Programmer sécurisé: contre quoi se prémunir ? Comment ? Avec quels outils ?

Marie-Laure Potet

Laurent Mounier

Prenom.nom@univ-grenoble-alpes.fr

CyberSkills@UGA

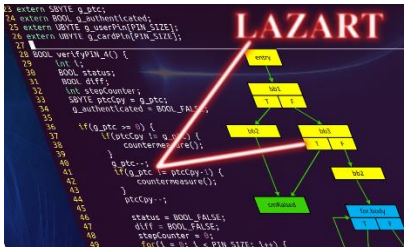
Parcours

- Méthode B : développement prouvé par construction.

- Outils d'analyse de code pour recherche de vulnérabilités

BINSEC

BINARY CODE ANALYSIS FOR SECURITY



- Méthodologie et outils pour les certifications de sécurité : Critères Communs

VISA DE SÉCURITÉ

AGENCE NATIONALE DE LA SÉCURITÉ DES SYSTÈMES D'INFORMATION

ANSSI

Programmer sécurisé

- Les objectifs :
- Vulnérabilités classiques
- Comment se protéger
- En quoi les langages nous aident (ou pas)
- En quoi les outils nous aident (ou pas)
- Et pour l'embarqué ?

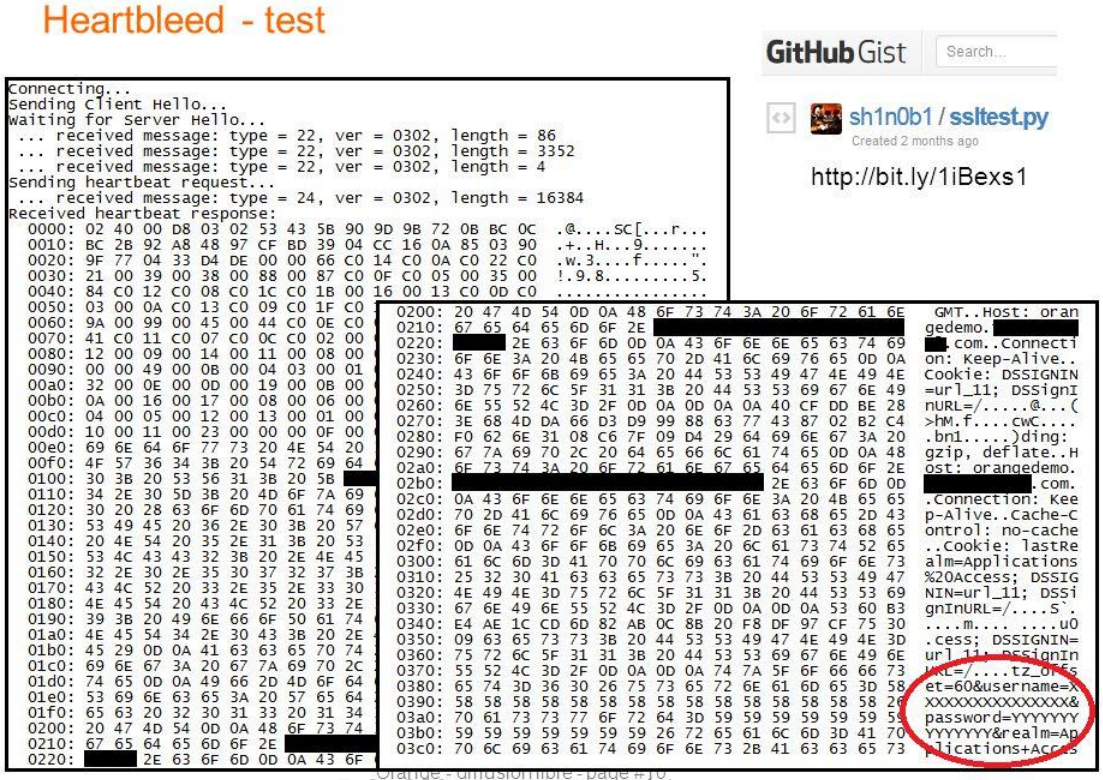
- TPs : attaques/défenses, code C et binaire

Ecrire et vérifier du code sécurisé

les problèmes, les outils, les solutions

heartbleed

A permis de récupérer 64KB de la mémoire distante du processus du serveur. Cette mémoire peut contenir la clé privé du serveur, les clés des sessions SSL/TLS ou encore les requêtes/réponses des requêtes HTTPS avec éventuellement les identifiants de connexion à l'application sous-jacentes ou les cookies de sessions.



Heartbleed (CVE-2014-160) révélée en avril 2014

- Bibliothèque Openssl 1.0.1 (2012)
- Concrètement, un serveur Https sur deux de la planète était concerné avec une compromission possible des clés privées, des mots de passe, de toute autre information présente en mémoire du process. . .
- C'est un simple oubli de vérification de bornes dans le code d'une fonction non critique du protocole SSL/TLS
- Revient régulièrement !

La vulnérabilité:

```
hbtype = *p++;
n2s(p, payload);
pl = p;
```

Le patch :

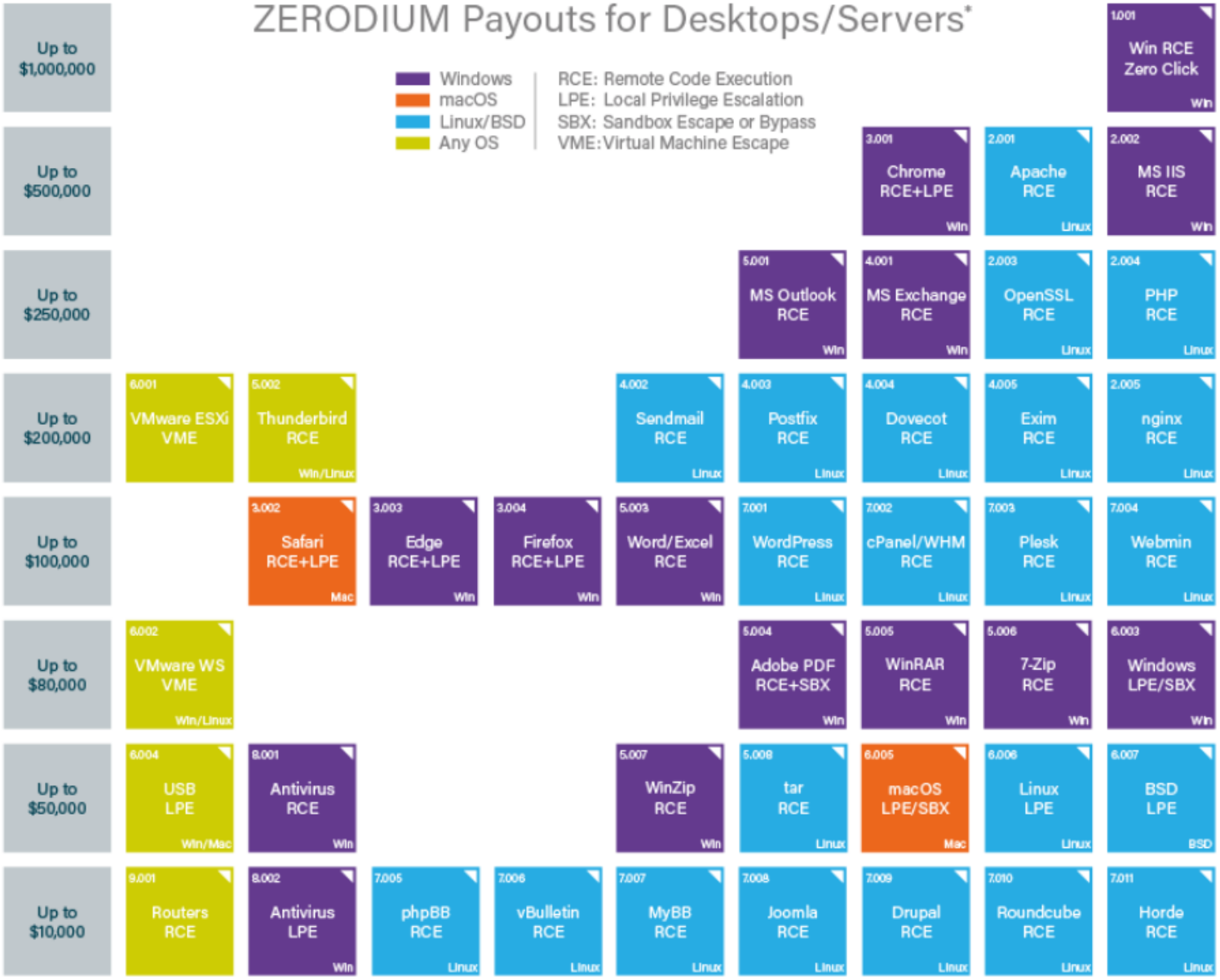
```
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length) return 0
pl = p;
```

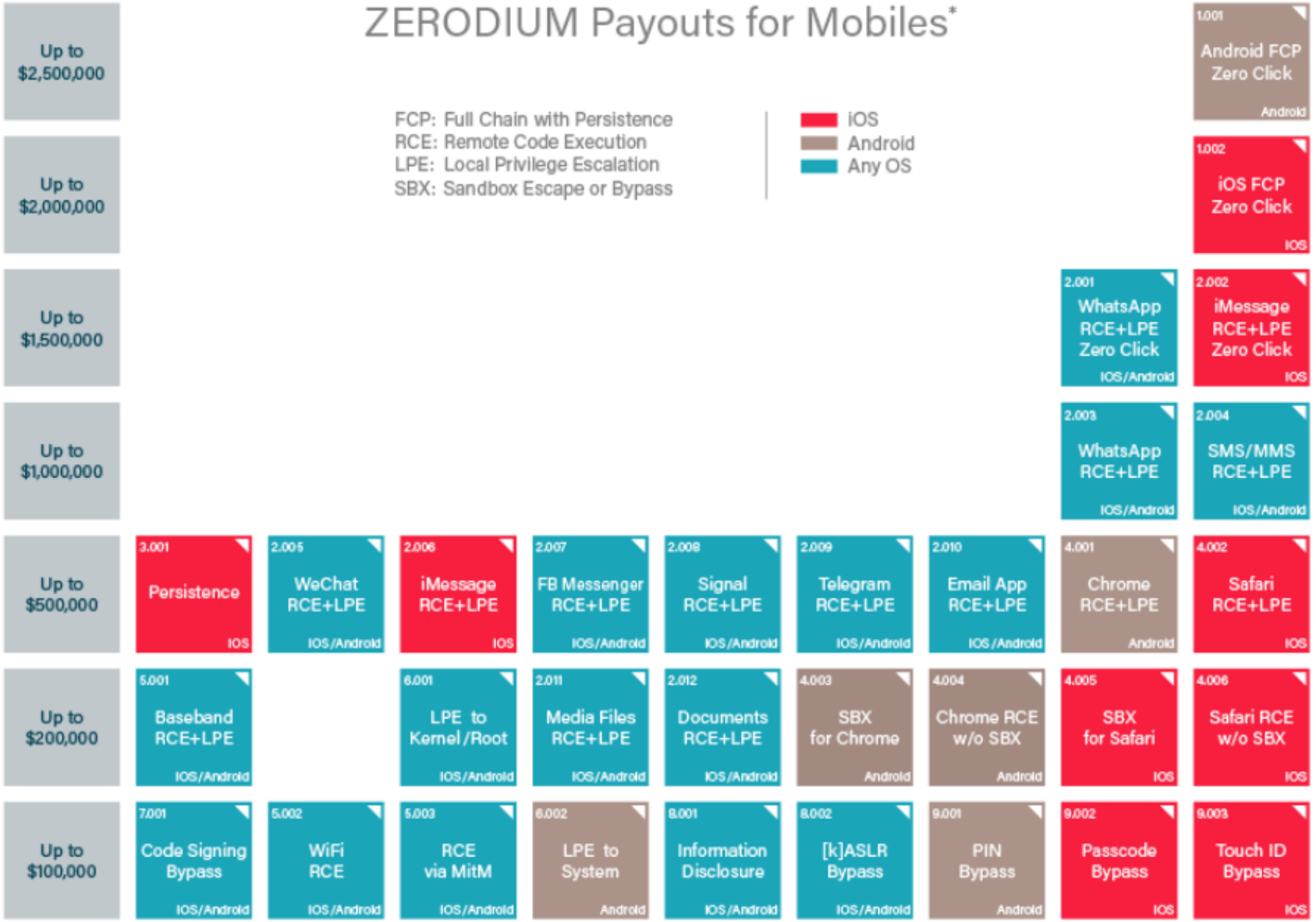
Un peu de vocabulaire

- Vulnérabilité : faiblesse dans le code
- Charge utile (payload), une action malicieuse : détruire des fichiers, faire un déni de service, augmenter ses privilèges, obtenir un terminal (shell), installer un cheval de troie (trojan)
- Exploit : exploite la vulnérabilité pour exécuter la charge utile

Code sécurisé

- Connaitre les vulnérabilités classiques (ou au moins savoir où chercher)
- Connaitre leur dangerosité (les exploiter)
- Connaitre les outils permettant de les détecter
- Connaitre les protections possibles
- Connaitre les forces et faiblesses des environnements de développement (langages, compilateurs, …)


ZERODIUM Payouts for Desktops/Servers*
Windows
macOS
Linux/BSD
Any OS
RCE: Remote Code Execution
LPE: Local Privilege Escalation
SBX: Sandbox Escape or Bypass
VME: Virtual Machine Escape
Up to $1,000,000
1.001 Win RCE Zero Click Win
Up to $500,000
3.001 Chrome RCE+LPE Win
2.001 Apache RCE Linux
2.002 MS IIS RCE Win
Up to $250,000
5.001 MS Outlook RCE Win
4.001 MS Exchange RCE Win
2.003 OpenSSL RCE Linux
2.004 PHP RCE Linux
Up to $200,000
6.001 VMware ESXi VME
5.002 Thunderbird RCE Win/Linux
4.002 Sendmail RCE Linux
4.003 Postfix RCE Linux
4.004 Dovecot RCE Linux
4.005 Exim RCE Linux
2.005 nginx RCE Linux
Up to $100,000
3.002 Safari RCE+LPE Mac
3.003 Edge RCE+LPE Win
3.004 Firefox RCE+LPE Win
5.003 Word/Excel RCE Win
7.001 WordPress RCE Linux
7.002 cPanel/WHM RCE Linux
7.003 Plesk RCE Linux
7.004 Webmin RCE Linux
Up to $80,000
6.002 VMware WS VME Win/Linux
5.004 Adobe PDF RCE+SBX Win
5.005 WinRAR RCE Win
5.006 7-Zip RCE Win
6.003 Windows LPE/SBX Win
Up to $50,000
6.004 USB LPE Win/Mac
8.001 Antivirus RCE Win
5.007 WinZip RCE Win
5.008 tar RCE Linux
6.005 macOS LPE/SBX Mac
6.006 Linux LPE Linux
6.007 BSD LPE BSD
Up to $10,000
9.001 Routers RCE
8.002 Antivirus LPE Win
7.005 phpBB RCE Linux
7.006 vBulletin RCE Linux
7.007 MyBB RCE Linux
7.008 Joomla RCE Linux
7.009 Drupal RCE Linux
7.010 Roundcube RCE Linux
7.011 Horde RCE Linux
* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.
2019/01 © zerodium.com



** All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.*

2019/09 © zerodium.com

Mais aussi

- https://www.zerodayinitiative.com

Un exemple pour s'échauffer

Un très mauvaise procédure d'authentification

Authentification

```
int  main(void)
{ const char *const pass = "monpassword";
  char *result;
  char  externpass[20];
  int ok;
  /* Read in the user's password */
            printf(Password:") ;
            gets (externpass) ;
            ok = strcmp (externpass, pass) == 0;
            puts(ok ? "Access granted." : "Access denied.");
            return ok ? 0 : 1;
}
```

strcmp

Le prototype, suivant la norme ISO/CEI 9899:1999, est le suivant :

```
int strcmp(const char *s1, const char *s2);
```

strcmp compare lexicalement les deux chaînes caractère par caractère et renvoie 0 si les deux chaînes sont égales, un nombre positif si s1 est lexicographiquement supérieure à s2, et un nombre négatif si s1 est lexicographiquement inférieure à s2.

```
int strcmp (const char* s1, const char* s2)
{
   while (*s1 != '\0' && (*s1 == *s2)) {s1++; s2++;}
   return (s1==s2) ? 0 : (*(unsigned char *)s1 - *(unsigned char *)s2);
}
```

gets

Description :

The C library function **char *gets(char *str)** reads a line from stdin and stores it into the string pointed to by str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.

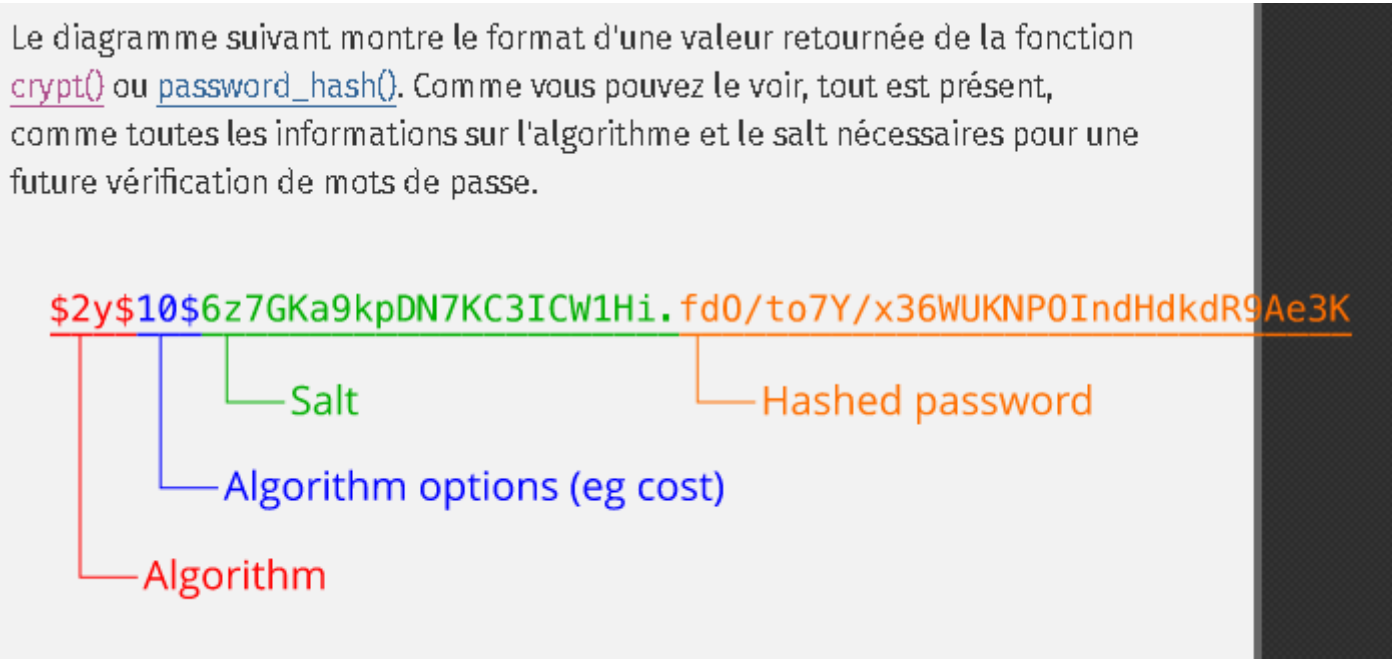
Return Value :

This function returns str on success, and NULL on error or when end of file occurs, while no characters have been read.

Les faiblesses

- Problème des tailles de buffer
  - Vérifier la taille en entrée
- gets est une fonction bannie (utiliser fgets)
- Les cas d'erreurs ne sont pas traités
- Password en clair, il faut hasher avec du sel
- Nettoyer externpass (mise à 0)
- Attaque sur le temps
- …

Stockage des mots de passe



Un site qui explique comment programmer le stockage et la vérification … et surtout quelles bibliothèques utiliser :

https://www.arsouyes.org/blog/2019/58_Store_Password_Hash

CWE Top 25 (2022)

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

Plan

- Vulnérabilités classiques :
  - Injection de code et vérification des entrées
  - Erreurs à l'exécution

- Les protections :
  - Codes sûrs
  - Protections à l'exécution
  - Protections de la plate-forme

1) Vérifier les entrées

Vérifier les entrées

- Entrées mal formatées qui provoque des comportements potentiellement nocifs
  - Injection de code (OS, SQL, ldap, …)
  - Accès non autorisés (répertoires, fichiers, …)

```sql
SELECT uid FROM Users WHERE name = '(nom)' AND password = '(mot de passe hashé)';
```

```sql
SELECT uid FROM Users WHERE name = 'Dupont';--' AND password = '4e383a1918b432a9bb7702f086c56596e';
```

Les caractères `--` marquent le début d'un commentaire en SQL. La requête est donc équivalente à :

```sql
SELECT uid FROM Users WHERE name = 'Dupont';
```

Injection de commandes

Soit le programme php suivant qui vise à liste le contenu du répertoire d'un user :

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

Attaque : on met l'entrée ;rm -rf /

Et on exécute ls -l /home/;rm -rf / !!!

Tous les langages sont sensibles … Solutions ?

solutions

- Vérifier les entrées : semble simple … mais en général non !
  - Être exhaustif, détecter les encodages de caractères, …
  - on écrit %3script%3 à la place de <script> (-:
  - Les formats complexes et dynamiques : format de fichiers, de messages
- Utiliser des commandes ou bibliothèques sûres

Defense Option 1: Avoid calling OS commands directly

The primary defense is to avoid calling OS commands directly. Built-in library functions are a very good alternative to OS Commands, as they cannot be manipulated to perform tasks other than those it is intended to do.

For example use `mkdir()` instead of `system("mkdir /dir_name")`.

If there are available libraries or APIs for the language you use, this is the preferred method.

cheatsheetseries.owasp.org/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.html

Les erreurs à l'exécution exploitables

- Vulnérabilités classiques
- Détection / protection

Sémantique d'un langage

- Vérification à la compilation / vérification à l'exécution :
  - dépend des langages
  - Limitations par le théorème de Rice

- Comportements à l'exécution :
  - Comportements nominaux
  - Cas d'erreurs
  - Comportements non spécifiés
  - Comportements non définis (UB)

=> Compilateur correct ?

C, C++ : quelques points de sémantique

- Les entiers : calculs dépendant de la représentation, wrap around, conversion implicite
- Les comportements non définis : déréférencer un pointeur null, conversion illicite, erreur dans l'initialisation des objets …
- Gestion de la mémoire : pas d'initialisation, gestion par le développeur

Des sites vulnérabilités/protections

- On aime bien le C (-:
  - 203 comportement non définis
  - 58 comportements non spécifiés

- Cert coding rules. Les règles pour le C :

https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard

+ liste des comportements indéfinis

Le guide de l'anssi :

www.ssi.gouv.fr/guide/regles-de-programmation-pour-le-developpement-securise-de-logiciels-en-langage-c/

Entier signé/non signé

Débordement possible sur les signés, troncature sur les unsigned :

operation	*domaine*	*valeur*
add_Sint_n	$x \in Sint_n \wedge y \in Sint_n \wedge x + y \in Sint_n$	$x + y$
add_Sint_n	$x \in Sint_n \wedge y \in Sint_n \wedge x + y \notin Sint_n$	*undefined behavior*
add_Uint_n	$x \in Uint_n \wedge y \in Uint_n$	$(x + y)\ mod\ 2^n$

(Usual arithmetic conversions) : si un opérande de type S et un opérande de type U alors l'opérande signé est converti en non signé.

```
int si = -1 ;
unsigned int ui = 1 ;
printf("%d\n", si < ui);
```

`si` est converti en non signé. Comme -1 est négatif il devient positif (la valeur est convertie en ajoutant la valeur maximum du type jusqu'à être dans le range). Et donc le test est faux.

Exemple

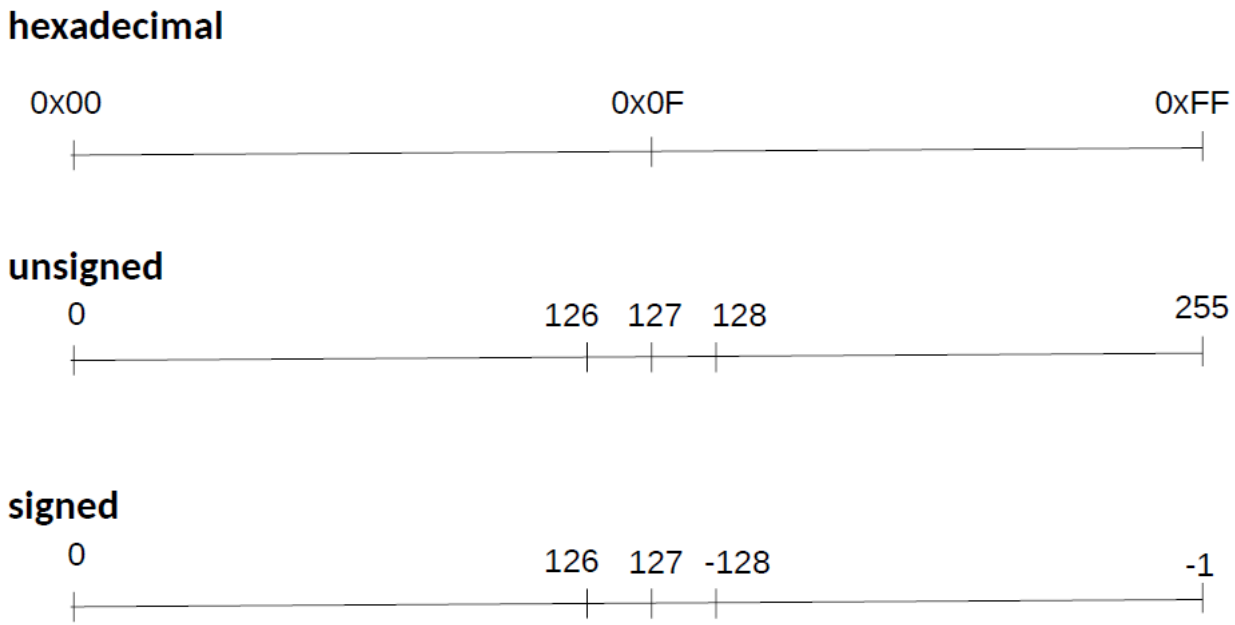
Norme C 99. Exemple sur une architecture 32-bit en complément à 2:

C type	domaine		définition
Signed	$Sint_n$	$=$	$-2^{n-1}..2^{n-1}-1$
Unsigned	$Uint_n$	$=$	$0..2^n-1$

short : n=16, int : n = 32, long n = 64
Les conversions:

operation	*domaine*	*valeur*	*condition*
$S2U_n$	$x \in Sint_n$	x $2^n + x$	*si* $x \geq 0$ *sinon*
$U2S_n$	$x \in Uint_n$	x $x - 2^n$	*si* $x \leq 2^{n-1}-1$ *sinon*

Complément à 2



Un programme vulnérable

```
Void vuln(unsigned nb, int *tab)
{
  int *dst ;
  dst = (int *) malloc(sizeof(unsigned int)*nb);
  if (!dst) return ;
  for (int i=0;i<nb;i++) dst[i]=tab[i];
}
```

Explication – 32 bits

- Sizeof renvoie un unsigned
- Si nb grand (>2^{30}) alors sizeof(unsigned int)*nb peut devenir petit (wrap around) et donc dst sera de petite taille, voire 0.
- Malloc(0) est undefined. Généralement renvoie un bloc de taille 0

Correction

Règle INT30-C. Ensure that unsigned integer operations do not wrap

```
Void vuln(unsigned nb, int *tab) {
  int *dst ;
 if (nb > UINT_MAX / sizeof(unsigned int)) return ;
  dst = (int *) malloc(sizeof(unsigned int)*nb);
  if (!dst) return ;
  for (int i=0;i<nb;i++) dst[i]=tab[i];
}
```

=> Beaucoup, beaucoup de CVE sur cette erreur …

Quelques vulnérabilités classiques : dangers et protections

Types d'attaques sur le code (1)

- Exploiter des buffers overflow
  - Modifier ou lire des mémoires normalement non accessibles
- Exploiter des integer overflow (qui peuvent provoquer des buffer overflow)
- Attaquer la pile/le tas
  - lire/modifier des valeurs : adresse de retour, variable contenant un pointeur de fonction, handler d'exception

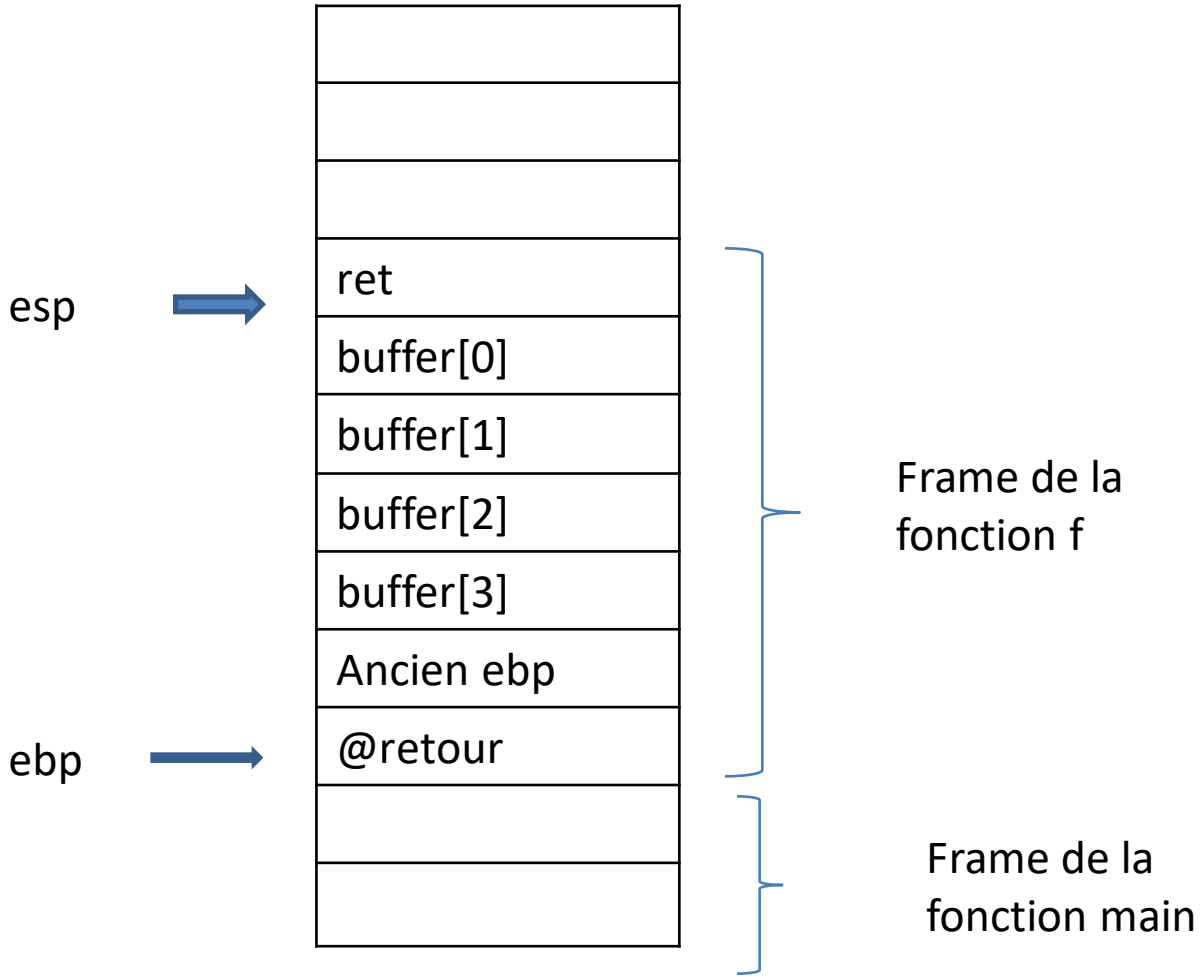
Modification du flot d'exécution

Smashing the Stack For Fun and Profit - Aleph One

```
Void f ()
{   char buffer[4] ;
    int * ret ;
    ret = buffer + 6 ;
    (*ret) = *ret +8 ; }                    /* on sautera x=1 ; */


Void main()
{   int x ;
    x = 0 ;
    f();
    x=1 ;
    printf (« %d \n », x);}
```

Pile à l'exécution (32 bits)



Exploitation classique

- Réécriture de l'adresse de retour avec une adresse sur un shellcode

- Si pas de protection par exemple en passant un tableau contenant du binaire

- Sinon on y arrive quand même …

Un shellcode

```
unsigned char code[] =
 "\x48\xc7\xc0\x3b\x00\x00\x00\x48"
 "\xc7\xc2\x00\x00\x00\x00\x49\xb8"
 "\x2f\x62\x69\x6e\x2f\x73\x68\x00"
 "\x41\x50\x48\x89\xe7\x52\x57\x48"
 "\x89\xe6\x0f\x05\x48\xc7\xc0\x3c"
 "\x00\x00\x00\x48\xc7\xc7\x00\x00"
 "\x00\x00\x0f\x05";

int main(int argc, char **argv) {
   int (*ret)() = (int(*)())code;
   ret(); }
```

Compiler avec la pile exécutable :

```
gcc -z execstack -o shellcode shellcode.c
```

La chaine binaire correspond à l'exécution du code suivant :

```
void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL)
   exit(0);
}
```

voir le site ci-dessous pour plus d'explications :

https://www.arsouyes.org/blog/2019/54_Shellcode

Protection à l'exécution contre les buffer overflows (1)

- Ajout d'un « canari » :

  Pile = … @retour ancien esb random buffer1[3] … buffer1[0] ret

⇒ Lors du branchement à l'adresse retour on vérifie le canari : si modifié alors arrêt de l'exécution

(StackGuard pour GCC 2.7.X, ProPolice pour GCC 4.1.x)

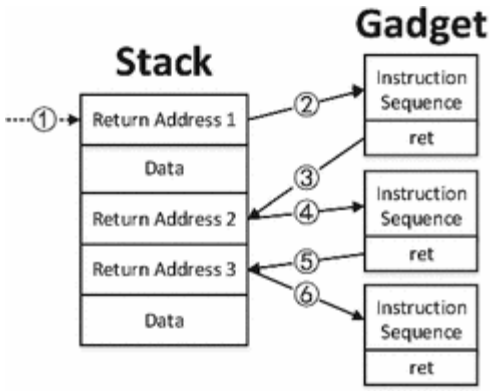
- Mémoire non exécutable (Data Execution Prevention)

- Plus difficile à protéger : heap overflow

Protection à l'exécution contre les buffer overflows (2)

- Ranger les différents segments (data, stack …) de manière aléatoire (ASLR, adress space layer randomization)
  - 2 exécutions ne travaillent pas sur les mêmes adresses mémoire

- Mémoire non exécutable (Data Execution Prevention, NX, W+X)
  - Return on libc
  - ROP (return oriented programming)



Types d’attaques sur le code (2)

- Remplacer des méthodes (remplacer un appel à une méthode virtuelle par un appel à une méthode statique)
- Variables non initialisées
- Exploiter la désallocation et les références pendantes (dangling pointer, UseAfterFree, DoubleFree)

Un exemple

```
typedef struct { void (*f)( void ); } st;
int main (int argc , char * argv [])
{  st *p1;
    char *p2;
    p1 =( st *) malloc ( sizeof (st));
    free (p1);
   p2= malloc ( sizeof (int));
   strcpy (p2 , argv [1]) ;
   p1 ->f();
return 0; }
```

Use after free

- Une vulnérabilité classiquement exploitée liée aux langages objet

https://cwe.mitre.org/data/definitions/416.html

- Protection : des gestionnaires de mémoire durcis
- AddressSanitizer: pas de réallocation

=> Des solutions partielles : souvent des allocateurs maison

Comment se protéger ?

Comment se protéger ?

Plusieurs niveaux :

- Ecrire du code sécurisé = Pas de vulnérabilités
- Surveiller l'exécution = détecter des actions malveillantes
- Protéger la plate-forme d'exécution = limiter l'impact d'une action malveillante

=> dépend de l'attaquant contre lequel on veut se protéger …

Les attaquants

- Attaquant externe : accès aux inputs/outputs
  - Vérifier les inputs et les outputs
- Attaquant connaissant le code de l'application:
  - Application robuste
- Attaquant sur la plate-forme d'exécution :
  - Plus compliqué …

Ecrire du code sécurisé

- Les bonnes pratiques de programmation
- Utiliser des bibliothèques sécurisées
- Vérifier les erreurs à l'exécution dans du code (statiquement/par test)
- Ajouter des protections à l'exécution pour détecter l'effet d'exécutions dangereuses

Outils d'analyse de code

- Des analyses dédiées :
  - Élévation de privilèges, manipulation de fichiers, vérification des APIs, …
  - Teinte

- https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers

- https://www.perforce.com/products/klocwork

Guide de l'anssi

- Règles de programmation

- Processus de compilation :
  - Compiler sans warning et sans erreur (avec options et optimisations)

  - Utiliser les fonctionnalités de sécurité des compilateurs

Les options de compilation quelques exemples

- L'option -Werror assure que le programme ne sera pas compilé tant que des messages d'avertissement subsistent. C'est une façon simple et radicale d'assurer que tous les avertissements seront sérieusement étudiés par le programmeur.

- L'option -fstack-protector de gcc protège la pile contre l'écrasement de l'adresse de retour d'une fonction par débordement de tampon.

- L’option -Wconversion de gcc permet de détecter certaines conversions de types implicites dangereuses.

- Exemple : une instruction unsigned int x = -1; génère une conversion implicite à l’exécution du programme.

5.3.4 Débordements d'entiers

Les débordements d'entiers signés ne sont pas définis par le standard C et sont donc particulièrement dangereux. Par exemple, selon les architectures matérielles et les compilateurs, une variable de type `int` atteignant la valeur `INT_MAX` peut boucler (*wrap*) après une nouvelle incrémentation, c'est-à-dire passer à la valeur `INT_MIN`, ce qui peut s'avérer très problématique notamment dans le cas d'une variable représentant un compteur de références à une allocation mémoire. Le compilateur peut être capable de détecter certains débordements d'entiers signés.

RECO 41

RECOMMANDATION – Activer les options du compilateur permettant de détecter les débordements d'entiers signés

GCC et CLANG supportent notamment l'option `-ftrapv`, qui conduit le compilateur à instrumenter le code source afin de générer une exception à l'exécution du programme pour tout débordement d'entiers signés lors d'une addition, d'une soustraction ou d'une multiplication.

Attention

Bien que les débordements d'entiers **non signés** constituent eux un comportement bien défini du C, ils n'en représentent pas moins un aspect périlleux et peuvent également conduire à l'introduction de bogues et de vulnérabilités dans un logiciel. Le développeur doit donc rester particulièrement prudent lorsqu'il effectue des opérations susceptibles de déborder, même avec des opérandes non signés.

Analyse statique et Compilation

- Les outils d'analyse statique viennent en complément des vérifications du compilateur
- Les vérifications faites par un compilateur en lien avec les options dépendent du niveau d'optimisation qui implémentent des analyses plus fines mais plus couteuses

RÈGLE – Activer un niveau d'optimisation raisonnablement élevé

Pour GCC et CLANG, le niveau d'optimisation doit être au moins -O1, et idéalement -O2 ou -Os.

Attention

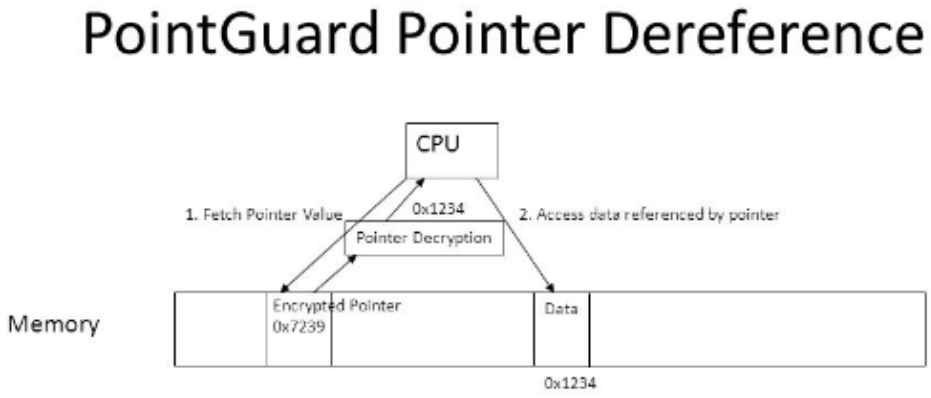
Le développeur doit s'assurer qu'un haut niveau d'optimisation n'élimine pas du code défensif ou des contre-mesures logicielles manuellement ajoutées.

Information

Par exemple, la ligne de commande minimale pour une compilation avec GCC ou CLANG est : `gcc/clang -O1 -Wall`[a] `-Wextra`[b] `-Wpedantic`[c] `-Werror -std=c99/c90`[d] `file.c -o file.exe`

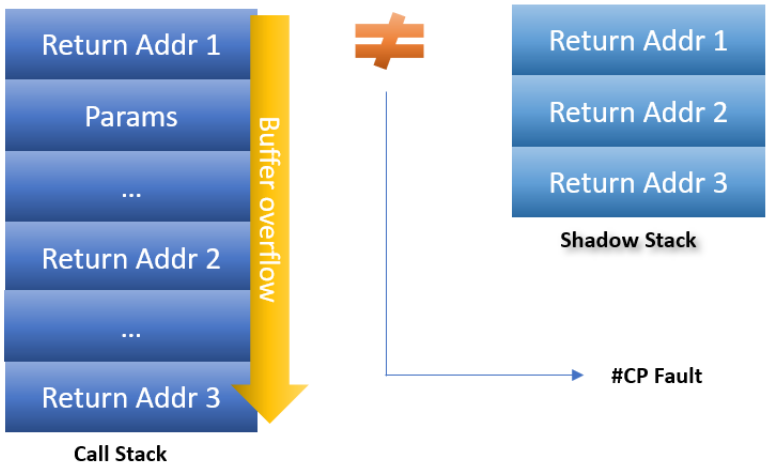
Les contre-mesures introduites à la compilation (1)

- Placement des objets, canaries
- Chiffrement des pointeurs
- Protection mémoire



Les contre-mesures introduites à la compilation (2)

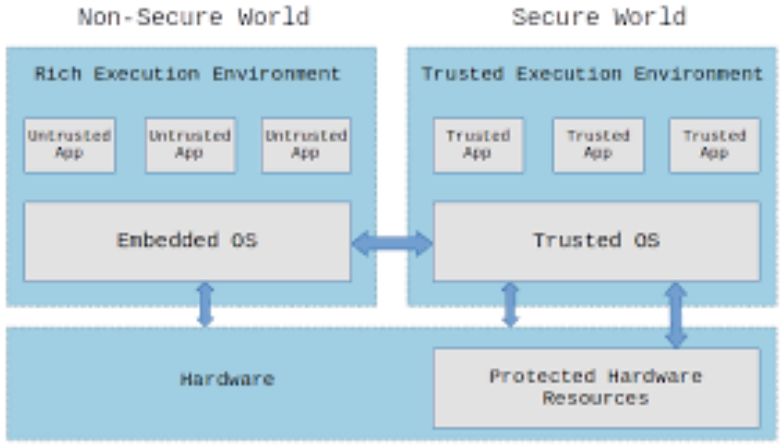
- Shadow memory et intégrité (@retour, meta-data du tas)
- Intégrité du flot :
  - Statique
  - dynamique
  - surapproximé



Les contre-mesures sur la plate-forme

- Mémoires segmentées (rwe)
- Randomisation des espaces mémoire (ASLR)

- Et aussi :
  - Contrôle d'accès à grain fin (accès et exécution)
  - Sandboxing, vm, …
- Et aussi : les TEE



Propriétés des langages

Langages de programmation : forces et faiblesses

- Les langages offrent des concepts plus ou moins adaptés pour la sécurité :

- Les garanties :
  - type safety, memory safety, thread safety …
  - Des librairies sûres et offrant des traitements complexes sûrs (i.e. sql input validation …)

- Pour toute exécution (à la compilation)/pour une exécution(runtime)

Type Safety

- Type safety : le compilateur et l'exécution garantissent que les chaînes de bits sont traitées suivant le type qui leur est associé.
  - Typage statique fort
  - Coercions et affectation de types vérifiées à l'exécution
- Détection statique ou dynamique des erreurs

Un exemple

```
char *c ;
   f() {char cc = ‘a’ ; c = &cc ;}
   g() {int i = - 99 ;   }
   main () { f() ; g() ; printf(« %c », *c) ; }
```

=> peut imprimer le caractère ÿ de code -1
(undefined dans la sémantique)

Préservation des types

- Si un programme s'exécute correctement (sans erreur) alors les valeurs à l'exécution sont du type correspondant aux déclarations.

$$e : t \quad \wedge \text{eval}(e, \text{mem}) = v$$
$$\Rightarrow v : t$$

voir contre-exemple en C (c pointe sur un entier qui n'est pas un caractère)

Memory safety

- Memory safety : le compilateur et l'exécution garantissent que le programme ne peut pas accéder à la mémoire en dehors d'une portion bien définie.
  - Pas d'accès direct à la mémoire
  - Runtime vérification des bornes de tableaux
  - Initialisation des objets
  - Pas de déallocation explicite (ramasse-miette)

- Détection statique ou dynamique des erreurs

UseAfterFree

```
Typedef struct Dummy { int a; int b; } Dummy ;
Void foo (void) {
Dumy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
Dummy * alias = ptr ;
free (ptr) ;
Int a = alias.a ;                    -- Use after Free
free(alias) ;                        -- Double Free
```

Garanties liées à memory safety

- Propriété de base :

- - on exécute P dans un état initial S1 contenant toutes les variables normalement accédées par le programme. Soit S1' l'état mémoire résultat.

- On exécute P dans S1 $\cup$ S2 avec :
  - S2 $\cap$ S1= $\varnothing$
  - S2 $\cap$ S1'= $\varnothing$

Et on obtient en sortie S1' $\cup$ S2

$\Rightarrow$ S2 pas modifié par l'exécution de P

$\Rightarrow$ Pas d'interférence avec l'extérieur (ni dans un sens ni dans l'autre)

Gestion mémoire – les solutions

- Langages avec désallocation explicite
- Langage sans désallocation

- Les solutions :
  - Compteur de références (smart pointeurs)
  - Ramasse-miette
  - Ownership

Des exemples

- Memory-unsafe, typed, type-unsafe : C, C++
- Memory-safe, typed, type-safe : Java, C#, Rust, Go
- Python : dynamically typed, type-safe, memory-safe

On ne peut pas avoir type-safety sans memory-safety

Safe arithmetic

Unsafest approach: leaving this as undefined behavior
– C and C++

Safer approach : specifying how over/underflow behaves
– based on 32 or 64 bit two complements behaviour
–Java and C#

Safer still : integer overflow results in an exception
– checked mode in C#

Safest : have infinite precision integers & reals, so overflow never happens
– python, some experiments in functional programming languages

Outils d'analyse statique

- Langage et ses propriétés R vérifiées à la compilation
  - exemple : typage statique fin (nullable/nonnullable)
- Analyse statique : vérifier les propriétés R' en supposant les propriétés R
  - R' couvre des comportements qu'on ne voudrait pas voir à l'exécution
- Génération de code : on suppose R (on pourrait combiner avec R')

Architecture de sécurité JVM

- le code n'est pas exécuté directement sur le matériel mais via une couche logicielle :
  - Vérificateur de byte-code : vérifier les bonnes propriétés du code
  - Chargeur de classes (class loader) : gère le chargement dynamique de code et détermine les droits d'une classe
  - Contrôleur d'accès (access controller) : gère les droits d'accès à l'exécution

=> Vise à maitriser le code qui sera exécuté et à protéger la plate-forme d'exécution (isolation)

Un point sur Rust

- Maitrise des formats entiers :
  - taille fixée, conversion explicite
  - wrap-around mais possibilité d'utiliser des bibliothèques qui détectent les wrap
- Vérification des bornes de tableau :
  - À l'exécution voire à la compilation (énumération)
- Maitrise de la libération mémoire :
  - Notion d'ownership qui garantit à la compilation l'absence de références pendantes
- Mais le mot clé unsafe)-:

Initialisation des variables

Vis-a-vis des variables non initialisées, les langages de programmation adoptent différentes solutions :

1. rien n'est fait
2. détection d'une erreur a l'exécution
3. initialisation par défaut par le compilateur
4. vérification de l'initialisation des variables par le compilateur

Quels exemple de langages ?

Discuter de ces différentes solutions vis-a-vis des critères suivants :

1. coût de la solution
2. erreurs potentielles induites pour le développeur
3. assurances en sécurité

Déréférencer le pointeur null

Les langages de programmation adoptent différentes solutions :

1. rien n'est fait
2. détection d'une erreur a l'exécution
3. Type nullable/non-null (Rust, Kotlin)
4. vérification par le compilateur

Discuter de ces différentes solutions vis-a-vis des critères suivants :

1. coût de la solution
2. erreurs potentielles induites pour le développeur
3. assurances en sécurité