Olivier Crête

Les débordements de tampons sur les systèmes Unix

Mémoire présenté à
l'Organisation du baccalauréat international

Collège François–Xavier–Garneau
Québec

22 janvier 2001

Précis

Depuis les débuts de l’informatique, la sécurité préoccupe les concepteurs de systèmes. Les débordements de tampons[1] sont de loin le problème le plus criant dans le domaine de la sécurité informatique. La majorité des problèmes de sécurité découverts dans les dernières années sur les systèmes Unix sont causés par des débordements de tampons. Il est donc important de comprendre leur fonctionnement et les méthodes employées par les attaquants malicieux pour les exploiter. Les méthodes pour se protéger contre leurs effets seront présentées et comparées afin de trouver des solutions à ce problème. Ils peuvent se produire aussi bien dans la pile que dans d’autres sections de la mémoire, même si ceux se produisant dans la pile sont les plus communes, les plus exploitées et les plus étudiées. Il existe plusieurs manières de réduire les risques associés aux débordements de tampons. La méthode la plus efficace est la vérification manuelle, mais celle–ci étant longue et coûteuse des outils automatisés ont été inventés pour aider le vérificateur. Ces outils peuvent aussi bien être des outils d’analyse statique que des outils de vérification dynamique, reconnaissant les débordements de tampons dès qu’ils se produisent. La vérification manuelle étant malheureusement aussi souvent imparfaite, des compilateurs spéciaux ont été créés pour tenter de protéger les logiciels contre les conséquences des débordements de tampons. Ils tentent de s’assurer que ceux–ci deviennent impossible à exploiter ou, à tout le moins, que le programme puisse être arrêté en toute sécurité. Il existe enfin des solutions créées pour protéger les systèmes contre les logiciels pour lesquels le code source n’est pas disponible. Ce sont des bibliothèques modifiées protégeant les fonctions standards contre les débordements de tampons et aussi une modification du noyau empêchant l’exécution d’instructions à partir de la pile.

---

1 De l’anglais *buffer overflows*, selon selon *Le grand dictionnaire terminologique.*

Table des matières

Précis..2

Introduction..4

1. Le problème..6

1.1 Concepts de base...6
1.2 Types d'attaques...6
1.2.1 Attaques de la pile...6
1.2.2 Autres attaques..8

2. Les solutions..10

2.1 La vérification manuelle...10
2.2 Les outils de protection automatisés...12

Conclusion...16

Bibliographie..17

Introduction

Depuis les débuts de l'informatique, la sécurité préoccupe les concepteurs de systèmes. Déjà durant les années 60, les concepteurs de systèmes mettaient l'accent sur les différents moyens de protéger les systèmes informatiques contre les attaques extérieures. L'apparition de grands réseaux informatiques publics, en particulier l'Internet, a rendu la vie plus facile aux attaquants malicieux. La plupart de ces attaques sont faites en profitant d'erreurs de programmation dans les logiciels employés. La gravité du problème ne doit pas être sous–estimée: le Computer Security Institute de San Franscisco fait un sondage annuel. En 2000, cette société a sondé 643 experts en sécurité travaillant dans les secteurs privé et public où 74% de ces compagnies ont subi des pertes financières à cause d'intrusions dans leur système: le total s'élève à 265 millions de dollars américains en l'an 2000, alors que la moyenne des trois années précédentes était de 120 millions[1]. Dans la dernière décennie, le type le plus commun d'erreur[2] ayant causé des intrusions sont des débordements de tampons. Ce type de problème se retrouve sur presque tous les systèmes informatiques utilisés, il découle de la façon dont plusieurs langages informatiques ont été conçus, en particulier le C et ses descendants. L'étude des débordements de tampons et de leurs solutions nous permettra donc de déterminer les solutions privilégiées qui doivent être adoptées par les administrateurs de systèmes et les programmeurs pour réduire les intrusions dans les systèmes informatiques.

Le système Unix est le système pour lequel le C a été inventé et ses descendants dominent toujours les serveurs, surtout sur l'Internet. Ils sont la principale cible des attaques par les braqueurs informatiques[3]. L'étude des débordements de tampons sera donc limitée aux systèmes Unix à cause de leur grande popularité et l'existence d'une abondante documentation sur leur fonctionnement interne et de l'existence de plusieurs versions libres[4]. Bien que différentes solutions automatisées aient été proposées pour régler les problèmes de

---

1 Computer Security Institute, *Ninety percent of survey respondents detect cyber attacks.*

2 SANS Institute. *Curmudgeon's Executive Summary.*

3 De l'anglais *cracker*, selon *Le grand dictionnaire terminologique.*

4 L'expression *libre* est ici employé dans le sens défini par la Free Software Foundation (http://www.fsf.org). C'est–à–dire, libre comme la « liberté d'expression » dans son sens le plus démocratique.

débordement de tampons, aucune de ces méthodes n'est capable de donner un résultat aussi efficace que la vérification manuelle du code à sécuriser. Après avoir expliqué les débordements de tampons et les différentes manières de les exploiter, qu'ils soient situés dans la pile ou dans un autre segment de mémoire[1], il devient alors possible de discuter des solutions automatisées qui s'offrent au programmeur et à l'administrateur de système et d'expliquer leurs avantages et leurs inconvénients.

1 Par exemple, dans le *heap* qui ne possède pas de traduction plus précise.

1. Le problème

Un tampon est une région de la mémoire utilisée pour emmagasiner des données. Dans le langage C, la représentation la plus commune d'un tampon est le tableau[1], c'est–à–dire la répétition de plusieurs éléments identiques, le plus souvent des octets ou caractères pour former des chaînes de texte. En C, une chaîne de caractères se termine par un octet nul et traditionnellement la plupart des fonctions opérant sur les chaînes de caractères se fient à cet octet nul pour en connaître la fin. Par exemple, la fonction strcpy()[2] accepte comme argument un pointeur vers le début d'un tampon vide et le début d'un tampon contenant une chaîne de caractères et copie le contenu du second dans le premier jusqu'à atteindre l'octet nul. Dans des conditions normales, le tampon vide est assez grand pour contenir la chaîne en entier, mais s'il est trop petit, alors ce qui suit ce tampon dans la mémoire sera aussi écrasé par la chaîne de caractères. La chaîne déborde alors à l'extérieur du tampon et il est ainsi possible de modifier le contenu de la zone de mémoire qui suit le tampon d'une façon non prévue par le programmeur.

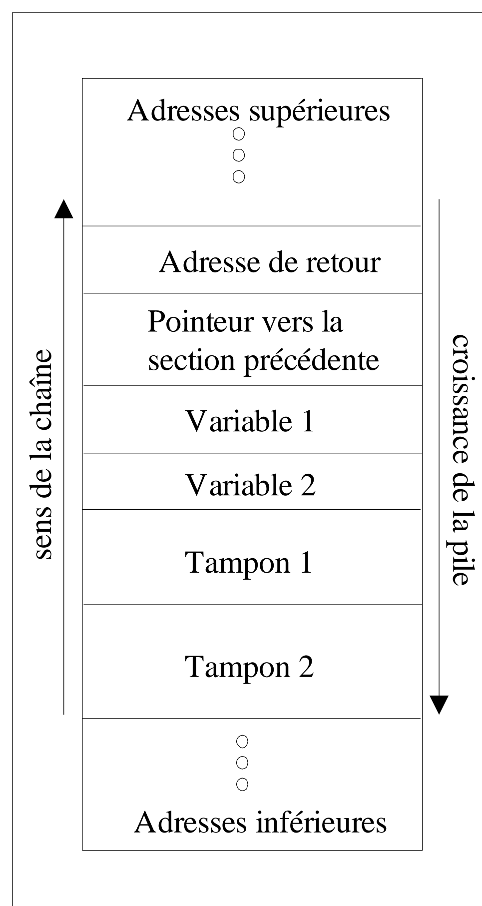

Illustration 1   La pile d'un logiciel sur Linux (ou un autre système Unix) sur un processeur de type x86

La zone de la mémoire qui est le plus souvent attaquée est la pile. Elle fonctionne à la manière d'un LIFO[3], c'est–à–dire « dernier entré, premier sorti », on ajoute les informations sur le dessus de la pile et on les retire au même endroit. Sur la plupart des architectures les plus communes (par exemple, x86, SPARC, MIPS, Alpha)[4], la pile croît du haut vers le bas, c'est–à–dire que le dessus a une adresse

1 De l'anglais *array*, selon *Le grand dictionnaire terminologique.*
2 *Linux Programmer's Manual*, version du 11 avril 1993.
3 De l'anglais *Last In, First Out*, selon *Le grand dictionnaire terminologique.*
4 Evan Thomas. *Attack Class: Buffer Overflows.*

inférieure au dessous. Quand une fonction est appelée, une nouvelle section[1] est créée sur la pile, on pousse l'adresse de la prochaine instruction à exécuter dans la fonction appelante, puis l'adresse de la base de la section précédente dans la pile et enfin on y crée les variables locales à la fonction. Une de ces variables locales peut être un tampon; si les données écrites dépassaient la longueur du tampon et débordaient, cela affecterait d'abord les variables qui le suivent et ensuite cela pourrait modifier l'adresse de retour.[2]

C'est là que réside le danger. L'adresse de retour ayant été modifiée, le programme, à la fin de l'exécution de la fonction, tentera de retourner à la fonction précédente en lisant l'instruction à l'adresse spécifiée. Si cette adresse ne fait pas partie de la mémoire allouée à ce logiciel, le noyau[3] arrêtera l'exécution du logiciel. Par contre, si cette adresse était valide, l'ordinateur continuerait à exécuter les instructions à partir de ce point. Il ne reste plus à l'attaquant qu'à insérer un morceau de code à exécuter. Ce type d'attaque requiert donc deux étapes, l'insertion du code étranger et son activation[4]. La méthode la plus simple consiste à mettre le code dans le tampon lui−même s'il est assez grand ou de le mettre après le tampon s'il n'est pas assez grand et qu'il y assez de place après celui−ci. On peut aussi placer le code dans un autre tampon, même si celui−ci n'est pas situé dans la pile.

Le but de ce genre d'attaque est donc de contrôler le flux de commande du logiciel et ainsi exécuter des instructions autres que celles désirées par le propriétaire du système. Le type d'instructions exécutées varie, mais le plus souvent il s'agit de ce que l'on appelle le « shellcode »[5], c'est−à−dire quelques instructions qui remplacent le contenu du processus[6] par l'exécution d'une coquille[7]. Il est aussi possible d'ouvrir un mécanisme d'échange[8] si le logiciel attaqué est un serveur et que l'on n'a donc pas accès à son entrée standard[9], et

1 En anglais, ces sections sont des *stack frames.*
2 Aleph One, *Smashing The Stack for Fun And Profit.*
3 De l'anglais *kernel*, selon *Le grand dictionnaire terminologique.*
4 Crispin Cowan et al. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer−Overflow Attacks*, p. 2.
5 La version la plus utilisée d'un tel « shellcode » est celle publiée par Aleph One dans son article *Smashing The Stack for Fun And Profit.*
6 De l'anglais *process*, selon *Le grand dictionnaire terminologique.*
7 De l'anglais *shell*, selon *Le grand dictionnaire terminologique*, un logiciel qui interprète des commandes.
8 De l'anglais *socket*, selon *Le grand dictionnaire terminologique.*
9 De l'anglais *stdin (standard input)*, selon *Le grand dictionnaire terminologique.*

remplacer son entrée standard par ce mécanisme d'échange[1]. Mais cette technique est plus compliquée et requiert plus de codes et plus de travail de la part du l'attaquant, mais elle permet d'exploiter des vulnérabilités à distance sur des systèmes ayant très peu de services ouverts ou même un seul, par exemple un serveur HTTP ou un serveur de courrier.

D'autres attaques ne visent pas à changer les instructions exécutées, mais uniquement à modifier les données. Le « Internet Worm » ou « Morris Worm », du nom de son auteur, est le premier virus capable de se déplacer via l'Internet et il aurait utilisé ce type d'attaque[2]. Il remplaçait le nom du logiciel exécuté par *fingerd.* Celui–ci avait */usr/ucb/finger* dans sa mémoire comme programme à exécuter, le « Morris Worm » le remplaçait par */bin/sh* en remplissant un tampon et en débordant jusqu'à la fin de la pile, le nom du programme à exécuter étant après la fin de celle–ci, il obtenait donc ainsi une coquille dont l'entrée et la sortie standard étaient connectées sur un mécanisme d'échange. Il est possible d'utiliser ce genre d'attaque pour modifier toutes sortes de données présentes dans la mémoire après un tampon d'où il est possible de faire déborder des données[3]. Par exemple, s'il y a un tampon de 32 octets pour contenir le nom de l'usager suivi d'un autre tampon qui contient un mot de passe, on peut déborder du nom de l'usager et écraser le mot de passe pour le modifier et ainsi avoir accès au système. Il peut aussi modifier le nom de l'usager ou son numéro pour obtenir des privilèges plus élevés[4]. Ce type d'attaque où on change des données peut être utilisé sur la pile, mais l'est plus fréquemment quand le débordement a lieu ailleurs que sur la pile, dans une autre partie de la mémoire.

Ces autres parties de la mémoire sont le plus souvent la section contenant les variables statiques initialisées et non–initialisées (BSS), et le segment de mémoire[5] alloué dynamiquement[6]. Ces sections contiennent uniquement des données et sont donc identiques si elles sont considérées du point de vue des débordements de tampons. Sur Unix, la mémoire réservée à un logiciel est divisée en plusieurs morceaux, parmi ceux–ci, il y a la

1 Taeho Oh. *Advanced buffer overflow exploits.*
2 Bob Page. *A Report On The Internet Worm.*
3 Matt Conover. *w00w00 on Heap Overflows.*
4 Ces exemples sont pris presque textuellement de Matt Conover. *w00w00 on Heap Overflows.*
5 De l'anglais *heap*, selon *Le grand dictionnaire terminologique.*
6 Par exemple les variables dont l'espace est crée par *malloc* en C ou *new* en C++.

pile, dont il a été question, puis la section contenant le programme lui–même, que l'on appelle aussi *text*[1]. Il y a aussi les données initialisées qui sont prises directement dans le fichier exécutable quand le logiciel est chargé dans la mémoire et la zone contenant des données non–initialisées (ou BSS[2]) dont seuls les symboles identifiant le contenu sont chargés, mais dont le contenu est initialisé à zéro par le noyau au chargement du programme. Ces différentes zones contiennent toutes des données et peuvent, bien sûr, contenir des tampons. On peut donc aussi déborder de ces tampons, on peut changer le contenu de certaines parties de la mémoire et remplacer les données que cette mémoire contient.

En C et en C++, ces sections de la mémoire peuvent souvent contenir des pointeurs vers des fonctions[3], ces pointeurs permettent l'existence de fonctions qui peuvent être remplacées, par exemple à l'aide de plugiciel[4]. Quand le programme appelle la fonction et passe par ce pointeur, il exécute alors les instructions à partir de l'endroit pointé, exactement comme l'adresse de retour dans la pile. Si l'on peut modifier ce pointeur, il est alors possible de modifier le flux de commande et de forcer le logiciel à exécuter des instructions qui ont été mises quelque part dans la mémoire du logiciel, il est ainsi possible d'exécuter un « shellcode » ou quelque autre code.

Les débordements de tampons présentent donc de nombreuses facettes et il est possible de les utiliser de plusieurs manières à des fins malicieuses. Ils peuvent se produire partout où il peut y avoir des tampons, c'est–à–dire dans presque toute la mémoire contenant des données. Il existe diverses méthodes pour les exploiter pour pénétrer le périmètre de sécurité d'un système et ainsi le compromettre.

1 Tirunelveli J. Sundar. *Unix processes.*
2 *id.*
3 Traduction libre de *fonction pointer.*
4 De l'anglais *plug–in*, selon *Le grand dictionnaire terminologique.*

2. Les solutions

Différentes solutions s'offrent aux problèmes posés par les débordements de tampons. Ceux–ci étant la principale cause d'intrusions dans les systèmes informatiques à l'heure actuelle, de nombreuses équipes dans le monde se sont penchées sur différentes solutions. Il y a d'abord un point de vue radical qui consiste à rejeter le C et le C++ et à proposer d'autres langages qui vérifient les tampons tels le Fortran, le Java ou le ADA; mais il est utopique de croire que les millions de lignes de code déjà écrites puissent être remplacées. Il existe plusieurs autres solutions, certaines nécessitent l'accès au code source comme la vérification manuelle ou à l'aide d'outils d'analyse statique et dynamique. Il est aussi possible d'utiliser des compilateurs modifiés protégeant le programme partiellement ou totalement contre les débordements. Il y a aussi d'autres solutions, ne nécessitant pas d'accès au code source, qui ont pour but de diminuer les risques et de rendre plus difficile l'exploitation de ces débordements à des fins malicieuses, sans les éliminer. Entre autres, l'usage de fonctions standards modifiées et de certaines modifications au noyau permettent de réduire l'impact des débordements de tampons.

La méthode la plus commune et sûrement la plus efficace pour éliminer les problèmes de dépassement de capacité est la vérification manuelle par des humains de chaque ligne des logiciels à sécuriser. Il est aussi important d'utiliser des techniques de programmation favorisant la sécurité[1]: par exemple, limiter au minimum la taille des logiciels nécessitant des privilèges et limiter ces privilèges à leur niveau minimal. Il est également important de définir les interfaces de manière très précise et de vérifier toute entrée de données pour s'assurer qu'elles entrent strictement dans les normes énoncées.

L'usage de logiciels dont le code source est public semble aussi être très avantageux, car ceux–ci sont plus souvent vérifiés par des individus autres que les auteurs, ceux–là peuvent ainsi découvrir les erreurs avant qu'elles ne deviennent problématiques. Ces logiciels semblent donc afficher un niveau de qualité beaucoup plus élevé que leur

1 Peter Galvin. *The Unix Secure Programming FAQ.*

contrepartie dont le code source reste caché[1]. Le projet OpenBSD[2], qui vise à créer le système d'exploitation le plus sécuritaire possible et dont les membres vérifient manuellement chaque ligne du système, n'a d'ailleurs pas eu de trou de sécurité exploitable à distance dans les trois dernières années[3]. Il est donc évident que la vérification manuelle donne d'excellents résultats. Mais cette méthode est très coûteuse en main d'oeuvre ce qui a entraîné la création d'outils pouvant faciliter ce genre de travail.

Il existe plusieurs types d'outils permettant d'aider le vérificateur dans son travail d'analyse du logiciel en lui permettant de découvrir les trous de sécurité potentiels. Il existe tout d'abord des outils d'analyse statique. Par exemple, à l'aide de modèles synthétiques créés à partir du code source des logiciels, il est possible de détecter certains débordements de tampons qu'il aurait été impossible de trouver sans l'aide de tels outils. Il en est ainsi des erreurs où le tampon est débordé d'une seule unité qui sont particulièrement difficiles à reprérer et c'est alors que ce genre d'outil montre toute sa force[4]. Malheureusement, il est très difficile de créer un modèle complet et correct de la plupart des logiciels. Wagner et al. utilisent une méthode simplifiée qui ne vérifie pas certains cas et qui laisse donc passer des erreurs; elle introduit un très grand nombre de fausses alertes, les essais effectués montrent qu'environ une erreur rapportée sur dix est réelle et que le reste sont des erreurs d'approximation de leur système. Il serait également possible de rendre leur système plus complet et leur analyse plus précise, mais cela requerrait une puissance de calcul beaucoup plus importante[5]. Cette méthode ne peut donc se substituer à une vérification manuelle et peut, au mieux, l'aider, mais plus probablement lui nuire car cela incite le vérificateur à ne pas vérifier attentivement le reste du programme et d'analyser toutes les erreurs potentielles qu'une analyse automatisée n'aurait pas été en mesure de détecter.

Il existe aussi des outils dynamiques qui vérifient les débordements de tampons pendant l'exécution du logiciel. Le plus connu est probablement Purify de Rational[6], qui est disponible sur la plupart des versions commerciales de Unix (Solaris/SPARC, HP–UX, SGI

1 Miller, Fredrickson et al. *An Empirical Study in the Reliability of UNIX utilities.*
2 *OpenBSD.*
3 Selon leur page web (http://www.openbsd.org) et ce en date du 17 janvier 2001.
4 David Wagner et al. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*, p.2.
5 id. p. 5.

IRIX et Siemens Reliant Unix). Ce logiciel vérifie en temps réel chaque appel vers un tampon et vérifie si le pointeur ne sort pas du tampon appelé. Évidemment, exécuter un logiciel à l'intérieur d'un tel système impose une importante réduction de la vitesse du logiciel et ne peut être utilisé que dans un contexte de débogage. Il existe aussi une version modifiée de GCC (GNU C Compiler)[1] qui introduit du code dans chaque programme pour vérifier chaque accès à un tampon. Un tel compilateur modifie donc le programme pour ajouter du code supplémentaire qui vérifie chaque accès à un tampon et arrête le programme immédiatement si un pointeur ne pointe plus vers un endroit valide. Cela permet donc de détecter facilement les débordements de tampons. Cet outil est classé parmi les outils de débogage plutôt que dans la catégorie suivante, celle des compilateurs protecteurs, parce qu'un programme sur lequel il est utilisé ralentit de cinq à six fois[2]. Ce genre de technique dynamique ne peut découvrir des débordements que s'ils se produisent, elle ne peut pas trouver les débordements lors de séquences inhabituelles qu'une analyse statique pourrait trouver: celle−ci vérifiant l'ensemble des possibilités théoriques sans égard à leur utilisation. Les auteurs de cette version modifiée de GCC proposent d'utiliser leur compilateur sur le produit final pour ainsi empêcher tous les débordements, au moins pour les applications les plus sensibles, mais cela est peu pratique.

Il existe d'autres compilateurs modifiés qui permettent de réduire les débordements de tampons ou, à tout le moins, de rendre leur exploitation plus difficile tout en ayant un effet beaucoup moins important sur la vitesse. Ces compilateurs modifiés tentent de réduire l'impact sur la performance des logiciels: les auteurs de StackGuard affirment n'avoir aucun impact mesurable[3] sur Apache[4] et sshd[5]. Celui−ci est basé, comme tous les autres compilateurs qui offrent ce genre de fonctions, sur GCC. StackGuard place une variable juste

6 Rational, *Rational Purify for Unix*.
1 À partir de la version 3, GCC veut dire GNU Compiler Collection car il supporte maintenant de nombreux langages autres que le C (entre autres le C++, le Objective C, le Java et le Fortran) . La page web officielle est située à http://gcc.gnu.org
2 Jones, R. W. M. et Kelly, P. H. J. *Backwards−compatible bounds checking for arrays and pointers in C programs*, p. 9.
3 Crispin Cowan et al. *Protecting Systems from Stack Smashing Attacks with StackGuard*, p. 6–7.
4 Serveur Web le plus populaire, ayant une part de marché d'environ 60% selon *The Netcraft Web Server Survey*.
5 SSH (Secure Shell) est un remplacement sécuritaire pour rsh (Remote Shell) et est un des éléments critiques de la sécurité de très nombreux sites.

avant l'adresse de retour, cette variable peut contenir deux différents types de valeurs, il peut être soit un finisseur[1], c'est–à–dire un caractère qu'il est impossible de mettre dans une chaîne de caractères, soit une valeur aléatoire. Cette variable est appelée « canari » et on ne peut donc arriver jusqu'à l'adresse de retour en débordant d'un tampon sans écraser ce canari. Lorsque qu'une fonction se termine, le logiciel vérifie ce canari avant de sauter vers l'adresse de retour et le compare à la valeur attendue; si le canari a été modifié, le programme est alors arrêté[2]. Cette méthode est très efficace pour empêcher l'exploitation des tampons situés dans la pile. Par contre, elle ne protège pas des attaques dans une autre partie de la mémoire ou d'une attaque qui ne tente pas de modifier l'adresse de retour.

Il est aussi possible d'exploiter un logiciel compilé avec StackGuard dans certains cas s'il y a un pointeur situé après le tampon. Si un pointeur suit un tampon vulnérable et que ce pointeur sera utilisé pour copier des données qui peuvent aussi être modifiées (ou copier des données à partir d'un autre pointeur qui peut lui aussi être modifié), il est alors possible d'écraser uniquement l'adresse de retour; ou, à tout le moins, de la modifier en évitant de toucher au canari rendant la méthode de StackGuard inopérante.

Une équipe de IBM située au Japon propose une méthode pour régler ce problème; aussi elle a testée cette méthode grâce à sa propre version modifiée de GCC et qui semble basée sur celle de StackGuard. Non seulement insère–elle un canari avant l'adresse de retour comme StackGuard, mais de plus, elle modifie l'ordre des variables dans la mémoire pour placer les tampons après tous les pointeurs, rendant ainsi impossible de modifier un pointeur à l'aide d'un débordement dans la pile[3]. Par contre, cette méthode n'est pas efficace contre les débordements qui se produisent dans le segment de mémoire où des tampons peuvent être alloués de manière dynamique[4].

Il existe aussi un troisième type de compilateur modifié. Le groupe ayant produit ce compilateur a également basé son travail sur GCC; cependant, il utilise une méthode

1 De l'anglais *terminator*, selon *Le grand dictionnaire terminologique.*
2 Crispin Cowan et al. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer−Overflow Attacks.*
3 Hiroaki Etoh et Kunikazu Yoda, *Protecting from stack−smashing attack*, section 4.2.
4 id, section 4.4.

différente: elle ne tente pas de détecter les attaques une fois qu'elle se sont produites ou de les rendre impossible, elle tente plutôt de les rendre inopérantes. Pour ce faire, à chaque fois qu'une fonction est appelée, une copie de l'adresse de retour est enregistrée dans une table située au tout début de la mémoire et qui n'est donc précédée d'aucun tampon. Il est par conséquent impossible de modifier cette table à l'aide d'un débordement. À la fin de l'exécution de la fonction, l'adresse de retour située dans cette table est comparée à la valeur située dans la pile. Si elles sont différentes, il peut, soit arrêter le programme, soit écraser la valeur située dans la pile et continuer l'exécution du logiciel comme si aucun événement ne s'était produit[1]. Cette méthode est par contre sensible aux mêmes attaques que celles qui ont été proposées contre StackGuard, c'est–à–dire que si un pointeur peut être modifié, il sera alors possible de modifier cette table et l'adresse de retour située dans la pile et ainsi prendre le contrôle du programme.

Toutes les méthodes exposées précédemment requiert l'accès au code source des logiciels. Il existe malheureusement encore certains logiciels pour lesquels cela est impossible. D'autres méthodes ont alors été proposées pour tenter d'offrir un certain degré de protection aux logiciels pour lesquels le code source n'est pas disponible. La première et la plus ancienne de ces méthodes est de rendre impossible l'exécution de codes à partir de la pile. Il existe une option sur les versions récentes de Solaris[2] et il existe aussi une rustine[3] pour Linux produite par Solar Designer[4]. La plupart des microprocesseurs permettent de donner des permissions à chaque région de la mémoire, en particulier, de spécifier s'il est permis de lire, d'écrire ou d'exécuter le contenu d'un morceau de la mémoire. Par exemple, il est habituellement impossible d'écrire dans la région contenant la portion exécutable d'un logiciel[5]. De la même manière, il a été proposé de rendre le contenu de la pile non–exécutable, il devient alors impossible d'insérer le code malicieux. Il est très facile de contourner cette protection en le mettant dans une autre partie de la mémoire où en utilisant du code déjà présent dans la mémoire du logiciel[6]. Le principal avantage de cette méthode est

1 Vendicator, *Stack Shield: A "stack smashing" technique protection tool for Linux.*
2 John McDonald, *Defeating Solaris/SPARC Non−Executable Stack Protection.*
3 De l'anglais *patch*, selon *Le grand dictionnaire terminologique.*
4 OpenWall Project, *Linux kernel patch from the Openwall project.*
5 En anglais, c'est le *text.*
6 Rafal Wojtczuk, *Defeating Solar Designer's Non−executable Stack Patch.*

de rendre inopérant la plupart des logiciels préfabriqués pour exploiter les trous connus; tant que son usage ne sera pas répandu, la plupart des attaquants passeront probablement au prochain ordinateur plutôt que de tenter d'attaquer un système protégé ainsi.

Il a été constaté[1] que la grande majorité des débordements de tampons sont causés par l'usage inapproprié de certaines fonctions standards qui n'ont pas été conçues en tenant compte d'éventuels problèmes de débordements de tampons. Ces fonctions sont: strcpy(), strcat(), getwd(), gets(), realpath(), scanf(), fscanf(), sscanf() et sprintf()[2]. Il existe des remplacements sécurisés tels strncpy()[3] que les programmeurs devraient employer au lieu des versions traditionnelles. Malheureusement, celles–ci sont souvent encore trop utilisées. Il a donc été proposé de les réécrire de manière à ce qu'elles ne permettent pas de dépasser la fin de sa section de la pile et ne puissent ainsi corrompre l'adresse de retour[4]. Cela permet donc de réduire le nombre d'attaques. Cette méthode a par contre des limites importantes[5]. Tout d'abord, pour que cette méthode réussisse, il faut que le pointeur vers la section précédente soit présent dans la section actuelle, car c'est grâce à lui que le début de la section est repérée, certains compilateurs peuvent l'éliminer lors de leur optimisation. Il faut également que ces fonctions ne soient pas inclues dans le programme, mais appelées à partir de bibliothèques dynamiques. Et évidemment, elle ne protège que contre l'usage inapproprié de certains fonctions et non contre l'ensemble des débordements de tampons.

1 *Navjot Singh, Arash Baratloo et Timothy Tsai. Transparent Run–Time Defense Against Stack Smashing Attack*.
2 Ainsi que vscanf(), vfscanf(), vsscanf() et vsprintf().
3 Linux Documentation Project. *Linux Programmer's Manual*, scanf(3).
4 Arash Baratloo, Timothy Tsai, et Navjot Singh. *Libsafe: Protecting Critical Elements of Stacks.*
5 Crispin Cowan et Perry Wagle, *Re: libsafe.*

Conclusion

Le problème posé par les débordements de tampons est central à la sécurité des systèmes informatiques modernes. Ils permettent de prendre le contrôle du flux de commande des logiciels et ainsi de compromettre les systèmes. Ils peuvent aussi permettre de modifier les données et de les corrompre. Il est heureusement possible de les arrêter. Quelques méthodes existent pour protéger les logiciels dont le code source n'est pas public, mais ces méthodes ont un effet limité. La plupart des techniques de protection requièrent l'accès au code source. Bien que les différentes techniques proposées incluent généralement la compilation des logiciels, ce ne sont en fait que des béquilles pour pallier les erreurs de programmation.

La seule méthode vraiment efficace est celle qui consiste à trouver et à régler ces erreurs par une vérification manuelle complète et exhaustive de tous les aspects sensibles du système. En fait, les logiciels dont le code source est public sont habituellement de meilleure qualité[1] que leurs concurrents commerciaux. Cette grande qualité serait due à la plus grande présence de contrôle par les pairs[2]. L'erreur étant le propre de l'homme, il est peu probable que les débordements de tampons disparaissent tant que des langages peu sûr seront utilisés, mais en publiant le code source, la qualité des logiciels pourra s'améliorer et offrir une plus grande sécurité pour tous.

---

1 Miller, Fredrickson et al., *Fuzz revisited: A Re−examination of the Reliability of UNIX Utilities and Services*, p. 6.
2 Eric S. Raymond. *The Cathedral & the Bazaar*, ch. 4.

Bibliographie

Aleph One. *Smashing the Stack for Fun and Profit*, Phrack, Volume 7, Numéro 49, 8 novembre 1996.

Bakke, Peat, etSteve Beattie, Crispin Cowan, Aaron Grier, Heather Hinton, Dave Maier, Calton Pu, Perry Wagle, Jonathan Walpole et Qian Zhang. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer−Overflow Attacks*, Oregon Graduate Institute of Science & Technology, 1997.

Baratloo, Arash et Timothy Tsai, et Navjot Singh. *Libsafe: Protecting Critical Elements of Stacks*, Page web [visitée le 8 janvier 2001], http://www.bell−labs.com/org/11356/libsafe.html
Beattie, Steve et Crispin Cowan, Calton Pu , Perry Wagle etJonathan Walpole. *Attacks and Defenses for the Vulnerability of the Decade*, Oregon Graduate Institute of Science & Technology, 2000.

Bulba et Kil3r. *Bypassing StackGuard and StackShield*, Phrack, Volume 10, Numéro 56, 1er mai 2000.

Computer Security Institute, *Ninety percent of survey respondents detect cyber attacks, 273 organizations report $265,589,940 in financial losse*, 22 mars 2000, San Francisco.

Conover, Matt (a.k.a. Shok) & w00w00 Security Team. *w00w00 on Heap Overflows*, Page web [visitée le 8 janvier 2001], http://www.w00w00.org/files/articles/heaptut.txt

Cowan, Crispin, et Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, et Erik Walthinse. *Protecting Systems from Stack Smashing Attacks with StackGuard*, Oregon Graduate Institute of Science & Technology, 1999.

Cowan, Crispin et Perry Wagle. *Re: libsafe*, security−audit@ferret.lmh.ox.ac.uk (25 avril 2000).

Etoh, Hiroaki et Kunikazu Yoda. *Protecting from stack−smashing attack*, IBM Research Division, Tokyo Research Laboratory, Yamato, 2000.

Free Software Foundation, *GNU's Not Unix*, Page web [visitée le 17 janvier 2001], http://www.fsf.org

Galvin, Peter. *The Unix Secure Programming FAQ*, 1998, Page web [visitée le 7 janvier 2001], http://www.sunworld.com/sunworldonline/swol−08−1998/swol−08−security.html

GCC Team. *GCC Home Page*, page web [visitée le 20 janvier 2001], http://gcc.gnu.org

Jones, R. W. M. etKelly, P. H. J. *Backwards−compatible bounds checking for arrays and pointers in C program*, Imperial College of Technology, Science etMedice, London, 1997.

Kirch, Olaf. *Re: libsafe*, security−audit@ferret.lmh.ox.ac.uk (25 avril 2000).

Lefty. *Buffer Overruns, whats the real story?*, Page web [visitée le 8 janvier 2001], http://julianor.tripod.com/stack−history.txt

Linux Documentation Project, *Linux Programmer's Manual, 2000.*

McDonald, John. *Defeating Solaris/SPARC Non−Executable Stack Protection*, 2 mars 2000, Page web [visitée le 7 janvier 2001], http://julianor.tripod.com/non−exec−stack−sol.html

Miller, Fredrickson et al. *Fuzz revisited: A Re−examination of the Reliability of UNIX Utilities and Services*, University of Wisconsin, Madisson, 2000.

Mudge @ l0pth. *How to write Buffer Overflows*, 20 octobre 1995, Page web [visitée le 9 janvier 2001], http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html

Netcraft, *The Netcraft Web Server Survey*, décembre 2000, page web [visitée le 7 janvier 2001], http://www.netcraft.com/survey

Office de la langue française. *Le grand dictionnaire terminologique*, Québec, Page web [visitée le 12 janvier 2001], http://www.granddictionnaire.com

OpenBSD, *OpenBSD*, Page web [visitée le 17 janvier 2001], http://www.openbsd.org/

Openwall Project. *Linux kernel patch from the Openwall Project*, Page web [visitée le 8 janvier 2001], http://www.openwall.com/linux/

Page, Bob. *A Report on the Internet Worm*, University of Lowell, Lowell, 7 novembre 1988.

Prym. *finding and exploiting programs with buffer overflows*, Page web [visitée le 7 janvier 2001], http://destroy.net/machines/security/buffer.txt

Rational. *Rational Purify for Unix*, page web [visitée le 8 janvier 2001], http://www.rational.com/products/purify_unix/index.jsp

Raymond, Eric S. *The Cathedral & the Bazaar,* O'Reilly etAssoociates, Cambridge, 1999, 279 pp.

SANS Institute. *Curmudgeon's Executive Summary*, The SANS Network Security Digest, volume 1, numéro 5, 23 juin 1997.

Singh, Navjot et Arash Baratloo et Timothy Tsai. *Transparent Run−Time Defense Against Stack Smashing Attack*, Bell Labs Reseach, Murray Hill, 2000.

Smith, Nathan P. *Stack Smashing Vulnerabilities in the UNIX Operating System*, Southern Connecticut State University, New Haven, 1997.

Sundar, Tirunelveli J. *Unix processe*, Page web [visitée le 10 janvier 2001], http://www.employees.org/~tjsundar/html/tutorials/unixprocesses.html

Taeho Oh. *Advanced buffer overflow exploits*, Page web [visitée le 7 janvier 2001], http://www.securityfocus.com/data/library/advanced.txt

Thomas, Evan. *Attack Class: Buffer Overflows*, avril 2000, Page web [visitée le 10 janvier 2001], http://students.ou.edu/W/Amos.P.Waterland−1/wellspring/buffer_overflow.html

Vendicator, *Stack Shield: A "stack smashing" technique protection tool for Linux*, 2000, Page web [visitée le 10 janvier 2001], http://www.angelfire.com/sk/stackshield/info.html

Wagner, David et Jeffrey S. Foster, Eric A. Brewer et Alexander Aiken. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*, University of California, Berkeley, 2000

WireX Communications, *StackGuard*, Page web [visitée le 7 janvier 2001], http://www.immunix.org/stackguard.html

Wojtczuk, Rafal. *Defeating Solar Designer's Non−executable Stack Patch*, bugtraq@netspace.org, 30 janvier 1998.