Programmation défensive

Matthieu Herrb

TLS-Sec, 10 octobre 2023

http://homepages.laas.fr/matthieu/cours/mh-prog-defensive.pdf

Licence

Agenda

Agenda

Introduction

Ce cours devrait être inutile...

Mais :

- avec les langages de programmation courants
  $\rightarrow$ erreurs inévitables
- limites/coût des méthodes formelles

$\Rightarrow$ techniques de programmation pour limiter l'impact des erreurs

ou bien: changer de langage (Rust) ...

Contexte

- Programmation « système » :
  - noyau,
  - démons réseau,
  - outils bas-niveau
  - systèmes embarqués

  langage : essentiellement C
- Expériences acquises par le projet OpenBSD
- Reprises ailleurs, généralisables

Règles de codage

- Correction du code d'abord
  $\rightarrow$ meilleure fiabilité $\rightarrow$ meilleure sécurité.
- Conception recherchant la simplicité (« KISS »).
- Principe de revue par les pairs à tous les niveaux
- Recherche systématique des erreurs
- Fonctionnalités avancées de gcc/clang :
  - option `-Wbounded` (vérifie longueur buffer),
  - attribut `__sentinel__` (vérifie fin varargs)
  - retguard (protection améliorée de la pile)
- Outils : clang static analyser, Coverity, Parfait, Syzkaller, etc.

Mitigations

Modifications peu coûteuses qui affectent les attaques en les rendant plus difficiles / moins efficaces

- Sûr par défaut (configuration initiale)
- Échec clair et efficace en cas d'anomalie
- Adaptation aux évolutions des systèmes
- Durcissement de l'environnement d'exécution

Technologies pour la sécurité

- API plus sûres
- protection de la mémoire
- protection de la pile (SSP) & Stackgap
- introduction d'aléas (ld.so, malloc, mmap)
- révocation des privilèges (ex. `ping`)
- séparation des privilèges (ex. OpenSSH)
- mise en cage (`chroot`)
- uids distincts par service
- restrictions fonctionnelles (*bac à sable*)

Agenda

Bibliographie

- *Better Embedded System Software*, P. Koopman, Drumnadrochit Education LLC, 2010 ISBN: 978-0-9844490-0-2
- *Secure Coding in C and C++ 2nd edition*, Robert C. Seacord, Addison Wesley, 2013. ISBN: 978-0-321-82213-0.

`https://www.openbsd.org/innovations.html`
`https://www.openbsd.org/events.html` :

- *How OpenBSD's malloc helps the developer*, Otto Moerbeek, EuroBSDCon, Coimbra, 2023
- *Unveil in OpenBSD*, Bob Beck, BSDCan, Ottawa, 2019
- *Synthetic Memory Protections, An update on ROP mitigations*, Theo de Raadt, CanSecWest, Vancouver 2023
- *Pledge, Where did it come from ?*, Theo de Raadt, EuroBSDCon 2017, Paris, 2017
- *arc4random - randomization for all occasions*, Theo de Raadt, Hackfest 2014, Quebec City.
- *Security Measures in OpenSSH*, Damien Miller Asia BSD Conference 2007
- *Preventing Privilege Escalation*, Niels Provos, Markus Friedl and Peter Honeyman, 12th USENIX Security Symposium, Washington, DC, August 2003.

Agenda

API plus sûres

- beaucoup d'API ne sont pas pensées pour la sécurité
  - → risques d'erreurs non détectées à l'exécution
  - → difficiles à utiliser
- proposer des remplacements plus sûrs
- encourager l'abandon des API dangereuses
- ne pas refaire ces erreurs :
  attention à la sécurité dans les nouvelles API

Manipulation de chaînes de caractères en C

Problème numéro un : débordement de buffer

```
strcpy(path, getenv("$HOME"));
strcat(path, "/");
strcat(path, ".foorc");
len = strlen(path);
```

API Dangereuses

- `strcat()`, `strcpy()` : API dangereuse, aucune vérification sur la taille des buffers → À Bannir !

```
NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);

       char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
       The  strcpy()  function  copies the string pointed to by src, including
       the terminating null byte ('\0'), to the buffer  pointed  to  by  dest.
       The  strings  may  not overlap, and the destination string dest must be
       large enough to receive the copy.

       The strncpy() function is similar, except that at most n bytes  of  src
       are  copied.  Warning: If there is no null byte among the first n bytes
       of src, the string placed in dest will not be null-terminated.

       If the length of src is less than n, strncpy() pads  the  remainder  of
       dest with null bytes.
```

API Dangereuses (2)

- variantes « n » (**strncat()**, **strncpy()**) très difficiles (impossible ?) à utiliser correctement : laissent les chaînes non terminées par nul.

```
strncpy(path, homedir, sizeof(path) - 1);
path[sizeof(path) - 1] = '\ 0';
strncat(path, "/", sizeof(path) - strlen(path) - 1);
strncat(path, ".foorc", sizeof(path) - strlen(path) - 1);
len = strlen(path);
```

Remplacement

Solution proposée par Todd Miller (1999): **`strlcpy()`** et **`strlcat()`** : tronquent les chaînes trop longues mais garantissent que le résultat est toujours une chaîne valide. Possible de tester si troncation a eu lieu.

NAME
strlcpy, **strlcat** - size-bounded string copying and concatenation

SYNOPSIS
#include <string.h>

size_t
strlcpy(char *dst, const char *src, size_t size);

size_t
strlcat(char *dst, const char *src, size_t size);

DESCRIPTION
The **strlcpy**() and **strlcat**() functions copy and concatenate strings respectively. They are designed to be safer, more consistent, and less error prone replacements for strncpy(3) and strncat(3). Unlike those functions, **strlcpy**() and **strlcat**() take the full size of the buffer (not just the length) and guarantee to NUL-terminate the result (as long as size is larger than 0 or, in the case of **strlcat**(), as long as there is at least one byte free in dst). Note that a byte for the NUL should be included in size. Also note that **strlcpy**() and **strlcat**() only operate on true ``C'' strings. This means that for **strlcpy**() src must be NUL-terminated and for **strlcat**() both src and dst must be NUL-terminated.

strl* : exemple

Exemple simple sans vérifications :

```
strlcpy(path, homedir, sizeof(path));
strlcat(path, "/", sizeof(path));
strlcat(path, ".foorc", sizeof(path));
len = strlen(path);
```

→ risque de troncation, mais pas de débordement

strl* : exemple avec vérifications

```
len = strlcpy(path, homedir, sizeof(path);
if (len >= sizeof(path))
    return (ENAMETOOLONG);
len = strlcat(path, "/", sizeof(path);
if (len >= sizeof(path))
    return (ENAMETOOLONG);
len = strlcat(path, ".foorc", sizeof(path));
if (len >= sizeof(path))
    return (ENAMETOOLONG);
```

printf() est *Turing-complete* !

Bannir `%n` dans les chaînes de format des fonctions **`printf()`** like :

```
printf("%s%n\n", str, &len);
```

à remplacer par :

```
len = printf("%s\n", str);
```

Référence : *Control-Flow Bending: On the Effectiveness of Control-Flow Integrity*, N. Carlini, A. Baresi, M. Payer et D. Wagner, Proceedings of the 24th USENIX Security Symposium, août 2015.

→ Brainfuck interpreter in printf...

TOCTOU

Time Of Check, Time Of Use

Comment créer un fichier temporaire dans `/tmp` sans écraser un fichier existant ?

```
/* Generate random file name */
name = mktemp("/tmp/my-temp-file.XXXXXXXX");
/* verify that it is non-existant */
if (stat(name, &statbuf) == 0) {
  return EEXISTS;
}
/* Good, open it */
fd = open(name, O_RDWR);
```

→ pas bon !

TOCTOU - remèdes

mkstemp() remplace **mktemp()** et **open()** de manière sûre.

```
fd = mkstemp("/tmp/my-temp-file.XXXXXXXX");
```

Autre solution: le flag *O_EXCL* de **open()** qui produit une erreur si le fichier existe déjà:

```
fd = open(name, O_RDWR | O_CREAT | O_EXCL);
```

Débordements arithmétiques

Allouer n structures d'un type donné :

```
n = getIntFromUser();
if (n <= 0)
    return EINVAL;
data = (struct foo *)malloc(n*sizeof(struct foo));
if (data == NULL)
    return ENOMEM;
```

Si n est assez grand $\rightarrow$ débordement et allocation d'une zone mémoire trop petite, puis débordement mémoire.

Débordement arithmétiques (2)

Utiliser `calloc()` :

```
data = (struct foo *)calloc(n, sizeof(struct foo));
if (data == NULL)
    return ENOMEM;
```

Pas de débordement arithmétique possible.
Si *n* trop grand, calloc retournera `NULL`.

Débordements arithmétiques (3)

Nouvelle proposition **`reallocarray()`** :

- pas de mise à zéro de la mémoire
- traite le débordement pour l'API `realloc()`

```
void *
reallocarray(void *ptr, size_t nmemb, size_t size);
```

```
data = (struct foo *)reallocarray(NULL, n, sizeof(struct foo));
if (data == NULL)
    return ENOMEM;
```

Débordements arithmétiques (4)

```
n = getIntFromUser();
if (n <= 0 || n*sizeof(struct foo) > MAX_BUF_SIZE)
   return EINVAL;
```

$\rightarrow$ si n est assez grand, la condition testée va être fausse.

Utiliser :

```
if (n <= 0 || n > MAX_BUF_SIZE/sizeof(struct foo))
    return EINVAL;
```

Protéger les secrets

- `explicit_bzero()` - garanti la mize à zéro d'un buffer

```
char pass[MAX_PASS_LEN];
readpassphrease("password:", pass, sizeof(pass), 0);
if (valid(pass)) {
    explicit_bzero(pass, sizeof(pass));
    ...
```

- `malloc_conceal()` - garanti la destruction des données

```
char *pass = malloc_conceal(MAX_PASS_LEN);
readpassphrease("password:", pass, sizeof(pass), 0);
if (valid(pass)) {
    free(pass);
    ...
```

Faire des sockets TLS avec OpenSSL est très pénible...
`libtls` rend cela plus simple avec moins de risques d'erreur :

- nouvelle API
- masque de nombreux détails d'implémentation (pas de structures ASN.1, x509,...)
- comportement par défaut sûr (vérification hostnames/certificats,...)
- privilege separation friendly
- exemples d'utilisation : OpenSMTPd, relayd, httpd...
- toujours en cours de développement

Happy Bob's libtls tutorial

Agenda

Aléas

Générateur pseudo-aléatoire (`getentropy()` & `arc4random()`) :

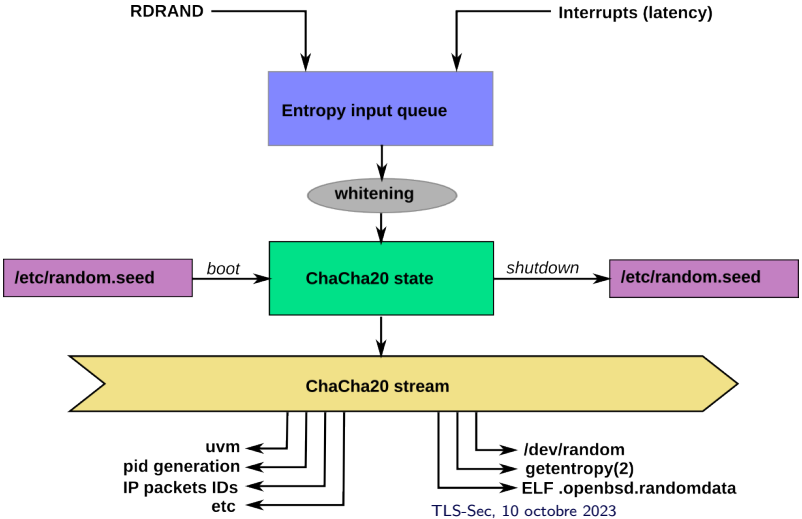
- non prédictible
- simple d'utilisation
- robuste
- disponible aussi tôt que possible

```
uint32_t
arc4random(void);

void
arc4random_buf(void *buf, size_t nbytes);

uint32_t
arc4random_uniform(uint32_t upper_bound);
```

Générateur d'aléa d'OpenBSD : noyau



Agenda

Applications privilégiées

Applications qui ont besoin des privilèges de `root`

- ouverture de ports TCP/UDP < 1024
- écriture dans répertoires système
- allocation de pseudo-tty
- accès à des périphériques à accès restreint

Deux grandes familles d'applications privilégiées :

- démons système démarrés par `root`
- programmes avec le bit *setuid*

Privilèges et vulnérabilités

Une erreur dans un programme privilégié a un impact plus élevé

- exécution du code malicieux avec privilèges

Réduire les privilèges :

- si plus nécessaires, révocation définitive après initialisation
- sinon : *séparation des privilèges*

Réduction des privilèges

Révoquer définitivement les privilèges des commandes privilégiées (setuid) ou démons lancés avec privilège (par root au démarrage) une fois que toutes les opérations nécessitant un privilège sont effectuées.
Grouper ces opérations le plus tôt possible.
Exemples:

- ping
- named

/usr/src/sbin/ping/ping.c

```
int
main(int argc, char *argv[])
{
    /* Declarations ... */

    if ((s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0)
            err(1, "socket");

    /* revoke privs */
    uid = getuid();
    if (setresuid(uid, uid, uid) == -1)
            err(1, "setresuid");
    ....
}
```

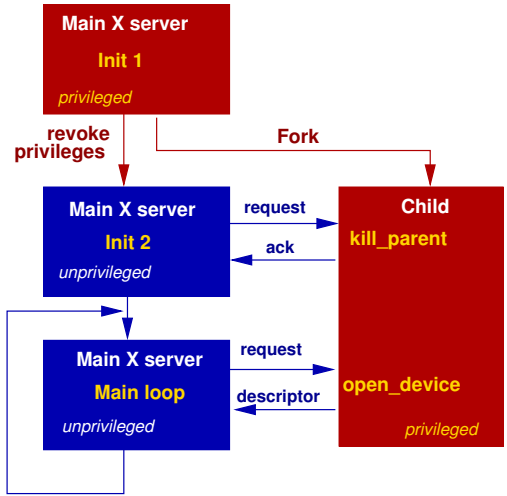
Séparation des privilèges

[Provos 2003]

- exécuter les démons système
  - avec un uid $\neq 0$
  - dans une cage chroot(2)
- processus additionnel d'aide qui reste privilégié mais vérifie de manière paranoïaque toutes ses actions.
- passage de descripteurs de fichiers ouverts entre processus via `imsg_*()`

Une douzaine de démons ainsi protégés (sshd, ntpd, bgpd,...)

Exemple : serveur X



https://www.openbsd.org/cgi-bin/cvsweb/xenocara/xserver/os/privsep.c

Bac à sable : pledge(2) et unveil(2)

- Permet de limiter les accès d'une application
  - appels système (`pledge()`)
  - système de fichiers (`unveil()`)
- Intégré au code source des applications (par les développeurs)
- Modes d'échec clairs et non programmables (`abort()` ou `ENOENT`)
- Fonctionnellement plus ou moins équivalent à seccomp ou ebpf sur Linux

Pledge

Restreint les appels système autorisés pour un processus

```
int
pledge(const char *request, const char *paths[]);
```

request est une liste de mots clés caractérisant le comportement normal du processus et les appels système autorisés.

Le comportement de certains appels système est limité
(par ex. `ioctl()`)

Les autres appels système sont interdits.

Pledge(2)

Quelques requètes :

`""`	seul `_exit` est autorisé
`stdio`	appels systèmes les plus courants de la libc, y compris allocation mémoire, mais sans ouverture de nouveaux fichiers
`rpath`	accès en lecture à des fichiers
`wpath`	accès en écriture à des fichiers
`tmppath`	lecture/écriture/création de fichiers dans `/tmp`
`inet`	accès sockets IPv4 et IPv6
`dns`	sous-ensemble de `inet` pour résolution DNS
`tty`	manipulation du terminal (*termios*)
`proc exec`	fork / exec

Pledge : exemple wc.c

```
int
main(int argc, char *argv[])
{
        int ch;

        setlocale(LC_ALL, "");

        if (pledge("stdio rpath", NULL) == -1)
                err(1, "pledge");

        while ((ch = getopt(argc, argv, "lwchm")) != -1)
        ...
```

unveil(2)

Dévoile une partie d'un système de fichier

```
int unveil(const char *path, const char *permissions);
```

- le premier appel masque tout le système de fichiers, sauf pour *path* avec *permissions*
- les appels suivants dévoilent des bouts supplémentaires
- `unveil(NULL, NULL);` fige définitivement la liste.

Permissions :

r	`rpath`
w	`wpath`
x	`exec`
c	`cpath`

Agenda

Conclusion

- Erreurs de programmation $\rightarrow$ impact potentiel sur la sécurité
- Impossible de les éliminer à 100 %
- Adopter des bonnes pratiques pour limiter les risques d'erreurs
  - API plus sûres
  - Limiter les privilèges
  - Faire relire son code

Questions ?