2008.

[82] Jeremiah Grossman. Clickjacking 2017. http://jeremiahgrossman.blogspot.com/2009/06/clickjacking-2017.html, June 2009.

[83] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[84] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[85] Andrew Kalafut, Abhinav Acharya, and Minaxi Gupta. A study of malware in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 327–332, New York, NY, USA, 2006. ACM.

[86] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *World Wide Web Conference*, 2006.

[87] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 247–256, New York, NY, USA, 2006. ACM.

[88] Nenad Jovanovic Engin Kirda and Christopher Kruegel. Preventing Cross Site Request Forgery Attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm), Baltimore, MD*, 2006.

[89] Lavakumar Kuppan. Split and Join, Bypassing Web Application Firewalls with HTTP Parameter Pollution, June 2009. `http://andlabs.org/whitepapers/Split_and_Join.pdf`.

[90] A. Lakhina, J.W. Byers, M. Crovella, and P. Xie. Sampling biases in ip topology measurements. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 332–341. IEEE, 2003.

[91] S. Lawrence and C.L. Giles. Searching the world wide web. *Science*, 280(5360):98, 1998.

[92] H. Liu, J. Janssen, and E. Milios. Using hmm to learn user browsing patterns for focused web crawling. *Data & Knowledge Engineering*, 59(2):270–291, 2006.

[93] Insecure.Com LLC. Network Mapper (NMap), 1996-2011. `http://nmap.org/`.

[94] Michael Mahemoff. Explaining the "Don't Click" Clickjacking Tweetbomb. `http://softwareas.com/explaining-the-dont-click-clickjacking-tweetbomb`, 2 2009.

[95] Giorgio Maone. Hello ClearClick, Goodbye Clickjacking! `http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/`, 10 2008.

[96] Giorgio Maone. X-frame-options in firefox. `http://hackademix.net/2009/01/29/x-frame-options-in-firefox/`, 2009.

[97] J. Matthews, T. Garfinkel, C. Hoff, and J. Wheeler. Virtual machine contracts for datacenter and cloud computing environments. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 25–30. ACM, 2009.

[98] P. Mell, K. Scarfone, and S. Romanosky. A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-Forum of Incident Response and Security Teams*, 2007.

[99] F. Menczer, G. Pant, and P. Srinivasan. Topical web crawlers: Evaluating adaptive algorithms. *ACM Transactions on Internet Technology (TOIT)*, 4(4):378–419, 2004.

[100] Microsoft. IE8 Clickjacking Defense. `http://blogs.msdn.com/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx`, 01 2009.

[101] Microsoft Corporation. Security attribute (frame, iframe, htmldocument constructor). `http://msdn.microsoft.com/en-us/library/ms534622(VS.85).aspx`.

[102] MITRE. Common vulnerabilities and exposures (cve). `http://cve.mitre.org/`.

[103] Dave Morgan. Storage-n-harddrives. where are we and where are we going? `http://semiaccurate.com/2010/07/24/storage-n-harddrives-where-are-we-and-where-are-we-going/`.

[104] A. Moshchuk, T. Bragin, S.D. Gribble, and H.M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium*, pages 17–33. Citeseer, 2006.

[105] Mozilla Foundation. `https://bugzilla.mozilla.org/show_bug.cgi?id=154957`, 2002.

[106] S. Neuhaus and T. Zimmermann. Security trend analysis with cve topic models. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 111–120. IEEE, 2010.

[107] J. Nielsen. The same origin policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

[108] Computer Security Division of National Institute of Standards and Technology. National vulnerability database version 2.2. `http://nvd.nist.gov/`.

[109] OWASP. Owasp top10 project. `https://www.owasp.org/index.php/Top_10_2010`.

[110] OWASP AppSec Europe 2009. *HTTP Parameter Pollution*, May 2009. `http://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf`.

[111] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large scale internet measurement. *Communications Magazine, IEEE*, 36(8):48–54, 1998.

[112] N. Provos, P. Mavrommatis, M.A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15. USENIX Association, 2008.

[113] Alen Puzic. Cloud security: Amazon's ec2 serves up 'certified pre-owned' server images, April 2011. `http://dvlabs.tippingpoint.com/blog/2011/04/11/cloud-security-amazons-ec2-serves-up-certified-pre-owned-server-images`.

[114] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proceedings of the International Conference on Very Large Data Bases*, pages 129–138. Citeseer, 2001.

[115] J.W. Ratcliff and D. Metzener. Pattern matching: The gestalt approach. *Dr. Dobb's Journal*, 7:46, 1988.

[116] Dark Reading. CSRF Flaws Found on Major Websites: Princeton University researchers reveal four sites with cross-site request forgery flaws and unveil tools to protect against these attacks, 2008. http://www.darkreading.com/security/app-security/showArticle.jhtml?articleID=211201247.

[117] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[118] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, pages 52–78. Springer-Verlag, 2011.

[119] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[120] Mark Russinovich. Psexec, 2009. http://technet.microsoft.com/en-us/sysinternals/bb897553.

[121] Stefan Saroiu, Steven D. Gribble, and Henry M. Levy. Measurement and analysis of spywave in a university environment. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 11–11, Berkeley, CA, USA, 2004. USENIX Association.

[122] T. Scholte, D. Balzarotti, and E. Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications.

[123] Amazon AWS Security. How to share and use public amis in a secure manner, June 2011. http://aws.amazon.com/articles/0155828273219400.

[124] Amazon AWS Security. Reminder about safely sharing and using public amis, June 2011. http://aws.amazon.com/security/security-bulletins/reminder-about-safely-sharing-and-using-public-amis/.

[125] Tenable Network Security. Nessus vulnerability scanner, 2002-2011. http://www.tenable.com/products/nessus.

[126] IBM Managed Security Services. Trend and risk report, March 2011.

[127] Y. Shavitt and E. Shir. Dimes: Let the internet measure itself. *ACM SIGCOMM Computer Communication Review*, 35(5):71–74, 2005.

[128] WinRecovery Software. Winundelete windows data recovery tool, 2001-2011. http://www.winundelete.com/.

[129] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Temu. http://bitblaze.cs.berkeley.edu/temu.html.

[130] Internet World Stats. Internet usage statistics. http://www.internetworldstats.com/stats.htm.

[131] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Symposium on Principles of Programming Languages*, 2006.

[132] Symantec. Symantec intelligence report, July 2011.

[133] H. Takabi, J.B.D. Joshi, and G. Ahn. Security and privacy challenges in cloud computing environments. *Security & Privacy, IEEE*, 8(6):24–31, 2010.

[134] Amazon AWS Team. Amazon rds, 2011. http://aws.amazon.com/rds/.

[135] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project. http://www.nessus.org/, 2010.

[136] US-CERT. CVE-2008-4503: Adobe Flash Player Clickjacking Vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4503, 10 2008.

[137] J. Varia. Architecting for the cloud: Best practices. *Amazon, Inc., Tech. Rep., Jan*, 2010.

[138] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42. Citeseer, 2007.

[139] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007.

[140] Web Application Attack and Audit Framework. http://w3af.sourceforge.net/.

[141] Y.M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys. In *Proceedings of the 2006 Network and Distributed System Security Symposium*, pages 35–49. Citeseer, 2006.

[142] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not.*, 42(6):32–41, 2007.

[143] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, New York, NY, USA, 2008. ACM.

[144] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 91–96. ACM, 2009.

[145] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.

[146] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[147] C. Yue and H. Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th international conference on World wide web*, pages 961–970. ACM, 2009.

[148] Michal Zalewski. Browser security handbook, part 2. `http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_(UI_redressing)`, 2008.

[149] Michal Zalewski. Dealing with UI redress vulnerabilities inherent to the current web. `http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2008-September/016284.html`, 09 2008.

[150] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the chinese web. *Managing Information Risk and the Economics of Security*, pages 225–244, 2009.