RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

Un Bac à Sable Juste à Temps

Nicolas Loriant, Marc Ségura-Devillechaise, Jean-Marc Menaud

École des Mines de Nantes, projet Obasco EMN/INRIA

Résumé
La complexité toujours croissante des logiciels implique un accroissement des bogues. Ces derniers restent le vecteur principal d'attaque des pirates informatiques et imposent aux éditeurs logiciels d'émettre régulièrement des mises à jour pour leurs corrections. Cependant, peu d'utilisateurs sont prêts à arrêter leurs applications et à investir le temps nécessaire pour mettre à jour leur poste de travail. Dans les faits, les correctifs sont rarement déployés, même quand ceux-ci sont critiques.
Nous proposons un atelier de déploiement permettant à un administrateur système d'appliquer à chaud, à distance et sans l'intervention des utilisateurs, des rustines de sécurité ciblées sur les quatres principaux types de bogues. Pour éviter l'arrêt des logiciels concernés, notre approche est basée sur un tisseur dynamique d'aspects, Arachne. La technique et les correctifs proposés garantissent le maintien de la cohérence de l'application.

Abstract
The ever growing software complexity entails an ever increasing number of bugs. These bugs are the hacker privileged entry points. Despite the efforts made by software publishers to reguarly broadcast updates, few users are ready to stop their applications and, or to take the time needed to update them. In practice, even critical updates are rarely deployed. Considering the main types of security holes, this paper proposes a deployment tool allowing administrators to apply patches correcting the bugs remotely on the fly on running applications without user intervention. To avoid stopping applications, our approach is based on a dynamic weaver, Arachne. The technique and the patches considered here grants the security and the coherency of the application during and after the update.

Mots-clés : mise à jour de programme à chaud, programmation par aspect, sécurité

Keywords : hot software update, aspect-oriented programming, security

---

1. Introduction

La problématique liée à la fiabilité des logiciels est un sujet récurrent. Si certains bogues informatiques n'entraînent qu'un dysfonctionnement du logiciel, déjà en soit potentiellement dramatique, d'autres en revanche permettent de déstabiliser le système dans son ensemble. Ces bogues sont alors appelés failles ou trous de sécurité. Poussés par une mise sur le marché au plus tôt, très peu d'éditeurs effectuent une réelle vérification de leurs codes, même lorsque les failles sont identifiables.
En conséquence, et pour la seule année 2003, 63% des entreprises ont subies une attaque informatique [6]. Actuellement, le « Survival Time », le temps nécesaire pour qu'une machine

connectée sans parefeu à l'internet soit infectée, est passé de 40 minutes en 2003 à 20 minutes en 2004 [8]. Cette tendance est renforcée par l'accroissement exponentiel du nombre d'alertes diffusées par le CERT. Elles n'étaient que de 417 en 2000, alors qu'elles atteignaient le nombre de 4 129 en 2003.

Actuellement, les attaques informatiques à distance sur les logiciels peuvent être classées en deux catégories : celles exploitant des vulnérabilités liées à l'architecture, à des erreurs d'implémentation ou à la configuration des logiciels, et celles exploitant les vulnérabilités de l'environnement d'exécution des programmes eux-mêmes. Les premières vont typiquement consister à exploiter un port de communication laissé ouvert dans le fichier de configuration d'un pare-feu ou utiliser une erreur d'implémentation de l'application comme l'ouverture de fichier sans vérification des droits. Les secondes vont exploiter des défauts de conceptions liés à l'environnement d'exécution et de programmation tels que la gestion de la mémoire.

Partant de la constatation que les virus et les vers sont souvent développés et déployés après la divulgation d'une faille de sécurité, nous avons proposé dans nos précédentes publications une solution systématique à base de mise à jour dynamique déduite des correctifs sources diffusés par les éditeurs de logiciels [11]. Mettre à jour juste à temps l'application, permet de s'affranchir de l'utilisation de langages de programmation, de compilateurs ou d'environnements d'exécution spécifiques et répond donc mieux aux problématiques de la sécurité des applications patrimoniales. L'aspect dynamique de la solution permet d'éviter l'arrêt de l'application, qui dans certains cas comme l'administration de grands parcs informatiques ou les serveurs à haute disponibilité, est impossible. Si notre solution, nommée Minerve, est une approche complète pour l'ensemble des failles de sécurités liées à des défauts d'implémentation de l'application ou des défauts de conceptions liés à l'environnement d'exécution et du langage de programmation, elle n'offre en revanche aucune garantie sur l'exécution du programme mis à jour. En effet, il reste très difficile de vérifier que l'état de l'application avant mise à jour soit compatible avec la mise à jour. De plus, nous ne pouvons mettre à jour juste à temps l'application que si le correctif de sécurité à été diffusé. Or, le temps entre la divulgation de la faille et l'apparition du virus est de plus en plus court [9].

La solution proposée dans cet article se veut plus prompte et sécurisante en proposant des correctifs préconçus et ciblés sur les principaux défauts de conceptions liés aux langages C et C++. De plus, elle garantit la validité de l'application avant et après mise à jour. Pour ce faire, nous n'injectons plus de code arbitraire dans l'application mais uniquement un code chargé d'observer un ou des points d'exécution particuliers et d'arrêter l'application si des conditions de validation ne sont pas vérifiées. De ce fait, l'état de l'application avant et après l'injection du code, code pouvant être considéré comme un bac à sable spécialisé, reste inchangée. De plus, l'application étant arrêtée si une faille de sécurité est exploité, nous pouvons garantir que l'application supervisée ne fait rien de plus que l'application non supervisée [3].

Nous avons implémenté trois correctifs ciblés, permettant de traiter les quatre principaux bogues : le débordement de tampon (buffer overflow), le bogue de format (format string bug), la double désallocation (double free bug) et débordement d'entier (integer overlow). Ces quatre types de bogues représentent 60% des failles répertoriées par le CERT entre 1999 et 2002 [15].

Cet article est organisé comme suit. Dans un premier temps, nous présentons les quatre types de bogues cités précédemment (section 2). Ensuite, nous décrivons l'infrastructure logicielle qui nous permet de corriger ces bogues (section 3). Enfin, la section 4 présente les premiers résultats de nos expérimentations, avant de dresser un état de l'art des travaux apparentés (section 5) et de conclure.

2. Injection et exploitation d'un code malveillant

L'objectif d'une attaque informatique consiste à prendre le contrôle de l'exécution du processus visé en maîtrisant la valeur de son compteur ordinal afin d'exécuter un code malveillant avec les droits de cette application. Ce compteur (IP Instruction Pointer ou PC Program Counter) contient l'adresse de la prochaine instruction devant être exécutée. Dans la grande majorité des architectures matérielles, ce compteur est situé dans un registre du processeur. Il est modifié par les instructions de branchement (*ex :*`call, jump`) ou d'interruption (*ex :*`int`). Puisqu'un attaquant ne peut modifier le code d'une application depuis l'extérieur, ces derniers vont essentiellement exploiter les erreurs de programmation contenues dans l'application. L'exploitation se fait en deux temps. Il faut d'abord injecter par un moyen quelconque, le plus souvent à travers une entrée demandée à l'utilisateur, le code malveillant. Puis profiter de l'erreur de programmation pour faire en sorte que le compteur de programme pointe sur le code malveillant. Atteindre cet objectif, repose donc sur deux opérations : charger un code exécutable dans le processus attaqué puis modifier une des instructions de manipulation du compteur ordinal.
La première étape de ces attaques consiste donc à écrire en mémoire un code malveillant qui soit considéré comme exécutable par le système d'exploitation. Cette opération est facilitée par des systèmes comme Windows ou Linux qui autorisent l'exécution de code dans les segments de données (pile et tas) des processus[1]. Il n'est donc pas nécessaire d'écrire le code malveillant dans le segment de code du programme.
La seconde étape consiste à détourner l'exécution du processus vers le code malveillant. Pour cela, la majorité des attaques modifie sur la pile, l'adresse de retour d'une fonction. Dès lors, une fois l'exécution de la fonction terminée, celle-ci se poursuit dans le code malveillant plutôt que dans la fonction appelante.
Les deux techniques les plus couramment utilisées, à savoir le bogue de format et le débordement de tampon, sont basées sur une même approche : utiliser une opération d'écriture dans un zone mémoire ne vérifiant pas l'adéquation entre la taille de la zone mémoire initialement allouée et la taille des données à écrire. Schématiquement, en allant écrire au delà d'un emplacement réservé, il est alors possible d'écrire n'importe quelle valeur à n'importe quel endroit de la mémoire, que ce soit le code malveillant ou la modification d'une instruction manipulant le compteur ordinal. La dernière technique utilisée, la double désallocation, est bien plus complexe et repose sur les mécanismes de gestion mémoire des différents systèmes d'exploitation. L'idée consiste à forcer la bibliothèque de gestion de la mémoire à réécrire une adresse de retour située dans la pile.
Les sections suivantes présentent de manière synthétique les trois attaques les plus courantes pour l'injection et l'exécution de code malicieux : le débordement de tampon, le bogue de format et la double désallocation. Cette section se termine par la présentation du débordement d'entier, bogue ne pouvant entraîner par lui même l'exécution d'un code malveillant, mais pouvant mettre le programme dans un état vulnérable à l'une des trois attaques précédentes.

2.1. Le débordement de tampon

Le débordement de tampon est sans nul doute la faille de sécurité la plus répandue, aussi bien dans les logiciels libres que propriétaires. En ne considérant que les alertes de sécurité pour l'année 2004 de la distribution Linux Debian - une distribution libre et populaire du système Linux - ce bogue représente à lui seul 48% des failles de sécurité. D'autres études confirment

---
[1] Cette fonctionnalité est nécessaire à certaines optimisations de compilation qui se basent sur du code automodifiant

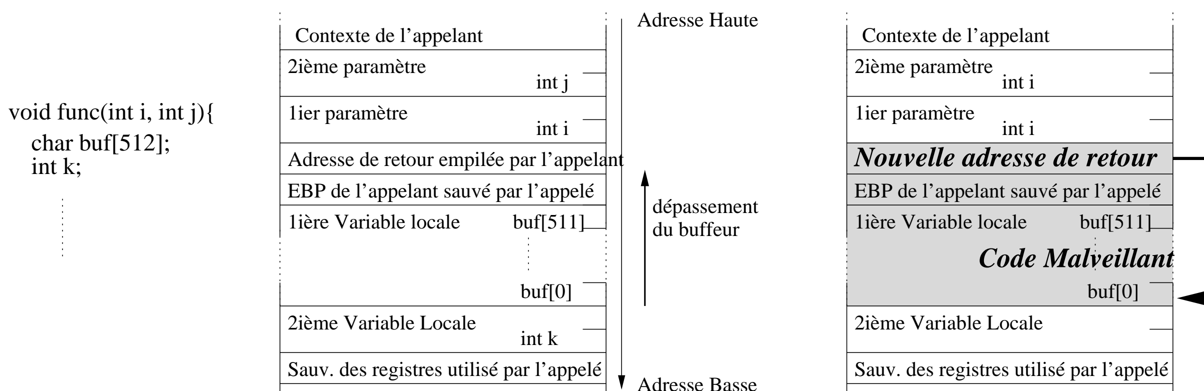

FIG. 1 – le débordement de tampon

cette proportion [15]. Ce trou de sécurité est lié au fait que dans le langage C, lors d'une écriture mémoire dans un tableau, aucune vérification n'est effectuée sur les bornes d'un tableau. De ce fait, il est possible d'écrire des données qui débordent de l'espace alloué à une variable. Dans un grand nombre de cas, si l'écriture des données est effectuée dans une zone de mémoire protégée, ceci entraîne l'arrêt du processus par le système. Dans un nombre restreint de cas, ces écritures se feront dans une zone mémoire non protégée, n'entraînant donc pas l'arrêt du processus, mais modifiant son état puis son exécution. C'est en exploitant au mieux ce cas, que le processus peut être amené à exécuter n'importe quel type de code avec les droits du processus.
La figure 1 décrit symboliquement le processus d'attaque d'un logiciel par écriture du code malveillant dans un tableau situé sur la pile, puis par écrasement de l'adresse de retour de fonction.

2.2. Le bogue de format

Le bogue de format représente 8% des bogues répertoriés, il affecte les fonctions à paramètres variables (*ex :* `printf`). Ces fonctions utilisent une chaîne de format qui décrit les paramètres qui lui sont passés. Ni le langage C ni son environnement d'exécution ne proposent un support permettant de connaître le nombre et le type des arguments fournis à une fonction à paramètres variables. Ce bogue est similaire au débordement de tampons en ce sens qu'il est possible en altérant la chaîne de format de charger un code malicieux et d'écraser l'adresse de retour de la fonction. Mais bien que ce type de failles soit très facilement repérable dans le code source d'une application, leur exploitation est plus difficile que celle des débordements de tampons. Elle requiert non seulement une connaissance pointue de l'organisation de la pile d'exécution du processus, mais également de faire attention à ne pas modifier les variables locales de l'appelant lors de l'écrasement de l'addresse de retour.

2.3. La double désallocation

2% des bogues sont des doubles désallocations. Ce bogue est une erreur de programmation exploitable par un attaquant qui intervient lorsque le programme désalloue une zone mémoire non allouée. Il est possible de tirer partie de cette erreur pour écrire à n'importe quel point du

programme une donnée arbitraire de 4 octets, et donc d'écraser par exemple l'adresse de retour d'une fonction.
Son explication est beaucoup plus complexe que pour les bogues précédents. En effet, il est nécessaire de présenter le fonctionnement de la gestion du tas. Dans la plupart des systèmes, la gestion mémoire est divisée en deux niveaux : le bas niveau, géré par le système d'exploitation, permet aux applications de réserver de grands blocs mémoire grâce aux appels systèmes `brk` et `mmap`, et le haut niveau, géré par les librairies standards qui permettent de fragmenter ces grands blocs à travers les fonctions de la famille `malloc`.
La zone mémoire allouée par le système est alors distribuée à la demande à l'application. Pour ce faire les bibliothèques de haut niveau telles que la bibliothèque C GNU (`glibc`), utilisent deux listes doublement chaînées, l'une contenant l'ensemble des blocs (libres et occupés) et l'autre uniquement les blocs libres. La particularité de ce mécanisme est que les structures de données associées à la gestion des listes sont directement inscrites dans les blocs mémoires alloués pour l'application. La figure 2 montre que les 8 premiers octets sont octroyés à la gestion de la première liste et les 8 suivants à la seconde. Lors d'une demande de réservation de mémoire par l'application (via `malloc`) la `glibc` recherche dans la liste de blocs libres un bloc convenant à la demande (de taille égale ou supérieure), retire ce bloc de la liste des bloc libres et retourne l'adresse du bloc plus huit. Ainsi seuls les 8 premiers octets de gestion de la liste des blocs sont conservés. Les octets réservés à la liste des blocs libres sont, eux, détruits. Cette optimisation permet de ne perdre que 8 octets par bloc au lieu de 16. Le bogue de double désallocation, repose à la fois sur une erreur de gestion de la liste des blocs libres et sur cette optimisation.

En effet, l'algorithme chargé de gérer la liste consiste, lors de l'insertion d'un bloc (B2), à faire pointer le pointeur `fd` (forward) du bloc précédent (B1) sur le bloc à insérer (B2) et le pointeur `bk` (backward) du bloc suivant (B3) sur le bloc B2. Si un bloc libre est relibéré, alors les pointeurs `fd` et `bk` du bloc B2 pointent sur eux-mêmes comme le décrit la figure 2. De ce fait, lors d'une allocation mémoire de taille adéquate, la `glibc` retournera correctement le bloc B2 en appliquant le mécanisme de libération qui, malheureusement laissera la liste des blocs libres inchangés. Par conséquent nous arrivons à une situation où l'application peut écrire dans un bloc considéré comme libre par la `glibc`, et donc, comme décrit précédemment, réécrire les pointeurs `fd` et `bk` de ce bloc (B2 pour notre exemple). Sans avoir encore réalisé l'exploitation de la faille, l'ensemble du mécanisme d'attaque est en place.
Pour l'exploiter, l'attaquant doit écrire dans le bloc B2 le code malveillant à exécuter. Si l'application utilise un tableau alloué dynamiquement pour stocker une requête d'un utilisateur cette opération ne pose aucun problème. Reste à déployer un mécanisme permettant de prendre le contrôle du compteur ordinal. Pour rester cohérent avec nos précédentes explications, nous utiliserons l'écrasement d'un retour de fonction mais toute autre instruction manipulant le compteur ordinal peut être utilisée.
Dans la deuxième étape, l'attaquant inscrit dans les 16 premiers octets (`fd` et `bk`) deux valeurs précises : dans `bk` l'adresse du code malicieux et dans `fd` l'adresse du code malveillant. En supposant que l'adresse d'une instruction liée à un retour d'une fonction soit `addRET`, la valeur stockée dans `fd` doit être `addRET+12`.
En effectuant un deuxième `malloc` identique en taille, la `glibc` libérera de nouveau le bloc B2 et réécrira les `fd` et `bk` des blocs précédents et suivants dans la liste. La pile contiendra alors le `bk` de B2, c'est-à-dire l'adresse du code malveillant. Au retour de la fonction le code malveillant sera alors exécuté.

2.4. Le débordement d'entier

Le débordement d'entier (2% des bogues connus) est sans aucun doute le plus simple des bogues à expliquer mais sûrement pas à exploiter. Ce bogue se déclenche lorsque le résultat d'un calcul dépasse la capacité de la variable dans laquelle il est stockée. Le résultat est alors tronqué. Après débordement, la variable contient une valeur non prévue lors de la conception. Ce cas se produit dans les applications écrites en C/C++ car ni le compilateur ni l'environnement d'exécution ne lève d'exception lorsque ce débordement se produit.
L'exploitation de ce bogue est beaucoup plus délicate car le débordement d'entier n'entraîne pas directement une faille de sécurité. C'est l'utilisation de cette valeur qui peut provoquer un comportement non prévu et/ou introduire une autre faille de sécurité comme celles décrites précédement [16, 1].
Supposons par exemple que notre programme avant copie dans une zone mémoire de 64 octets et vérifie que la taille annoncée d'un tableau est inférieure à 64. Dans cet exemple, nous utilisons une interprétation signée de cette taille (64). L'affectation à 255 de cette variable signée aura pour conséquence une interprétation erronée à la valeur -1 au moment du test (qui est bien inférieure à 64). La fonction de copie du tableau interprète la valeur non signée de la variable, dès lors des données seront écrites au delà des 64 octets réservés à notre tableau. Ici, c'est l'état du programme qui entraîne l'exploitation d'un débordement de tableau. Bien entendu les failles de ce type sont rarement aussi évidentes, les valeurs utilisées peuvent être des valeurs générées au cours de l'exécution (entrées par l'utilisateur, résultats de calculs...).

3. Une approche par réécriture de code à chaud

Cette section présente brièvement l'outil permettant d'injecter à chaud les correctifs, nommé Arachne, avant de décrire les trois correctifs utilisés pour corriger les trois failles que nous avons étudiées.

3.1. Arachne : injection dynamique des correctifs

Arachne est un tisseur dynamique d'aspect pour le langage C [13, 4]. Les correctifs génériques proposés peuvent être considérés comme des aspects car ils nécessitent une instrumentation en plusieurs points du programme. La section suivante présente les propriétés et les solutions techniques définissant les fondements d'Arachne.
Les principales propriétés d'Arachne sont d'effectuer une vérification syntaxique des aspects compilés, d'injecter le code sans arrêt ni préparation du processus, de garantir l'atomicité de la réécriture du code binaire, et de garantir la transparence et l'isolation de l'aspect injecté. Son principe de fonctionnement est de compiler un aspect sous la forme d'une bibliothèque dynamique native (DLL), de forcer le processus à charger cette DLL, et enfin de crocheter le programme, par réécriture du code binaire, en vue de lui faire exécuter le code présent dans la DLL. L'ensemble de ce mécanisme est présenté ci-après.
Sur une machine Pentium sous Linux, un appel de fonction au niveau source se traduit dans le code binaire de l'application par une instruction assembleur `CALL` suivie de l'adresse de la fonction à invoquer. Arachne désassemble le code binaire de l'application et y recherche les instructions `CALL`. Pour cela, il utilise les tables de symboles générées à la compilation pour retrouver, à partir de l'adresse manipulée par l'instruction `CALL`, le nom symbolique de la fonction invoquée. Au tissage, Arachne charge la DLL contenant les aspects et réécrit les différents `CALL` vers la fonction à remplacer par des `CALL` vers la fonction appropriée de la DLL. Un mécanisme similaire mais plus complexe permet à Arachne de remplacer les lectures/écritures

sur les variables globales.
En pratique, plusieurs problèmes compliquent la mise en œuvre du principe de réécriture décrit plus haut. En résumé, il s'agit de garantir l'atomicité des réécritures effectuées dans le code objet de l'application, d'assurer la cohérence de l'exécution, de garantir que l'ensemble des aspects de la DLL soit tissés d'une manière atomique (c'est-à-dire qu'un aspect ne puisse pas s'exécuter alors que les autres sont en train d'être tissés), de gérer les contraintes imposées par le système d'exploitation en termes de séparation des espaces d'adressage et finalement de résoudre les problèmes d'efficacité. Arachne résoud les problèmes d'atomicité des réécritures par un système de verrous tournants reposant sur la taille des instructions de sauts courts et sur la taille des accès mémoire atomiques sous Pentium. La cohérence de l'exécution est assurée par un crochet généré à la volée lors du tissage : ce crochet sauvegarde et restaure les registres nécessaires. L'utilisation d'un test sur une garde garantit que l'ensemble des aspects de la DLL sont tissés de manière atomique. La séparation imposée des espaces d'adressage par le système d'exploitation empêche un processus de modifier l'espace mémoire d'un processus tiers. La première fois qu'Arachne tisse dans un processus, il accède à son espace mémoire en utilisant les fonctions de déverminage `ptrace`, puis il crée un processus léger d'instrumentation partageant le même espace d'adressage que l'application. Arachne pilote ensuite ce processus léger d'instrumentation en utilisant une interface d'interconnexion : le tissage s'effectue ainsi globalement sans interruption de l'application. Arachne adresse les problèmes d'efficacité par des caches conçus pour n'examiner qu'une fois au plus le code binaire de l'application.

3.2. Le débordement de tampon

Comme précisé, le principe du débordement de tampon est basé sur l'écriture de données au delà de la zone réservée. Toutes les solutions proposées pour contrer ce type d'attaque consiste à vérifier que l'adresse des données inscrites correspondent bien à une adresse valide par rapport à une adresse de base et à une taille. Notre solution ne diffère pas de ce principe. Les solutions par réécriture de code analysent le code source du programme et déterminent tous les accès à une zone de mémoire. Cette opération est complexe au niveau source du fait de l'utilisation de pointeurs.
Notre approche par réécriture de code binaire permet par analyse de ce code, de déterminer plus simplement tous les accès à une zone mémoire donnée. Après analyse, le tampon à protéger est relogé dans une nouvelle page mémoire protégée en écriture. Tous les accès vers l'ancien emplacement du tampon sont réécrits pour prendre en compte le nouvel emplacement. De ce fait, tout accès en écriture sur la page provoque l'émission du signal `SIGSEGV`. Ce signal est capté, puis analysé pour déterminer l'adresse accédée. Si cette adresse est valide, la page mémoire est déprotégée, l'instruction est rejouée, puis la page mémoire reprotégée. Dans le cas contraire, le programme est arrêté.

3.3. Bogue de format

Le principe comme la solution du bogue de format restent très proches du cas du débordement de tampon. Notre solution consiste à contrôler tous les accès effectués dans la pile et de vérifier si ces accès sont autorisés ou non. En revanche, il ne s'agit plus de vérifier un tableau mais un ensemble de variables.
Contrairement à la solution du débordement de tampon, il n'est pas possible ni d'isoler les paramètres passés à une fonction du reste de la pile, ni de les protéger en écriture sans protéger toute la pile du processus. Dès lors, tous les accès réalisés par une fonction à prototype variable à ses paramètres sont remplacés dynamiquement dans le binaire du programme par l'opcode assembleur `int3`. Cette instruction, lorsqu'elle est exécutée, provoque l'émission d'un signal

SIGTRAP. Le signal est capté et contient l'adresse mémoire d'où il a été déclenché. À partir de cette information, il est possible de retrouver l'instruction d'origine qui a été écrasée et de vérifier si son exécution est valide. Si oui, l'instruction d'origine est rejouée et le programme reprend son exécution normale, sinon le programme est arrêté.

3.4. La double désallocation

Pour résoudre ce type de faille, notre solution consiste à superviser tous les appels aux fonctions d'allocation et de désallocation mémoire effectués par le programme. Ainsi, il est possible de maintenir une liste des données allouées et d'empêcher toute désallocation d'une zone non-allouée. Lorsqu'une double désallocation est détectée, le programme est arrêté.
Nous avons choisi de ne pas baser notre approche sur les listes maintenues par la librairie C. Ainsi, notre solution reste indépendante de l'algorithme de gestion de la mémoire implémenté par la librairie.

4. Évaluation

Cette section présente le surcoût induit par la protection appliquée par notre bac à sable. Pour cela, nous avons comparé les performances d'applications avec celle de leur version protégée. Notre bac à sable réécrit le code sensible qu'il remplace par un code sécurisé : ainsi le ralentissement introduit est strictement limité aux parties sensibles du code de l'application, celles qui sont vulnérables aux attaques. La première partie de cette section étudie le ralentissement relatif des parties sensibles du code. La seconde partie de cette section étudie le ralentissement global introduit par notre bac à sable dans une application réelle : le cache Web squid.

4.1. Ralentissement relatif des parties sensibles du code protégées

Mesurer le ralentissement relatif induit par la proteection sur le code sensible revient à effectuer des mesures distinctes pour chaque type de bogues. Pour le débordement de tampon, il s'agit de comparer le temps d'un accès à un tableau dans et hors du bac à sable. Dans le cas du bogue de format, compte tenu de la similitude des stratégies choisies, le ralentissement introduit par le bac à sable est comparable à celui introduit dans le cas des débordements de tampon. Dans le cas de la double désallocation, il s'agit de comparer le temps nécessaire pour réaliser une allocation (i.e. le temps nécessaire à l'exécution d'un `malloc`) et une désallocation (i.e. le temps nécessaire à l'exécution d'un `free`) dans et hors du bac à sable. Toutefois, différentes applications peuvent utiliser différents allocateurs de mémoire : par exemple le cache Web squid n'utilise pas les allocateurs de la bibliothèque C standard (i.e. `malloc` et `free`) mais ses propres allocateurs. De plus, le temps nécessaire à une allocation peut dépendre de l'état du tas. Aussi avons nous conduit notre évaluation du ralentissement introduit par notre bac à sable en remplaçant les allocateurs de mémoire par des routines vides, retournant immédiatement. Notre évaluation de la double désallocation n'indique donc pas le ralentissement typique introduit par notre bac à sable mais celui qu'il introduira au pire. Nos résultats sont consignés dans la table 1. Évidemment, les résultats montrent que le mécanisme de protection sur l'accès à une donnée induit un surcoût important par rapport au mécanisme d'accès à la donnée, fortement optimisé par le processeur. Néanmoins la section suivante montre que ce surcoût est largement amorti dans l'exécution d'une application réelle.

4.2. Ralentissement introduit par le bac à sable dans une application réelle

Lorsque le cache Web squid dans sa version 2.4.5STABLE3 analyse une requête d'authentification FTP, la taille de certains tableaux alloués dynamiquement sur le tas n'est pas vérifiée

	Avec bac à sable (cycles)	Sans bac à sable (cycles)	Ratio (%)
Accès à une donnée	$9729^{\pm 4.9\%}$	$1^{\pm 0.6\%}$	9729
gestion mémoire	$121^{\pm 0.5\%}$	$63^{\pm 1.7\%}$	3.2

TAB. 1 – Ralentissement introduit par le bac à sable dans les parties sensibles du code

Taille de l'objet (Mo)	Téléchargement à travers squid avec/sans bac à sable – Cas d'un défaut de cache: avec (s)	sans (s)	Ratio	Cas d'un objet en cache: avec (s)	sans (s)	Ratio
0.19	0.2	0.2	1	0.2	0.2	1
1.2	0.6	0.6	1	0.5	0.5	1
3.8	1.3	1.3	1	1	1	1

TAB. 2 – Ralentissement introduit par le bac à sable dans la vitesse d'exécution de squid.

et ils peuvent être amenés à déborder. Ce débordement de tampon a fait l'objet d'une alerte diffusée par le CERT [2] début 2002. En juin 2003, plusieurs mailing-lists ont rapportées publiquement comment exploiter à distance cette vulnérabilité pour prendre le contrôle du cache. Notre bac à sable permet de sécuriser le cache sans l'interrompre. Dans cette section, nous étudions l'impact de notre bac à sable sur les performances de squid. Nous avons utilisé `wget` afin de télécharger des objets de taille modeste à travers un Squid utilisant ou n'utilisant pas notre bac à sable. Nos résultats sont consignés dans le tableau 2. Ces résultats montrent que la protection apportée par notre bac à sable induit un surcoût négligeable qui rend notre approche adaptée à des applications réelles.

5. Travaux apparentés

Notre travail est similaire aux travaux sur les analyseurs dynamiques de code, de part l'injection de test sur la validité des données d'un processus pendant son exécution. Néanmoins, contrairement à ces approches qui s'appuient sur des techniques comme l'inférence de bornes [10] ou l'injection d'erreurs [7], notre approche détecte l'occurrence de fautes logicielles plutôt que leur potentialité. De plus, les analyseurs de code interviennent sur les sources d'un programme avant la compilation tandis qu'Arachne travaille au niveau du code binaire et après le démarrage des applications.
Notre approche s'apparente également aux techniques usitées par les bacs à sable. En effet, ceux-ci restreignent la propagation d'une faute, afin d'empêcher son exploitation par un pirate informatique, ceci en isolant le logiciel du reste du système. Cette isolation peut être réalisée à l'aide d'une politique de sécurité décrivant strictement les droits d'une application [5] ou encore par la séparation des espaces d'adressage des processus [14]. Notre proposition se distingue des bacs à sables en ce sens ou l'isolation n'est pas appliquée à l'ensemble du logiciel, mais uniquement au code et aux données sensibles du programme.
À l'instar des systèmes de détection d'intrusion (IDS), notre approche surveille l'exécution d'un programme afin de déceler un chemin d'exécution révélant d'une utilisation malhonnête.

Contrairement, aux IDS, qui utilisent des bases de données de patrons d'exécution qui sont basés sur des données externes (appels systèmes ou données reçues) [12], Arachne examine l'exécution interne du programme à travers des règles génériques révélant les fautes d'exécution.

6. Conclusion

Nous avons présenté dans cet article un atelier pour le déploiement à chaud de correctifs de sécurité ciblés. Contrairement à nos précédents travaux sur l'injection à chaud de correctif quelconque, la solution proposée dans cette article garantit l'intégrité de l'application après correction. Pour cela, nous nous sommes limités dans notre cadre de conception aux quatre types de bogues les plus fréquents : le débordement de tampon, le bogue de format, la double désallocation bogue et le débordement d'entier qui représentent 60% des bogues connus. Les correctifs ont été écrits et déployés avec notre outil de réécriture de code à chaud Arachne. Cet outil injecte dynamiquement le correctif compilé dans l'application à mettre à jour. Pour des raisons techniques, nous nous sommes pour l'instant limité aux programmes écrits en C s'exécutant sur une plate-forme IA32 dans un environnement Linux. L'atelier a été testé sur plusieurs applicatifs systèmes complexes dont les derniers trous de sécurité répondent à un ou plusieurs des 4 types de bogues traités. Bien évidement, d'autres types de bogues peuvent être traités avec notre atelier.

Bibliographie

1. blexim. Basic integer overflows. *Phrack*, 11(60), December 2002.
2. CERT - Carnegie Mellon University. Vulnerability note vu#613459, February 2002. publié en ligne : `http://www.kb.cert.org/vuls/id/613459`.
3. Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 54–66, N.Y., January 19–21 2000. ACM Press.
4. Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Proc. of 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, March 2005. to appear.
5. Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies : A retrospective. In *New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999. ACM SIGSAC, ACM Press.
6. GMV Conseil. étude et statistiques de la sinistralité informatique en france. publié en ligne sur `https://www.clusif.asso.fr/fr/production/sinistralite/docs/Synthese200%2.pdf`, 2002. CLUSIF.
7. Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX Security Symposium*, pages 1–14, Berkeley, July 22–25 1996. Usenix.
8. Internet Storm Center. Survival Time. publié en ligne `http://isc.sans.org`, 2004.
9. Angelos D. Keromytis. Patch-on-demand saves even more time ?. In *in IEEE Computer, vol. 37, no. 8,*, pages 94–96, N.Y., August 2004. IEEE.
10. Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *Proceedings of the 11th USENIX Security Symposium*, pages 121–136. USENIX, August 2003.
11. Nicolas Loriant, Marc Ségura-Devillechaise, and Jean-Marc Menaud. Des correctifs de sé-

curité à la mise à jour - audit déploiement distribué et injection à chaud. In *DECOR'2004, 1ère conférence Francophone sur le déploiement et la (re)configuration de Logiciels*, pages 65–76, Grenoble, October 2004.
12. Fred B. Schneider. Enforcable security policies. Technical Report TR99-1759, Computer Science Department, Cornell University, July 1999.
13. Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect : Towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119, Boston, Massachusetts, USA, March 2003. ACM Press.
14. Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, December 1993. ACM Press.
15. J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, University of Illinois at Urbana-Champaign, May 2003.
16. Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

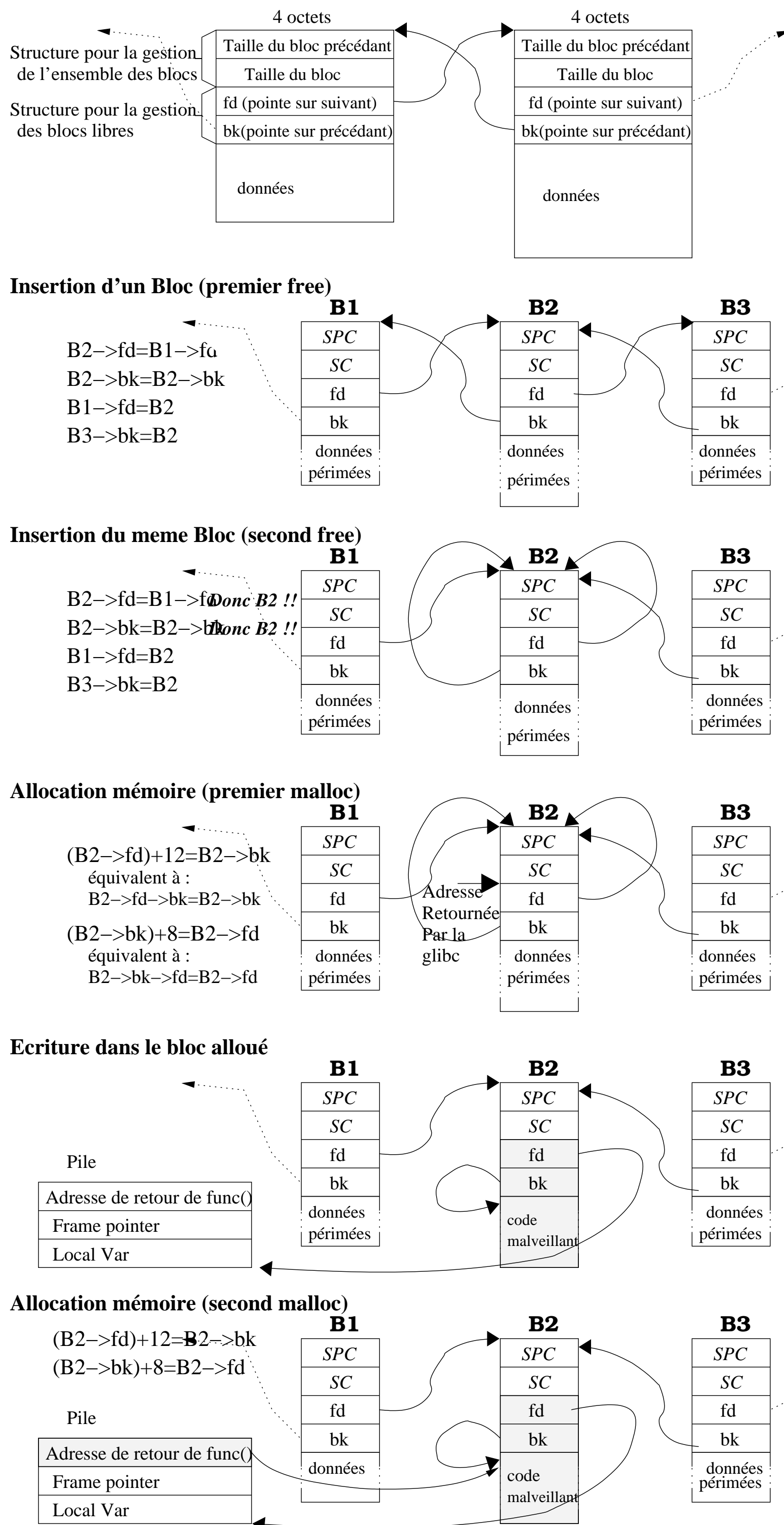

FIG. 2 – le bogue de double désallocation