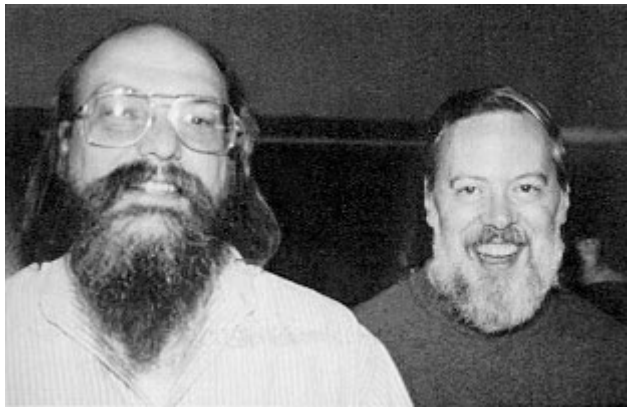
Les Vulnérabilités du Processus de Compilation

Raphaël Khoury

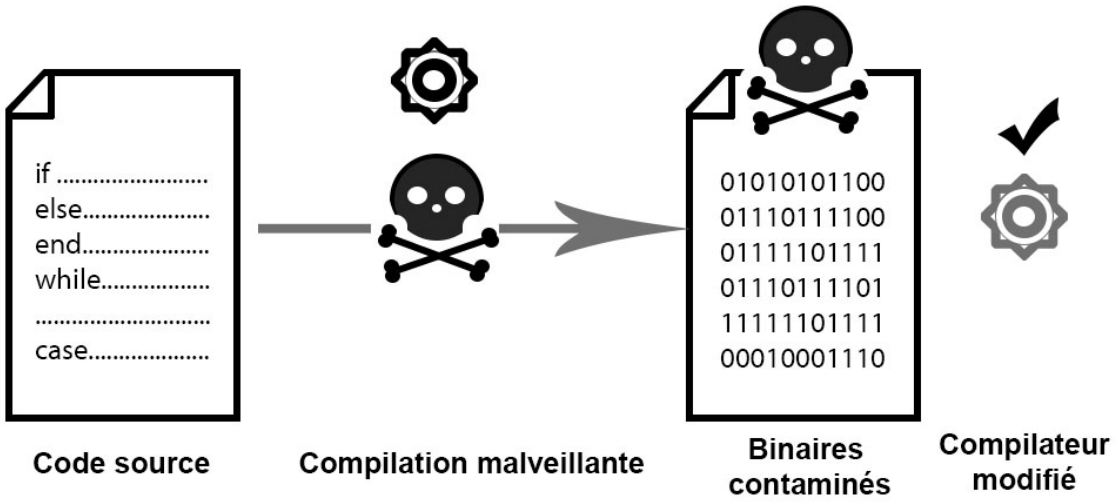
UQAC
Université du Québec
à Chicoutimi

Reflections on Trusting Trust

- Dans son allocution après avoir obtenu le *Turing Award*, Ken Thompson propose d'écrire un programme qui se modifie lui-même, après avoir effectué une attaque.
- Sur quelles hypothèses repose notre confiance?



Reflections on Trusting Trust



Plan:

- Définition du problème
- Libération des ressources
- *Overflow* d'entier
- Déréférencer le pointeur Null
- Enfreinte au contrat d'utilisation
- Utilisation de mémoire non-initialisée
- Ordre d'application des optimisations
- Contremesures

Problématique:

- Les compilateurs sont de plus en plus agressifs dans les optimisations qu'ils effectuent sur le code, pour améliorer l'efficacité du programme en temps et en espace.
- Ces optimisations peuvent interférer avec la capacité du développeur à prévoir le comportement déterministe de son programme.
- Deux concepts sont critiques à la compréhension de ce phénomène:
  - La spécification du langage
  - Les comportements non-définis

Les Vulnérabilités et la Compilation

Le comportement du compilateur est limité par la spec. du C ou du C++, qui forme un « contrat » entre le programmeur et le compilateur. Elle spécifie le comportement du langage de programmation d'une manière très détaillée.

C99 §6.5.6p8:

When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand.

Les Vulnérabilités et la Compilation

- Certains comportements sont dits: « non-définis »
  - **Shall ou shall not**
  - **Undefined behavior**
  - Omise dans la spec.
- Ces trois descriptions sont interchangeables.

- **C'est la tâche du programmeur de s'assurer que le comportement non-définis ne se produit pas; le compilateur suppose qu'il est impossible.**

Les Vulnérabilités et la Compilation

Exemples de comportements non-définis:

- Déréférencer le pointeur NULL
- Libérer la même mémoire deux fois
- Écrire hors des bornes d'un buffer
- Utiliser de la mémoire non initialisée
- L'overflow sur des entiers **signés**
- La division par 0
- Modulo d'un nombre négatif
- …

Les Vulnérabilités et la Compilation

- **Le compilateur suppose que le comportement non-défini est impossible, mais ne teste pas ce fait. S'il se produit, le compilateur pourrait:**
  - Agir d'une manière imprévisible
  - Arrêter l'exécution
  - Gérer le comportement non-défini d'une manière spécifiée et documentée.
  - "*When the compiler encounters [undefined behavior] it is legal for it to make demons fly out of your nose.*"—John Woods

Les Vulnérabilités et la Compilation

Deux autres cas:

- **Comportement non-spécifié:** Comportement pour lequel la spécification permet plus d'un comportement possible. Ex. malloc(0) peut retourner NULL, ou un espace mémoire inutilisable.
- **Comportement spécifique à l'implantation**: Un comportement spécifié dans le contexte d'une implantation spécifique. Ex. : sizeof(int)

Les Vulnérabilités et la Compilation

La règle « *as if* » impose au compilateur que le résultat final d'un calcul soit le même, malgré toutes optimisations. Il n'impose aucune restriction sur la manière d'atteindre ce résultat. Une étape intermédiaire peut être éliminée si

- elle n'affecte pas le résultat final, et
- elle n'engendre aucun effet de bord.

Les Vulnérabilités et la Compilation

Définition du problème

Exemple:

```
if(arg2 == 0)
        ereport(ERROR,…);
return ((int32) arg1 / arg2);
```

On a omis d'indiquer au compilateur que `ereport()` ne retourne pas. Il croit donc que la division a toujours lieu. Comme la division par 0 est non-définie, le compilateur peut supposer qu'elle est impossible. Il peut donc éliminer le test.

Les Vulnérabilités et la Compilation

La libération des ressources

- Une optimisation commune est : **l'élimination du code mort.**
- Habituellement, une écriture à une variable qui n'est plus lue est vue comme étant du code mort et est éliminée.

```
int foo (int x, int y)
{
    int z = x/y;
    return x*y;
}
```

←La division et l'affectation ne seront pas présentes dans le code compilé.

Les Vulnérabilités et la Compilation

La libération des ressources

```
void getPassword(void) {
      char pwd[64];
      if (GetPassword(pwd, sizeof(pwd)))  {
      …//vérification
      }
      memset(pwd, 0, sizeof(pwd));
}
```

Cas présent dans Tar.

Les Vulnérabilités et la Compilation

La libération des ressources

Solution: ZeroMemory()

- La fonction ZeroMemory est présente dans la plupart des environnements d'exécution MS.

```
void getPassword(void) {
      char pwd[64];
      if (GetPassword(pwd, sizeof(pwd)))     {
      …//vérification
      }
      ZeroMemory(pwd, sizeof(pwd));
}
```

Pourrait être optimisée tout comme memset();

Les Vulnérabilités et la Compilation

La libération des ressources

Solution: SecureZeroMemory()

- La fonction SecureZeroMemory ne peut pas être éliminée par les optimisations. (windows.h)
- Depuis C++11, plusieurs implantations ont memset_s, avec la même fonctionnalité.

```
void getPassword(void) {
    char pwd[64];
    if (GetPassword(pwd, sizeof(pwd)))  {
    …//vérification
    }
    SecureZeroMemory(pwd, sizeof(pwd));
}
```

Les Vulnérabilités et la Compilation

La libération des ressources

Solution: #pragma

```
void getPassword(void) {
      char pwd[64];
      if (GetPassword(pwd, sizeof(pwd)))     {
      …//vérification
      }
      #pragma optimize("", off)
      SecureZeroMemory(pwd, sizeof(pwd));
      #pragma optimize("", on)
}
```

Les Vulnérabilités et la Compilation

La libération des ressources

```
void getPassword(void) {
      char pwd[64];
      if (GetPassword(pwd, sizeof(pwd)))  {
      …//vérification
      }
      memset(pwd, 0, sizeof(pwd));
      *(volatile char*)pwd = *(volatile char*)pwd;
}
```

Sur certaines implémentations, seulement le premier byte sera protégé.

Les Vulnérabilités et la Compilation

La libération des ressources

Pour une défense en profondeur, initialisez à zéro un buffer ou un objet qui sera envoyé sur le réseau avant d'y copier des données.

Les Vulnérabilités et la Compilation

Exemple: JpegOfDeath

Une vulnérabilité présente dans le code qui gère l'affichage des jpg sur Windows XP et 2003 (GDI+). Le fichier contient une sdd de commentaires contenant une string et un short (2 bytes) contenant la taille des commentaires (**incluant le short lui-même**). Le logiciel lit ce champ, soustrait 2 et lit ensuite une string de la taille désirée et la copie dans un buffer de cette taille +1.

Remarquer le point de vulnérabilité commun à tous les logiciels qui lisent les jpeg.

Les Vulnérabilités et la Compilation

Exemple: JpegOfDeath

```
void getComment(unsigned int len, char *src)
{
      unsigned int size;
      size = len - 2;
      char *comment = (char *)malloc(size + 1);
      memcpy(comment, src, size);
      return;
}
```

Les Vulnérabilités et la Compilation

Exemple: JpegOfDeath

```
void getComment(unsigned int len, char *src)
{
unsigned int size;
size = len - 2; ←si len =1, size = 0xffffffff
char *comment = (char *)malloc(size + 1); ←malloc 0
memcpy(comment, src, size);  ←vulnerabilité unlink
return;
}
```

Les Vulnérabilités et la Compilation

L'overflow d'entiers

Considérez le code suivant:

```
int *ptr; // début du tableau
int *max; // fin du tableau
int len;
if (ptr + len > max)
   return ERROR;
```

Le programmeur a inséré ce test pour s'assurer de ne pas écrire hors du tableau.

Les Vulnérabilités et la Compilation

L'overflow d'entiers

```
int *ptr; // début du tableau
int *max; // fin du tableau
size_t len;
if (ptr + len > max)
   return ERROR;
```

Si le tableau est grand `ptr + len` peut dépasser la taille max des int et causer un overflow d'entier.

Les Vulnérabilités et la Compilation

L'overflow d'entiers

Est-ce que le code suivant est une solution adéquate (modifié de Linux)?

```
if (ptr + len < ptr || ptr + len > max)
      return ERROR;
```

Le compilateur pourra éliminer le premier test. ptr + len peut seulement être plus petit que ptr si un comportement non-défini se produit.

Les Vulnérabilités et la Compilation

L'overflow d'entiers

Est-ce que le code suivant est une solution adéquate?

```
if (ptr + len < ptr || ptr + len > max)
      return ERROR;
```

Les programmeurs de Linux ont fait une plainte à gcc, et le CERT a émis une notice de vulnérabilité contre gcc (Vulnerability Note VU#162289), mais le comportement du compilateur est correcte.

Les Vulnérabilités et la Compilation

L'overflow d'entiers

Le même problème se présente dans ce fragment de code:

```
int f(int *buf, size_t n) {
      return buf + n < buf + 100;}
```

Il sera optimisé à

```
int f(int *buf, size_t n) {
      return n < 100; }
```

Cette situation est suffisamment fréquente pour avoir sa propre entrée CWE:
CWE-733: Compiler Optimization Removal or Modification of Security-critical Code

Les Vulnérabilités et la Compilation

L'overflow d'entiers

Le test suivant règlerait le problème:

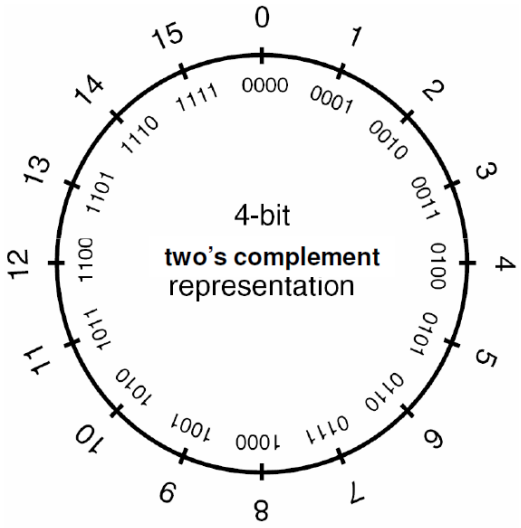
```
if (len > max – ptr)
 return Error;
```

- Utiliser des types non-signés donne aussi un plus grand degré de sécurité.
- **rsize_t** (C11) entier non-signé, garantit d'être au plus aussi grand que RSIZE_MAX (défini dans la spec.) Une exception est lancée si la valeur dépasse RSIZE_MAX.
- **ptrdiff_t** est un type défini pour les opérations sur les pointeurs. Un overflow est encore possible si on compare des pointeurs de tableaux différents, ou d'un tableau plus grand que la constante PTRDIFF_MAX.

Les Vulnérabilités et la Compilation

L'overflow d'entiers

Dans le cas des entiers **non-signés**, le même comportement est permis et bien défini. On le désigne « **wraparound**» ou « **wrap** », il s'agit d'un comportement fréquemment utilisé dans le comportement normal des programmes. Le compilateur ne peut pas le supposer impossible.

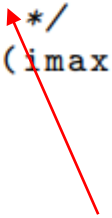


Les Vulnérabilités et la Compilation

```
#define IA 1103515245u
#define IC 12345u
#define IM 2147483648u

static unsigned int c_rand = 0;

/* Creates a random integer [0...imax] (inclusive) */
int my_irand (int imax) {
  int ival;
  /* c_rand = (c_rand * IA + IC) % IM; */
  c_rand = c_rand * IA + IC; // Use overflow to wrap
  ival = c_rand & (IM - 1);  /* Modulus */
  ival = (int) ((float) ival * (float) (imax + 0.999)
       / (float) IM);
  return ival;
```



L'usage du wraparound est bien documenté. Ce code est correcte, mais non-portable, et retournera des valeurs erronées dans un contexte où les valeurs max. des int diffèrent. Comment tester l'aléatoire?

Les Vulnérabilités et la Compilation

Pointeurs NULL

- Un pointeur NULL représenté par un vecteur de 0 (0x00000000).
- Comme 0x00000000 n'est pas une adresse valide, et le pointeur NULL ne pointe pas vers un objet réel, une tentative de déréférencer un pointeur NULL est un comportement non-défini et lance habituellement une exception.
- La spec. du C++ garantit qu'un pointeur NULL sera égal à 0 si on le compare à un int, et que deux pointeurs NULL seront égaux.

Les Vulnérabilités et la Compilation

Pointeurs NULL (Cas réel dans Linux)

Le compilateur va supprimer le test du pointeur NULL s'il est certain du résultat.

```
void bad_code(void *a) {
    int *b = a;
    int c = *b; ← si *b est NULL, une exception ici
    static int d;
    if (b){← alors le compilateur va enlever ce test
    d = c;
}
```

Cette optimisation peut être éteinte par

`-fno-delete-null-pointer-checks`

Car certains syst. (ex. machine virtuelle) permettent l'adresse 0. Des attaques utilisent ce fait.

Les Vulnérabilités et la Compilation

Erreur dans l'utilisation de memcpy (gzip):

```
/* (this test assumes unsigned comparison) */
if (w - d >= e)
{
  memcpy(slide + w, slide + d, e);
  w += e;
  d += e;
}
```

Le comportement de `memcpy` est non-défini si les buffers se chevauchent. Le programmeur a ajouté un test pour détecter ce cas. Une fonction de copie plus lente est appelée si le test échoue.

e est le nombre de bytes copiés, w et d sont des offsets pour accéder à la mémoire.

Les Vulnérabilités et la Compilation

Erreur dans l'utilisation de memcpy (gzip):

```
/* (this test assumes unsigned comparison) */
if (w - d >= e)
{
  memcpy(slide + w, slide + d, e);
  w += e;
  d += e;
}
```

Si d >w, un wraparound se produit et le test est passé malgré un chevauchement.

La vulnérabilité est restée 17 ans dans gzip.

Les Vulnérabilités et la Compilation

Erreur dans l'utilisation de memcpy (gzip):

La solution initiale fut d'ajouter ce test:

```
if (d < w && w - d >= e)
```

Ce test fonctionne, mais est sous-optimal, et la fonction plus lente est parfois appelée même si memcpy est sans risque. La solution correcte est:

```
unsigned int delta = w > d ? w - d : d - w;
if (delta >= e)
```

Les Vulnérabilités et la Compilation

L'utilisation d'une variable non initialisée

Le compilateur est autorisé à remplacer la valeur et **toute valeur qui lui est dérivée**, par une valeur arbitraire.

Code de FreeBSD libc (simplifié):

```
unsigned long junk;

randomSeed( junk ^ getpid()^ gettimeofday());
```

Les Vulnérabilités et la Compilation

L'ordre d'application des optimisations

Même l'ordre dans lequel les optimisations sont effectuées affecte le code produit.

```
void fonction(int *p) {
  int dead = *p;
  if (p == 0)
    return;
  *p = 4;
}
```

Les Vulnérabilités et la Compilation

L'ordre d'application des optimisations

Si le compilateur effectue l'optimisation «élimination du code mort » avant « élimination des *null check* redondants»:

```
void fonction(int *p) {
  int dead = *p;
  if (p == 0)
    return;
  *p = 4;
}
```

←Code mort, car dead n'est jamais utilisé

Le test `(p==0)` est préservé. Aucune autre optimisation n'est possible.

Les Vulnérabilités et la Compilation

L'ordre d'application des optimisations

Si le compilateur effectue l'optimisation « élimination des null check redondants» avant « élimination du code mort», l'optimisation se fait en 2 étapes:

```
void fonction(int *p) {
  int dead = *p;
  if (false)
    return;
  *p = 4;
}
```

←Le test est supprimé, car p vient d'être déréférencé

Les Vulnérabilités et la Compilation

L'ordre d'application des optimisations

Si le compilateur effectue l'optimisation « élimination des *null check* redondants» avant « élimination du code mort», l'optimisation se fait en 2 étapes:

```
void fonction(int *p) {
  int dead = *p;
  if (false)
    return;
  *p = 4;
}
```

←Tout ce code est mort et éliminé. Une fonction si courte sera surement « inliné ».

Les Vulnérabilités et la Compilation

Contremesures

- **La meilleure contremesure: La vigilance du programmeur.**
- Le programmeur doit être conscient de l'existence des comportements non-définis dans le langage, et de leur interaction possible avec les optimisations du compilateur.
- Il doit s'assurer que les comportements non-définis sont impossibles dans son code, au moyen de tests et\ou d'assertions.

Les Vulnérabilités et la Compilation

Contremesures: Le compilateur gcc

- L'option **–fwrapv** suppose que l'overflow des integers est possible. Ainsi, les optimisations mentionnées ici n'auront pas lieu. Par défaut en Java.
- **-fno-delete-null-pointer-checks** empêche l'élimination des null checks.
- Le flag **-Wstrict-overflow=n** lance un warning si une optimisation a lieu dans une situation où un overflow est possible.

Les Vulnérabilités et la Compilation

Impact des optimisations sur l'efficacité du programme

- Une étude de Wang et al. compare le temps d'exécution de 12 programmes avec et sans les optimisations de compilation.
- Un ralentissement (≈ 10%) est observable pour 2 des 12 programmes.
- Dans les deux cas, le ralentissement est localisé dans une seule boucle, et une attention particulière à celle-ci permet de l'optimiser manuellement.

Les Vulnérabilités et la Compilation

Contremesures: les outils

- FramaC est un outil gratuit qui examine le code source, et détecte le genre d'erreurs décrites dans cette présentation.
- Les lignes du code source qui seront supprimées par le compilateur sont surlignées selon un code de couleur.

- La librairie **SafeInt** de David Leblanc offre des types pour lesquels l'overflow lance systématiquement une exception. Dietz et al. ont identifié 43 overflow possibles.

Les Vulnérabilités et la Compilation

Conclusion

- Les comportements non-définis sont une réalité des langages C\C++, et de tous autres langages. Le programmeur doit en être conscient pour produire du code correct et sécuritaire.

Références


RAPHAËL KHOURY
LA SÉCURITÉ LOGICIELLE
une approche défensive

Les Vulnérabilités et la Compilation

Merci