Mitigation techniques in OpenBSD

Marc Espie <espie@openbsd.org>

See `https://openbsd.org/events.html`

December 14, 2022

25 ans

OpenBSD se positionnait initialement comme Unix généraliste orienté sécu, mais il s'est replacé comme labo expérimental sur la sécu il y a 10 ans

Mise en pratique

Beaucoup de techniques existent ailleurs mais ce qui fait les caractéristiques d'OpenBSD:

- La mise en œuvre systématique (et le debug afférent!)
- La plupart des mesures sont non débrayables

exemple: les canaris I

```
1  VarName_Get:
2          pushq        %rbp
3          movq         %rsp, %rbp
4          [...]
5          movq         %rdx, %r13
6          pushq        %r12
7          movq         %rdi, %r12
8          pushq        %rbx
9          movq         %rsi, %rbx
10         subq         $88, %rsp
11         movq         %r8, -112(%rbp)
12         movq         __guard_local(%rip), %rax
13         movq         %rax, -56(%rbp)
14         xorl         %eax, %eax
15         [...]
16         movq         -56(%rbp), %rax
17         subq         __guard_local(%rip), %rax
```

exemple: les canaris II

```
18          jne         .L9
19          addq        $88, %rsp
20          movq        %r15, %rax
21          popq        %rbx
22          [...]
23          popq        %r15
24          popq        %rbp
25          ret
26  .L9:
27          leaq        .LSSH0(%rip), %rdi
28          call        __stack_smash_handler@PLT
```

Déployé la première fois, ça a pété

- Xwindows (overflow dans libpixmap)
- ffmpeg (code assembleur "a la mano" avec frame bugguée)

canari2

- Ça a également pété gcc, assez fragile à l'époque.
- entre autres parce qu'on jouait avec l'ordre des variables
- et aussi parce que prologue/epilogue étaient fixes !

```
1
2  void
3  myfunction(const char *buf)
4  {
5          char array[80];
6          int a, b;
7          double f;
8          [...]
9  }
10
```

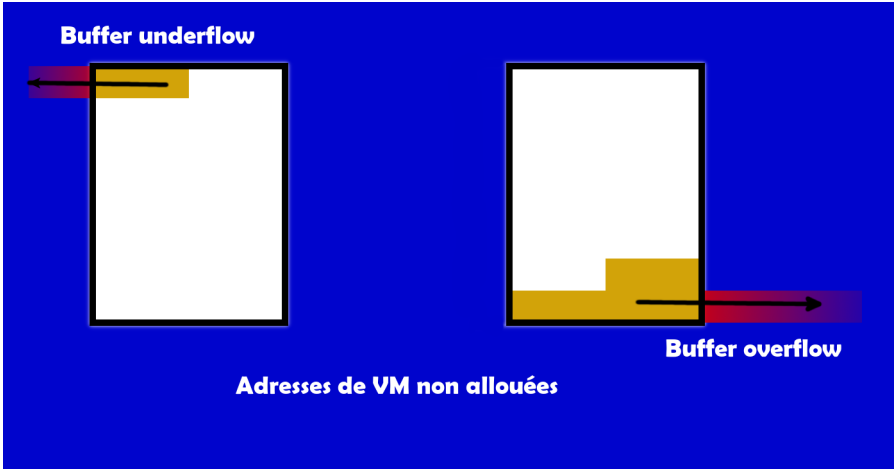
(propolice, cansecwest '03)

exemple 2: les guard pages

Principe:

- on laisse des "trous" dans la vm
- on place des allocations en début et fin de page préférentiellement

(Otto Malloc, eurobsdcon '09)



Évidemment ça pète!

- des buffer overflow, bugs "légitimes"
- mais aussi des underflow en lecture (dont dmake, composant de build d'openoffice

Obligatoire

Si les mesures peuvent être désactivées, elles le sont plutôt que de corriger les bugs chiants. Méfiez-vous des trucs à la GrSecurity/SeLinux, qui lorsqu'ils sont mis en prod ont généralement des listes d'exception longues comme le bras.

Jetez un dé

Pid

les numéros de process sont aléatoires depuis très longtemps (1997) (à cette époque des gens utilisent $$ dans des noms de fichier "aléatoires")

Espace mémoire

- l'emplacement exact de la pile est aléatoire
- le chargement des bibliothèques partagées aussi
- les exécutables également (PIE) (et non plus à l'adresse 0 en mémoire)

Problèmes de détail

- `mmap` à une adresse fixe ne marche pas (préchargement d'entêtes précompilées)
- certains bug sont difficiles à reproduire
- il y a beaucoup de sections dans un exécutable: ça casse certains linkers, voire emacs!
- On a essayé d'utiliser au mieux un espace mémoire 64 bits: ça casse les navigateurs web et la jvm

Aléatoire réel

- `arc4random` est disponible depuis une éternité (n'utilise plus rc4 mais bien chacha)
- appel système `getentropy` et pas `/dev/random` pour éviter de "perdre" l'aléatoire lors d'un `chroot`
- on sauvegarde l'aléatoire d'un boot au suivant

(Theo de Raadt, Hackfest 2014)

Aléatoire d'une fois à l'autre

- la libc et la libcrypto sont relinkées à chaque boot
- le noyau est relinké pour le boot suivant
- après un `fork` les 2 process ont la même carto mémoire, on peut faire un re-exec du même programme pour forcer la randomisation (par exemple sshd)

(Theo de Raadt, CUUG 2019)

De l'aléa partout

- on a changé `rand` pour être aléatoire (mais avec une possibilité de le remettre en déterministe, nécessaire pour certains jeux de test)
- `mkstemp` et consorts utilisent des chaines plus longues que ce que demande POSIX: 10 X plutôt que 6.

L'attaque progresse

Exécution

Les premiers exploits utilisent un buffer overflow pour poser du code exécutable → première contre-mesure, pile non exécutable.
Ça casse certains bouts de gcc (l'extension trampoline) et donc certains programmes, dont asterix, plus changement de la gestion des signaux (faite là aussi par un trampoline posé par le noyau)

Et overflows

Première contre-mesure déjà vue: les canaris, mais qui va évoluer au fil du temps

L'attaque progresse 2

Quid du tas

Les allocateurs modernes sont très vulnérables (si on peut choisir la taille de l'allocation): rendre un maximum de choses soit écrivables, soit exécutables mais pas les deux $\rightarrow$ le fameux $W \wedge X$
En particulier, ça complexifie fortement le layout des bibliothèques et des exécutables.

(Theo de Raadt, OpenCon 2005)

Exemple de protection: kbind

- optimisation nécessaire le "lazy binding"
- nécessite des pages en écriture et exécution (pour PLT)
- appel système `kbind`
  - qui passe "en dessous" de la mémoire virtuelle
  - qui est attaché à la première adresse mémoire qui l'appelle (`ld.so`)

L'attaque progresse 3

"Return Oriented Programming"

Confere David Mazières et al "Hacking blind" Le principe:

- on utilise un buffer overflow pour empiler plusieurs adresses de retour
- chaque adresse pointe sur un petit fragment de code (un gadget) qui fait un bout du travail
- mis bout à bout les fragments font ce qu'il faut: mettre les bons paramètres dans chaque registre (souvent depuis la pile)
- puis appeler l'appel système approprié (le plus souvent exec)
- plus facile sur intel qui a des instructions de taille variable !

Aligned Gadget

Terminates on an intended return instruction

```
Gadget: 0xffffffff81820653 : pop rbp ; ret // 5dc3

ffffffff81820653: 5d      popq    %rbp

ffffffff81820654: c3      retq
```

Polymorphic Gadget

Terminates on an unintended return instruction

```
Gadget: 0xffffffff810f72dc : pop rbp ; ret // 5dc3

ffffffff810f72db: 8a 5d c3   movb    -61(%rbp), %bl
```

Les canaris sous stéroïdes

- chaque fonction a un canari différent (section `OpenBSD-random-data`)
- le canari est xor'd avec l'adresse de retour $\rightarrow$ on détecte les overflows et les changements d'adresse de retour (retguard)

Éviter les nop-slides

- Une fonction usuelle contient des nop d'alignement
- On remplace les nop par un jump + une collection de traps
- Si on saute au hasard, on tombe plus facilement sur un trap, plus dur de trouver les gadgets

Défense 2

Enlever les gadgets

- sur intel 64 bits, changer l'ordre d'affectation des registres élimine certains gadgets
- ne pas générer certaines séquences d'instruction

(Todd Mortimer, Eurobsdcon 2018 et AsiaBSDcon 2019)

Interdire les appels système

- approche similaire à `kbind`: limiter l'endroit d'où certaines opérations peuvent être utilisées
- on a introduit `msyscall` qui permet de "sanctuariser" une plage d'adresse en dehors de laquelle tout appel système est interdit
- le linker dynamique s'en sert: les appels systèmes doivent passer par la libc (qui est randomisée, qui a des canaris)

Défense en cours!

- il y a des utilisations légitimes de pages en écriture **puis** en exécution: la génération de code (JIT)
- interdire d'avoir des pages qui ont les 2 modes. Flag sur le filesystem autorisant quand même cela car c'est difficile de nettoyer tout upstream
- problème avec les langages dynamiques comme python
- `msyscall` pose problème avec les langages qui refont tout comme rust ou go
- Nouveau mécanisme en cours de déploiement: mimmutable

pledge

- principe connu: limiter les opérations système que peut faire un process (par exemple: SElinux, capsicum, systrace, capabilities)
- **mais** pledge a 3 particularités:
  - les violations sont entièrement détectées de façon synchrone par le noyau
  - les «capabilities» sont fonctionnelles: souvent à très gros grain, mais parfois descendent plus bas que l'appel système
  - il est intégré au système et utilisable sans erreur par des «non spécialistes»

exemple

- `ioctl` a une surface d'attaque délirante
- → il existe des pledges pour des sous-systèmes (`tty`, `audio`) qui autorisent uniquement certains paramètres pour `ioctl`.
- Le réseau comporte plein de sous-systèmes
- → il y a un pledge spécifique pour le DNS et un type de socket prévu pour la fonctionnalité en question
- → et d'autres pledges pour d'autres sous-systèmes similaires.

bilan sur pledge

- la libc a connu beaucoup de petites modifs visant à rendre pledge aussi transparent que possible
- la majorité des utilitaires du système de base utilise pledge.
- les valeurs possibles ont permis d'appliquer pledge à des programmes aussi gros que firefox ou chrome

en conclusion

- Je me suis volontairement limité aux mesures techniques déployées dans OpenBSD, et j'en ai omises certaines
- Il y a aussi tout un "savoir-faire" (best practices), avec des API plus propres, des warnings de la chaine de compile, des techniques d'écriture de code (privilege separation, signal handlers, audit en profondeur)
- ça pourrait faire un autre séminaire entier
- Merci de votre attention (questions ?)

(**biblio**: `https://openbsd.org/events.html`)