Confusion de type en C++ : la performance au détriment de la « type safety »

Florent Saudel
florent.saudel@amossys.fr

AMOSSYS

Résumé. Depuis le début du premier semestre 2016, six CVE permettant l'exécution de code arbitraire ont été déposées. Ces CVE ont toutes en commun l'exploitation d'une vulnérabilité encore peu considérée « la confusion de type ». Elle peut, pourtant, à elle seule outrepasser les sécurités actuelles et mener à l'exploitation. Cette vulnérabilité correspond à la manipulation d'objets C++ d'une façon incompatible avec leur type réel. Son origine réside dans la mauvaise utilisation des opérateurs de conversions. Or, ces opérations sont courantes dans les logiciels où la programmation orientée objet et le polymorphisme ont une place importante. Cet article présente plus en détail l'origine de cette vulnérabilité et les mécanismes bas niveaux à l'origine de son exploitabilité.

1 Introduction

La première partie de cet article présente l'intérêt de cette vulnérabilité en prenant en compte les sécurités et autres contre-mesures actuelles. La suite s'attarde sur l'origine de la vulnérabilité, que nous nommons « mauvaise conversion » ou « *bad cast* », ainsi que de son influence sur le polymorphisme implémenté par les compilateurs. Puis, nous illustrons comment exploiter cette faille au travers d'un exemple. Enfin, la conclusion met en évidence l'existence d'une catégorie de programmes plus propice au confusion de type et présente des arguments pour expliquer ce phénomène. En ouverture, nous posons le problème de la détection des « bad cast » en présentant les travaux existants.

En étudiant les différentes base de données de vulnérabilités (Mitre[1], NVD[2] et Zerodium[3]), il ressort que la confusion de type n'est pas la catégorie de vulnérabilités la plus répandue. Aujourd'hui, les « use-after-free » et les « buffer overflow » sont plus communs. La première remarque notable est que la majorité des CVE[4] qui en résultent, mènent à une exécution

[1] https://cve.mitre.org/
[2] https://nvd.nist.gov/
[3] http://www.zerodayinitiative.com/advisories/
[4] Common Vulnerabilities and Exposures

arbitraire. Or, ces mêmes produits profitent de toutes les sécurités actuelles (DEP[5], ASLR[6], MemGC [1], *etc.*) car les navigateurs sont une cible privilégiée. Néanmoins, la confusion de type est suffisamment versatile pour passer au travers des sécurités et mener à bien l'exploitation. Le prochain paragraphe traite des particularités techniques expliquant les capacités de cette vulnérabilité.

Les contre-mesures actuelles cherchent à contrecarrer les possibles corruptions mémoires introduites par des bugs liés à la « memory safety », c'est-à-dire la mauvaise gestion des accès à la mémoire. Cependant, la vulnérabilité qui nous intéresse ici, réside dans le manque de « type safety » au sein des programmes C++. Une fois un programme compilé, les données manipulées ne sont plus typées. Il peut donc y avoir une différence entre le type « attendu statiquement » par le code source et le type « réel dynamique » manipulé par le binaire. Cette divergence d'interprétations des données, lorsqu'elle est contrôlable depuis l'extérieur du programme, peut alors être exploitée. Selon la situation, une confusion de type donne souvent lieu au trois primitives nécessaires à l'exploitation que sont la corruption mémoire, la fuite d'information et le détournement du flot de contrôle.

Un exemple flagrant est la CVE-2013-0912 qui affectait Google Chrome. Lors de l'édition de la compétition Pwn2Own 2013, le MWR Labs a produit une preuve de concept[7] reposant sur une **unique** confusion de type. Ce même bug a été suffisant pour contourner l'ASLR, en retrouvant l'adresse en mémoire d'une bibliothèque partagée, scanner la mémoire afin de trouver des gadgets pour dans le but de détourner le flot de contrôle au profit du shellcode généré dynamiquement. À l'origine de cet exploit, une « simple » erreur de conversion vers un objet censé représenté uniquement un tag SVG. La suite de l'article se penche sur ces erreurs de typage et leurs répercussions au niveau du binaire entrainant cette vulnérabilité.

2 Description et fonctionnement

Le langage C++ est typé statiquement et par conséquent c'est le compilateur qui est chargé de s'assurer de la cohérence des types des variables au sein du programme. Cependant, il existe quatre opérations de conversion : `static_cast`, `reinterpret_cast`, `dynamic_cast` et `const_cast`

[5] Data Execution Prevention
[6] Address Space Layout Randomization
[7] `https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/`, `https://bugs.chromium.org/p/chromium/issues/detail?id=180763`

laissant la possibilité au développeur de modifier, de façon potentiellement incorrecte, le type de ces objets. Seules les trois premières expliquent l'existence des confusions de types. Nous nous limitons donc à celles-ci :

- `static_cast` : elle vérifie statiquement qu'il existe une relation entre le type de départ et d'arrivée avant la conversion. Il est donc possible de convertir d'une classe parente vers une classe fille (« *downcast* ») mais pas entre deux classes sans aucune relation de parenté.
- `reinterpret_cast` : le compilateur réinterprète la donnée (ses bits) selon le nouveau type. C'est la conversion la plus arbitraire possible, le compilateur n'effectue aucune vérification sur la compatibilité des types.
- `dynamic_cast` : cette conversion rajoute une vérification sur le type concret de l'objet converti à l'exécution du programme. Une exception est levée si le type de départ est incompatible avec celui d'arrivée. Cette conversion détecte donc les confusions de types. Malheureusement, cette vérification dynamique impacte les performances. De ce fait, des projets comme Chrome ou Firefox bannissent son utilisation.

La source du bug vient donc de l'utilisation de conversion non sûre. En particulier, l'utilisation du `static_cast` pour réaliser un « *downcast* » incorrect semble être le bug le plus répandu [2].

Nous venons de voir les erreurs au sein du code source menant à une confusion de type. Le prochain point aborde les mécanismes bas niveaux afin de comprendre comment ils peuvent être détournés par un attaquant.

2.1 Exploiter l'implémentation du polymorphisme

Au niveau du binaire, la notion de type est inexistante. Un objet n'est représenté en mémoire que par une succession d'octets formant des champs (c'est une structure dans le sens du langage C). Par conséquent, l'accès à un champ ou une variable membre de l'objet ne correspond qu'à la lecture ou l'écriture d'une sous-partie de cette structure. À chaque champ est associé un décalage (*offset*) représentant le début du champ en mémoire par rapport à l'adresse où est stockée l'objet. Ces décalages sont calculés au moment de la compilation et ils dépendent du type de l'objet. Si lors de son exécution, le code généré traite un objet à la structure interne différente de celle attendue, des comportements indéfinis par le standard apparaissent.

Une confusion de type revient donc à appliquer un mauvais schéma à une donnée brute. La nature des confusions de type affectant un programme

dépend entièrement de ses classes et ses structures. Cependant, il est possible de citer quelques scénarios classiques qu'un attaquant cherchera à provoquer :

- l'objet concret est plus petit que celui attendu. Lors d'un accès à un des champs, le programme lit ou modifie des zones mémoires situées après l'objet en question.
- le champ de l'objet concret est un pointeur alors que le programme le manipule comme un entier ou un flottant.

L'accès aux champs d'un objet confus peut donc mener à la récupération d'une pointeur en mémoire ou à la corruption d'une donnée importante du programme en cours.

En programmation orientée objet, un objet possède des données et un comportement. Il est également possible de détourner à son avantage le comportement d'un objet confus. L'exploitation repose ici sur l'implémentation du polymorphisme au niveau binaire. Cette implémentation est définie par une spécification. Dans un souci de brièveté, nous ne présentons que celle utilisée par GCC et Clang, la spécification C++ Itanium[8].

Le code source dans le listing 1 présente un cas classique d'utilisation du polymorphisme en C++. Trois classes (D1, D2 et B) sont définies telles que D1 et D2 héritent de l'interface. Les classes D1 et D2 définissent toutes les deux la méthode virtuelle `isNext` déclarée par B. L'héritage et ces définitions permettent alors d'appeler `isNext` sur un objet B (ligne 22) sans se soucier du type concret de l'objet appelé (D1 ou D2).

Pour permettre cela, la spécification définit la manière de structurer les objets utilisant des méthodes virtuelles et la façon d'appeler ces méthodes. Une table de pointeurs de fonction appelée `vtable` ou `vftable` est générée à la compilation pour toutes les classes possédant au moins une méthode virtuelle. De plus, à chaque méthode virtuelle est associée un index dans la `vtable` qui sera utilisé lors de l'appel à la méthode.

Lors de la création d'un objet, un pointeur vers la `vtable` de sa classe est rajouté au début de la zone mémoire où ses champs sont stockés. Ainsi, le pointeur `this` de l'objet C++ pointe en réalité sur sa `vtable`. Pour l'exemple, la figure 1 montre la structure en mémoire d'un objet de type D1 et un de type D2. Les structures sont composées toutes les deux de deux pointeurs : un pointeur sur la `vtable` et leur champ.

Pour réaliser l'appel à une méthode virtuelle, les compilateurs utilisent une séquence particulière d'instructions terminant par un saut indirect. À partir du pointeur `this`, le programme récupère le pointeur vers la

[8] `http://mentorembedded.github.io/cxx-abi/abi.html`

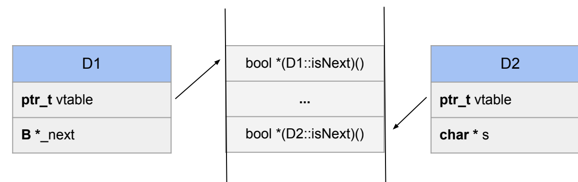

Fig. 1. Représentation en mémoire des objets de type D1 et D2

`vtable` puis l'adresse de la méthode à l'aide de son index. Il s'en suit un appel de fonction classique avec le pointeur `this` en tant que premier argument. Ce fonctionnement est commun à toutes les classes possédant une méthode virtuelle. Pour un attaquant, il « suffit » alors de présenter à cette séquence de code un pointeur vers une fausse `vtable` ou même un autre objet du programme pour lui faire exécuter le code qu'il souhaite.

Afin d'illustrer notre propos, le listing 1 montre comment un attaquant peut exploiter la confusion de type afin d'exécuter un code arbitraire représenté par la variable `shellcode`. Pour la suite des explications, nous supposons que le programme vulnérable a été compilé en 64 bit puis que l'attaquant récupère le pointeur vers l'objet confus `faux_d2` (ligne 33) de type concret D1. Comme les types D1 et D2 partagent un même alignement de leur champs (figure 1), l'accès `faux_d2->s` (ligne 34) revient à lire le pointeur `d2` stocké dans le champ `_next` de la variable `b`. À partir de là, l'attaquant peut remplacer la `vtable` de l'objet `d2` (ligne 35) par celle de la classe D1 afin de provoquer une autre confusion plus tard dans la méthode `traversal`. Ensuite, l'attaquant crée une fausse `vtable` dans le champ *s* de l'objet `d2` dont la première entrée pointe sur son *shellcode* (lignes 37 à 40). Elle sera utilisée plus tard pour lancer le *shellcode*. Le détournement du flot de contrôle s'opère lors de l'appel à la méthode `traversal` (ligne 42). La variable `b`, qui sert à parcourir la liste d'objets de type B, pointe au second tour sur l'objet `d2`. Celui-ci est traité comme un objet de type D1 à cause de l'appel à la méthode `isNext`. En effet, comme l'attaquant a échangé le pointeur de sa `vtable` avec celle de la classe D1, c'est la méthode `D1::isNext` qui est appelée. La variable `b` accède alors au champ `_next` de l'objet `d2`. Le champ `s`, comprenant la fausse `vtable`, est à son tour confondu avec un objet de type B. Le troisième appel à `b->isNext()` (ligne 22) saute à l'adresse du *shellcode*.

Nous souhaitons faire remarquer qu'à aucun moment, une adresse fixe n'a été utilisée et que par conséquent ce programme fonctionne avec l'ASLR activée.

```
const char *shellcode = "AAAAAAAA";

struct B {
    virtual bool isNext() = 0;
};
struct D1 : public B {
    D1(B *child) : _next(child) { }
    virtual bool isNext() { return _next != NULL; }
    void traversal();

    private:
    B *_next;
};
struct D2 : public B {
    virtual bool isNext() { return false; }
    char *s;
};
void D1::traversal() {
    B *b = this;
    // Saut à l'adresse de shellcode au tour 3.
    while(b->isNext()) {
        b = static_cast<D1 *>(b)->_next; // Badcast 2 au tour 2.
    }
    D2 *d2 = static_cast<D2 *>(b);
    std::cout << d2->s << std::endl;
}
B *setup() {
    D2 *d2 = new D2();
    return new D1(d2);
}
int main() {
    B *b = setup();
    D2 *faux_d2 = static_cast<D2 *>(b); // Badcast 1.
    char *p = faux_d2->s; // Fuite du pointeur d2.
    * (long long *) p = *((long long *) faux_d2); // Ecrasement de
        la vtable
    // Creation d'une fausse vftable.
    p += 8;
    * (long long *) p = (long long) p + 8;
    p += 8;
    * (long long *) p = (long long) &shellcode;
    D1 *d1 = static_cast<D1 *>(b); // downcast correct
    d1->traversal();
}
```

Listing 1. Déroulement d'une exploitation d'une confusion de type

3 Conclusion

Une classe de programme plus « vulnérable » à la confusion de type se démarque : les navigateurs et les interpréteurs. La première caractéristique commune à l'ensemble de ces programmes est la taille de leur hiérarchie de classe. En effet, l'ensemble des *parser* pour les formats HTML, SVG, XML,

etc. sont unifiés au sein d'une même interface, le *Document Object Model*. Cela aboutie à une arborescence de classes C++, à la fois profonde et large, cachée sous un même sur-type (le noeud DOM). Cette structure est donc susceptible d'enduire en erreur les développeurs sur le type concret des objets. Cet argument explique la présence de « bad cast » au sein des codes sources. Le prochain point démontre pourquoi ces vulnérabilités sont plus susceptibles de mener à une exploitation complète, lorsqu'elles touchent ce genre de produits.

Les navigateurs actuels possèdent tous un interpréteur Javascript intégré. À travers ce langage, un attaquant est capable de manipuler finement les objets présents en mémoire au cour de l'exécution du programme. Combiner à la multitude de classes existantes, les attaquants n'ont pas simplement une confusion entre deux types distincts mais bien plusieurs dizaines (CVE-2013-0912) à leur disposition. Ainsi, ils peuvent construire les confusions adéquates à l'obtention des trois primitives nécessaires à l'exploitation.

Les développeurs et testeurs ne sont pas totalement démunis face à cette famille de bug. Des outils appelés « *sanitizers* » ont été développés ces dernières années. Ils sont intégrés au sein des compilateurs GCC et Clang. Ils enrichissent le code binaire par des vérifications effectives lors de l'exécution du programme, à la manière des « *stack cookies* » [3]. UBsan[9] et CaVer [2] sont dédiés à la détection des « bad casts ». Leur fonctionnement consiste à remplacer tous les opérations de conversion par leur équivalent dynamique. Le surcoût engendré (mémoire et temps de calcul) par les vérifications est prohibitif pour une utilisation systématique en pratique. L'autre limitation inhérente à ces outils est la nécessité de recompiler le logiciel et donc d'avoir accès au code source. Néanmoins, ces solutions sont activement utilisées dans les phases de tests des produits tel que Google Chrome.

Références

1. MemGC : Use-After-Free Exploit Mitigation in Edge and IE on Windows 10, August 2015.
2. Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type Casting Verification : Stopping an Emerging Attack Vector. pages 81–96, 2015.
3. Perry Wagle, Crispin Cowan, and others. Stackguard : Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255. Citeseer, 2003.

[9] `http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer`