LES OPTIONS POUR FAIRE DU COMPILATEUR C UN AMI QUI VOUS VEUT DU BIEN

Olivier LEVILLAIN

Développer du code de manière sécurisée est une tâche complexe qui demande une attention de chaque instant. C'est d'autant plus vrai dans des langages relativement bas niveau comme le C, où les erreurs peuvent mener à des corruptions de la mémoire avec de graves conséquences sur la sécurité. Cependant, les compilateurs modernes offrent de nombreuses options pour détecter des comportements erronés et vous alerter de l'existence possible de problèmes dans votre code. Cet article présente certaines de ces options, qui vous aideront à produire du code de meilleure qualité et plus sécurisé, pour un effort modéré.

Dans le monde logiciel, un adage courant dit que plus un *bug* ou une faille de sécurité est repéré tard dans le cycle de vie d'un programme, plus sa correction coûtera cher. Pour ne citer qu'un exemple, détecter correctement du code mort et corriger le code en conséquence coûte sans doute moins cher qu'une vulnérabilité telle que `goto fail` dans une implémentation de TLS **[1]** (voir la section « Retour sur goto fail » en p. 104). Cependant, il n'est pas toujours facile de trouver le temps et les moyens de chercher et de corriger les erreurs que l'on retrouve immanquablement dans les logiciels.

Nous nous intéressons dans cet article au développement logiciel en langage **C**, un langage relativement bas niveau, dans lequel les manipulations mémoire sont à la charge du développeur, et où les structures de données offertes par le langage sont rudimentaires. Dans le contexte du C, il existe un outil qui interprète la totalité de votre code pour produire l'exécutable final : le compilateur. Il s'agit donc de l'endroit idéal pour intégrer des vérifications syntaxiques et sémantiques, de manière intégrée au processus de développement.

L'objet de cet article est de montrer que le compilateur C peut être un allié pertinent pour améliorer la qualité du code, réduire les surprises à l'exécution, et donc participer à une meilleure sécurité des programmes produits. Voyons donc comment il est possible de demander au compilateur de vous réprimander pour vous garder dans le droit chemin. Pour cela, l'article présente des exemples de code incorrects, pouvant mener à des failles de sécurité, ainsi que des options pertinentes du compilateur permettant de détecter les problèmes sous-jacents. Idéalement, la prise en compte des avertissements du compilateur doit devenir un réflexe chez le développeur, et c'est pourquoi il est essentiel d'enseigner ces bonnes pratiques au plus tôt. Cette démarche devrait également aussi concerner les cursus ne visant pas à former des spécialistes en sécurité du numérique. En effet, un spécialiste en sécurité du numérique, même excellent, sera démuni face à une horde de développeurs sans le bagage minimal en sécurité (c'est d'ailleurs l'objet du projet **[2]** d'initier tous les acteurs du système d'information à la sécurité).

Il ne faut néanmoins pas être naïfs : une démarche consistant à forcer la prise en compte des avertissements du compilateur C est plus facile à faire pour un nouveau projet que pour un projet avec un historique se comptant en décennies. Il faudra dans ce dernier cas faire preuve de patience pour lutter contre ce qu'il est commun d'appeler la dette technique. Il faudra alors adopter progressivement les bons usages, d'une part pour corriger l'existant et d'autre part pour veiller à ne pas accroître la dette technique.

Les exemples de cet article ont été vérifiés avec **gcc** 6.3 et **clang** 3.8.1. De nombreuses bonnes pratiques présentées ici sont bien sûr adaptables à d'autres compilateurs. De même, certains principes exposés peuvent être transposés à d'autres langages.

MISE EN JAMBES

Considérons une fonction vérifiant qu'un accès à une ressource est autorisé en vérifiant l'identifiant d'un utilisateur (`uid`) et l'identifiant du groupe auquel cet utilisateur appartient (`gid`). Pour cela, le développeur a défini deux constantes (`expected_uid` et `expected_gid`) et souhaite autoriser l'accès si l'utilisateur, ou son groupe correspond à celui qui est attendu. Cependant, dans l'extrait suivant, une typo s'est glissée :

```
static const int expected_uid = 1000;
static const int expected_gid = 2000;

int check_access (const int uid, const int gid) {
  if (uid == expected_uid)
    return 1;
  if (gid == expected_uid)
    return 1;
  return 0;
}
```

Contrairement à l'intention derrière ce code, le `gid` à tester est comparé à la valeur `expected_uid` et non à `expected_gid`.

Une manière de détecter l'erreur est de constater que la constante `expected_gid` n'est jamais utilisée. Le compilateur peut aisément vous avertir de cet écart :

```
$ gcc -Wall unused-global.c
unused-global.c:2:18: warning: 'expected_gid' defined but not
used [-Wunused-const-variable=]
 static const int expected_gid = 2000;
                  ^~~~~~~~~~~~
```

En ajoutant `-Wall` aux options de compilation, `gcc` vous indique que la constante `expected_gid` n'est pas utilisée. Il vous indique de plus le nom précis de l'avertissement qui a déclenché le message d'erreur (ici, `-Wunused-const-variable`, inclus dans le groupe d'avertissements activés par `-Wall`).

Le problème, c'est qu'il est encore trop facile d'ignorer cet avertissement si on ne fait pas attention. Il est donc important, voire essentiel, d'ajouter l'option `-Werror` qui indique au compilateur que les avertissements doivent être traités comme des erreurs. Cette option impose ainsi au développeur de corriger les problèmes détectés s'il veut que la compilation aboutisse.

Une variante du problème ci-dessus consiste à utiliser deux fois la variable **uid** dans les comparaisons. Le code correspondant est donné dans l'extrait suivant :

```
static const int expected_uid = 1000;
static const int expected_gid = 2000;

int check_access (const int uid,
const int gid) {
  if (uid == expected_uid)
    return 1;
  if (uid == expected_gid)
    return 1;
  return 0;
}
```

Ici, le problème est qu'**uid** (et non **gid**) est comparé à **expected_gid**.

Cependant, **gcc -Wall** sur ce code ne lève aucune erreur. Il faut ajouter une autre option (**-Wextra**) pour déclencher un avertissement (**-Wunused-parameter**).

Les esprits taquins pourraient remarquer à ce stade que les développeurs de **gcc** ont une vision intéressante de la théorie ensembliste, **-Wall** ne contenant à l'évidence pas vraiment *tous* les avertissements, mais il est amusant de constater en plus que **gcc** ne lève la dernière alerte présentée que si **-Wall** et **-Wextra** sont activés simultanément...

NOTE

Dans un langage de plus haut niveau, ce type de méprise peut être réglé en forçant les uid et les gid à vivre dans des types différents, mais il est difficile de faire cela de manière simple et succincte en langage C.

Étudions un dernier exemple lié aux options standard des avertissements. L'extrait suivant présente un programme simple comparant deux entiers et affichant le résultat (étonnant) de cette comparaison :

```
#include <stdio.h>

int main () {
  unsigned int a = 1;
  signed int b = -1;

  if (a<b)
    printf("%d<%d\n",a,b);
  else
    printf("%d>=%d\n",a,b);

  return 0;
}
```

Compiler et exécuter ce code rendra le résultat suivant : **1<-1**, ce qui peut remettre en cause certaines idées préconçues en maths... Ce comportement, bien que contre-intuitif, est pourtant prédictible, puisque le code proposé compare un entier signé et un entier non signé. Dans le cas présent, le standard C indique en effet que les deux opérandes doivent être converties vers le type **unsigned int**, ce qui ne se fait pas sans perte d'information, puisque **-1** n'est pas représentable, que cette valeur est dès lors interprétée comme un entier très grand.

S'il est utile de connaître ces règles et de manière plus générale les concepts de promotions et de conversions implicites du langage C, il semble également pertinent que le compilateur vous rappelle à l'ordre régulièrement sur le sujet, pour éviter des surprises désagréables en cas d'inattention ou de méconnaissance. C'est l'objet de l'avertissement **-Wsign-compare**, inclus dans **-Wall -Wextra** :

```
$ gcc -Wall -Werror -Wextra
conversion.c
conversion.c: In function 'main':
conversion.c:7:8: error: comparison
between signed and unsigned integer
expressions [-Werror=sign-compare]
   if (a<b)
        ^
```

Bien que les exemples présentés dans cette première section soient très simples et un brin artificiels, les erreurs présentées sont vraisemblables et peuvent mener à des failles de sécurité (par exemple, un contournement de la politique d'accès prévue pour les premiers exemples). Le premier conseil à retenir de cet article est donc de toujours utiliser **-Wall -Wextra -Werror** dans les projets développés en C. Dans le cadre d'un cours de programmation, il serait sans doute pertinent de retirer des points à un élève dont le code ne compile pas avec ces options !

UNE QUESTION DE FORMAT

Lorsqu'on s'intéresse à la sécurité du logiciel, on rencontre fréquemment le concept d'injection, qu'il s'agisse d'injection **SQL**, d'injection **shell** ou encore de XSS (*Cross-Site Scripting*). L'idée est qu'une chaîne de caractères, partiellement contrôlée par l'attaquant, est interprétée par *quelque chose* (un SGBD, le shell, le navigateur) pour reconstruire une structure mise en évidence par des caractères spéciaux. Dans certains cas, il devient possible, en introduisant ces caractères, de modifier la structure attendue par le développeur.

En C, on rencontre ce cas avec **printf** (et toutes ses fonctions dérivées) : son premier argument, la chaîne de format, permet de construire un texte à trous (représentés par des éléments signalés par %). Que se passe-t-il donc si un attaquant peut injecter des caractères % dans cette chaîne ? Cela ouvre la voie à une *format string attack*. Considérons le morceau de code suivant, qui décrit une vulnérabilité de type *format string attack*, dans son plus simple appareil :

```
#include <stdio.h>

int main (int argc, char* argv[]) {
 if (argc == 2)
        printf (argv[1]);
 return 0;
}
```

En appelant le programme compilé avec des arguments bien choisis, on peut révéler le contenu de la pile !

```
$ ./fmtstr '%p %p %p %p '
0x7fff3549b1d8 0x7fff3549b1f0 (nil) 0x558d3b465750
```

Mais au-delà de cette divulgation du contenu de la mémoire, un attaquant peut aussi modifier le contenu de la mémoire... en utilisant l'option **%n**, qui va consommer la valeur suivante dans la liste d'arguments, la considérer comme un pointeur vers un entier, et y inscrire le nombre d'octets déjà produits par **printf**.

Il est donc dangereux d'intégrer des entrées utilisateur dans une chaîne de format. Une règle simple est d'imposer que la chaîne de format soit une chaîne de caractères constante. Là encore, le compilateur C vous offre des options pour générer des avertissements, la famille **-Wformat***. De plus, si la chaîne de format est constante, cela permet en plus au compilateur de vérifier statiquement que votre liste d'arguments est cohérente avec la chaîne de format. La documentation du compilateur **[3]** donne d'amples détails, mais l'option **-Wformat=2** est la plus complète. Appliquons-la (avec **-Werror** bien sûr) à notre programme :

```
$ gcc -Wformat=2 -Werror fmtstr.c
fmtstr.c: In function 'main':
fmtstr.c:5:9: error: format not a string literal
and no format arguments [-Werror=format-security]
         printf (argv[1]);
         ^~~~~~
```

NOTE

De manière générale, lorsque le compilateur génère un avertissement, il faut accepter la triste vérité : c'est lui qui a raison. Attention donc à ne pas accepter les conseils donnés à la va-vite sur certains sites [4]. Ainsi, lorsque vous utilisez `printf` avec des arguments ne correspondant pas au format attendu (par exemple, un entier pour une chaîne de caractères), vous êtes prié de ne pas débrayer la sécurité pour éviter de corriger votre code ! Sauf cas vraiment exceptionnel, votre code devrait toujours compiler avec les options présentées dans cet article.

-WWRITE-STRINGS

Considérons à présent un autre programme, extrêmement simple, qui manipule et affiche des chaînes de caractères :

```
#include <stdio.h>

int main () {
  char* s = "Ho";
  printf ("%s\n", s);
  s[1] = 'a';
  printf ("%s\n", s);
  return 0;
}
```

Puis lançons-le :

```
$ gcc -o string-test -Wall -Wextra
-Werror string-test.c
$ ./string-test
Ho
Segmentation fault
```

À notre plus grande surprise, le programme, qui a été compilé sans problème malgré les options activées, plante à l'exécution. Étudions ce cas de plus près : l'erreur semble survenir lors de la modification de la chaîne de caractères `s`. En effet, en marche normale, sur des architectures modernes, les chaînes de caractères constantes (comme `"Ho"`) vivent dans une section intitulée `rodata` qui, comme son nom l'indique, est en lecture seule (`ro` signifie ici *read only*). L'affectation `s[1] = 'a'` va donc échouer...

Mais ici, la véritable source de l'erreur est ailleurs, car le compilateur a en fait déjà accepté qu'une variable de type `char*` (donc désignant des données modifiables) pointe vers une zone en lecture seule. Il est possible d'exiger du compilateur de vous en avertir avec l'option `-Wwrite-strings` :

```
$ gcc -Wwrite-strings -Werror string-test.c
string-test.c: In function 'main':
string-test.c:4:13: error: initialization
discards 'const' qualifier from pointer
target type [-Werror=discarded-qualifiers]
   char* s = "Ho";
             ^~~~
```

On obtient alors l'erreur au bon endroit, puisque `s` devrait être déclarée comme `const char*`.

Il est cependant surprenant que ce problème, trivialement détectable par le compilateur, ne génère pas systématiquement d'avertissement... surtout que compiler avec `g++` active par défaut cet avertissement ! L'explication de ce comportement est un peu triste, et se trouve dans la page de manuel de `gcc` : « *These warnings help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using «const» in declarations and prototypes. Otherwise, it is just a nuisance. This is why we did not make -Wall request these warnings.* ». Le lecteur avisé aura sans doute un autre avis, et considèrera peut-être que la *nuisance* est plutôt d'avoir une erreur à l'exécution statiquement prédictible, mais non prise en compte par le compilateur... Comme pour d'autres exemples, il suffit d'imaginer cette erreur dans un code d'une taille plus importante. Sauriez-vous le trouver facilement ?

NOTE

Cet exemple a été soumis à la sagacité d'élèves lors d'un TP. Une des propositions pour résoudre le problème était de remplacer la ligne `char* s = "Ho"` par `char s[] = "Ho"`. Le code ainsi modifié compile et ne déclenche plus d'erreur à l'exécution. C'est un bel exemple montrant une différence subtile entre tableaux et pointeurs en C. Pour comprendre la différence, il suffit de réfléchir à l'endroit où « vit » la chaîne de caractères dans les deux cas (la section `rodata` dans le premier cas, la pile dans le second cas).

RETOUR SUR GOTO FAIL

En 2014, une vulnérabilité a été publiée par **Apple** sur son implémentation TLS **[1]**. La faille concerne la vérification de la signature du message `ServerKeyExchange` par le serveur : dans le code ci-dessous, extrait de la fonction vulnérable, plusieurs tests sont réalisés.

```
    ...
    hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
    hashOut.length = SSL_SHA1_DIGEST_LEN;
    if ((err = SSLFreeBuffer(&hashCtx)) != 0)
        goto fail;
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
         goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    err = sslRawVerify(...);
    ...
```

Si un de ces tests échoue (en renvoyant une valeur non nulle dans `err`), la fonction sort en renvoyant une valeur non nulle. En revanche, à cause d'une ligne `goto fail` dupliquée, la seconde ligne correspond à un saut inconditionnel, dans un état où `err` vaut `0`, ce qui mène à un retour de la fonction nulle, qui signifie que tout s'est bien passé... À cause de ce bégaiement dans le code, la fin des tests n'est jamais exécutée, ce qui mène en pratique à ne jamais vérifier la signature !

Une manière d'interpréter ce problème est que le compilateur a accepté de traiter ce morceau de code sans signaler que de nombreuses lignes sont inaccessibles. Or, il semble trivial pour un compilateur de détecter statiquement ce type de code mort. Comme pour les exemples précédents, il existe une option standard pour générer un avertissement dans de tels cas : `-Wunreachable-code`.

Cependant, cette option ne fonctionne pas avec les versions `gcc` postérieures à 4.4. En effet, l'option n'était plus fiable, car la détection de code mort se faisait après plusieurs passes de transformation de code, ce qui pouvait mener à des fausses alertes. Les développeurs de `gcc` ont fait le choix, discutable, de conserver l'option, mais sans activer de tests.

Au lieu de râler contre les développeurs de `gcc`, pourquoi ne pas tenter à ce stade un autre compilateur ? `clang` fait alors très bien l'affaire ; l'architecture interne de ce compilateur rend en effet possible la détection de code mort :

```
$ clang -Wunreachable-code -Werror heartbleed.c
heartbleed.c:27:16: error: code will never be executed [-Werror,-
Wunreachable-code]
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
               ^~~~~~~~~~~~~~~~
1 error generated.
```

En pratique, il peut être utile de tester son code avec plusieurs chaînes de compilation différentes. Cette considération rejoint le problème plus général de la portabilité du code. Dans une certaine mesure, rechercher la portabilité du code peut améliorer la qualité du code. En compilant et en testant un programme dans différents contextes, on peut par exemple mettre en évidence des hypothèses non explicitées (un cas classique est la représentation des entiers en 32 et 64 bits).

NOTE

Dans le cas particulier de `goto fail`, les versions récentes de `gcc` peuvent néanmoins vous avertir qu'il y a un problème. En effet, `-Wall` active l'avertissement `-Wmisleading-indentation` (détection des indentations suspectes dans les blocs du langage), qui aurait été déclenché en l'occurrence.

UN MOT SUR LES CONVENTIONS D'ÉCRITURES

Dans de nombreux projets, il existe des règles de codage qui permettent d'appliquer de bonnes pratiques et d'homogénéiser les conventions sur l'ensemble du code. On peut citer par exemple les règles mises en œuvre dans le noyau Linux **[5]**. De telles règles peuvent améliorer la qualité et la lisibilité du code (et donc dans une certaine mesure, améliorer la sécurité du code). Cependant, ce n'est pas automatique.

Cette question débordant légèrement de l'objet de cet article, nous nous concentrerons sur deux points : le nommage des variables et les commentaires.

What's in a name?

Si certains choix relèvent de la question des goûts et des couleurs (variables en `camlCase` ou `avec_des_soulignés`), il est courant d'imposer que le nom des variables soit explicite. Attention cependant à ne pas imposer l'utilisation de nom explicite de manière trop générale. Il est inutile, voire contre-productif d'appeler `argument_counter_i` (le suffixe `_i` étant parfois utilisé pour rappeler le type de la variable dans son nom...) le nom de la variable parcourant le tableau des arguments reçu sur la ligne de commandes dans l'extrait suivant :

```
int main(int argc, char* argv) {
    for (int argument_counter_i = 1; argument_counter_
i<argc; argument_counter_i++)
        handle_argument(argv[argument_counter_i]);
    ...
}
```

Une règle classique de bon sens est que la longueur et la précision dans le nom d'une variable doivent être inversement proportionnelles à la portée de celle-ci. Une variable globale doit être explicite et sans ambiguïté, alors qu'une variable locale peut être nommée de manière plus légère. L'exemple précédent est le cas extrême d'une variable utilisée uniquement sur deux lignes de code, où utiliser un nom de variable comme `i` aurait suffi. Cette idée est décrite dans les règles de codage du noyau Linux (section « Retour sur goto fail » en p. 104).

Une manière de se convaincre qu'on a bien choisi ses noms de variables est de faire relire son code. Intéressons-nous par exemple à un morceau de code qui a été beaucoup regardé en avril 2014, la fonction prenant en charge le paquet **Heartbeat** dans **OpenSSL**, dont un extrait est présenté dans l'extrait suivant :

```
    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;
```

Au-delà du dépassement de tampon en lecture qui a été largement décrit dans la presse (la faille Heartbleed, CVE-2014-0160, publiée en avril 2014), il s'agit d'un contre-exemple au sujet discuté dans la présente section. En regardant le code, pouvez-vous facilement identifier ce que sont `p`, `pl` et `payload` ? Si le `l`

de `pl` suggère une longueur, en réalité, `p` et `pl` sont des pointeurs vers le contenu du paquet en cours de lecture, et `payload` correspond à la longueur de la charge utile du paquet Heartbeat. Même s'il s'agit de variables locales, il aurait été pertinent d'utiliser des noms plus évocateurs.

Le problème des conventions de nommage, c'est qu'elles sont difficiles à appliquer automatiquement. Il existe cependant un cas où le compilateur peut vous aider : la réutilisation d'un nom de variable de manière maladroite qui crée une confusion entre les deux variables. Dans de nombreux langages, le compilateur peut vous avertir lorsqu'une variable est masquée par une autre. Un exemple classique consiste à donner à une variable locale le même nom qu'une variable globale. En C, vous pouvez détecter ce type de collision avec l'option `-Wshadow`.

Des commentaires ?

L'utilisation des commentaires est souvent l'objet de débats passionnés, et l'ajout de règles de codage à leur sujet peut être extrêmement contre-productif. Par exemple, imposer que 30% du code soit des commentaires est stupide. Cela mène à des lignes comme `i++ // incrémentation de i`. Convenons que le commentaire n'apporte rien et dégrade même la lisibilité.

Sur-commenter son code a un impact négatif sur sa lisibilité. Dans la section 8 des règles de codage du noyau Linux, un conseil est donné : mieux vaut écrire du code dont le fonctionnement est évident à la lecture que produire du code incompréhensible et le commenter...

Il existe néanmoins deux situations (essentiellement les deux seules selon l'auteur de cet article) où les commentaires sont utiles et nécessaires : pour définir les interfaces (d'un module, d'une fonction), et pour mettre en évidence les points durs (et uniquement ceux-là) lors d'un algorithme complexe.

Il y a un autre aspect de la documentation qu'il ne faut pas négliger, qui ne relève pas des commentaires, et qui peut être vérifié automatiquement par le compilateur. Il s'agit de l'ensemble des indications que vous pouvez donner au compilateur sur le typage des fonctions et des variables. On peut citer par exemple le qualificatif `const` sur les arguments de fonction ou sur les variables, qui est toujours vérifié par le compilateur.

On peut également forcer le développeur à systématiquement annoncer le prototype des fonctions qu'il exporte, ce qui permet de les regrouper et de les documenter à un endroit du code. Toute modification de la signature de la fonction pourra ainsi être détectée par le compilateur, et il sera plus naturel d'assurer la cohérence de la documentation de l'interface, située près des prototypes, avec son implémentation, disséminée dans le code. L'option de `gcc` est `-Wmissing-prototypes`.

Enfin, pour donner un dernier exemple de bonne pratique liée à la documentation et au typage, il est souvent préférable d'éviter de définir des courtes fonctions sous la forme de macros, mais de leur préférer les déclarations de fonctions `static inline`, qui définissent au sein du langage des fonctions réelles, pour lesquelles le typage est vérifié, et qui seront dépliées (*inlinées*) si le compilateur le juge pertinent (et il aura tendance à avoir raison).

Ainsi, bien que son usage soit limité en C, le typage est un outil formidable en programmation puisqu'il représente une forme de documentation toujours à jour (du moins tant que votre programme compile). Il faut le considérer comme un atout et non comme une contrainte !

POUR ALLER PLUS LOIN

Le présent article propose des premières pistes pour utiliser le compilateur de manière à vous montrer le droit chemin. Il existe bien entendu de nombreuses autres options liées à l'émission d'avertissement. On peut citer par exemple la documentation de `gcc` **[3]** ou un billet de blog récent sur le sujet **[6]**. Si vous souhaitez juste la liste des options, vous pouvez aussi appeler `gcc -Q --help=warning`.

Au-delà de cette aide à la compilation, le compilateur est également un outil puissant pour instrumenter le code exécutable ou donner des directives pour améliorer la sécurité à l'exécution. On pensera naturellement à la série des options `-fstack-protector`, qui incluent des canaris dans la pile pour détecter certains débordements de tampon dans la pile. Citons également `-fPIC` et `-fPIE`, qui rendent le code produit compatible avec l'ASLR (*Address Space Layer Randomization*, randomisation de l'espace mémoire du processus). Enfin, il peut être utile de regarder les options de relocalisation de bibliothèques (options `relro` de l'éditeur de liens dynamique `ld`) ou l'option `D_FORTIFY_SOURCE=2`.

Pour aller encore plus loin, il existe des outils d'analyse statique, et des outils permettant d'étudier le comportement du code produit grâce à de l'instrumentation (par exemple `asan` et `ubsan`, inclus dans les versions récentes de `gcc` et `clang`).

Enfin, une ressource intéressante pour un développeur avisé est l'étude *Mind your langages* **[7]**, co-écrite par l'auteur du présent article, dans le cadre de travaux réalisés à l'ANSSI sur les langages. Cette étude présente un certain nombre de pièges des langages de programmation, qui peuvent mener à des failles de sécurité. Il est donc utile, voire nécessaire, de bien connaître les langages que l'on utilise.

CONCLUSION

Développer du code de manière sécurisée est une tâche difficile. Dans cet article, quelques options des compilateurs C ont été présentées pour générer des alertes en cas de problèmes potentiels. Prendre en compte ces avertissements de manière systématique permet d'améliorer la qualité du code et d'éviter certaines classes d'attaques de manière simple et efficace. Rappelez-vous que c'est le compilateur qui travaille pour trouver tous ces *bugs* potentiels !

Pensez donc à ajouter `-Wall -Wextra -WXXX -Werror` dans tout nouveau projet développé en C. Et pour les projets plus anciens, lorsqu'activer l'option `-Werror` n'est pas réaliste, ne baissez pas les bras, et essayez progressivement de réduire la dette technique en partant à la chasse aux avertissements levés par les autres options.

Il faut néanmoins rester humble et se rappeler que ces préconisations ne sont que la première ligne de défense. Il faut ensuite faire relire son code, tester, voire prouver certaines propriétés. ▌

REMERCIEMENTS

Je profite de ces lignes pour remercier Éric, Arnaud et Pascal pour les nombreuses discussions autour des langages (le C en particulier) et pour leur relecture attentive.

RÉFÉRENCES

[1] Vulnérabilité goto fail dans la pile TLS d'Apple, CVE-2014-1266 : **https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266**

[2] Site de l'association CyberEdu : **https://www.cyberedu.fr**

[3] *« Options to Request or Suppress Warnings »* : **https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html**

[4] Stack Overflow, *« Selectively remove warning message GCC »* (attention, il s'agit d'un contre-exemple au propos de cet article, et non d'un exemple de bonne pratique) : **https://stackoverflow.com/questions/925179/selectively-remove-warning-message-gcc**

[5] Équipe de développement du noyau Linux, *« Linux kernel coding style »* : **https://www.kernel.org/doc/html/latest/process/coding-style.html**

[6] WALFRIDSSON K., *« Useful GCC warning options not enabled by -Wall -Wextra »*, publié le 16/09/2017 : **https://kristerw.blogspot.fr/2017/09/useful-gcc-warning-options-not-enabled.html**

[7] JAEGER É. et LEVILLAIN O., *« Mind your Language(s): A Discussion about Languages and Security »*, publié en mai 2014 au workshop LangSec@IEEE SSP : **https://www.ssi.gouv.fr/agence/publication/mind-your-languages-nouvel-article-sur-limportance-des-langages-pour-la-securite/**