EXPLOITATION DE BINAIRES, REVERSE ENGINEERING : RAPPELS

Gauvain Roussel-Tarbouriech (gauvain@govanify.com)

REVERSE ENGINEERING SOUS GHIDRA

Partez de ce que vous connaissez, que ce soit un string ou l'entrypoint (si le binaire est petit !)

REVERSE ENGINEERING SOUS GHIDRA

Partez de ce que vous connaissez, que ce soit un string ou l'entrypoint (si le binaire est petit !)

Toutes les vues de Ghidra sont synchronisées, sélectionnez dans une fenêtre pour voir ce a quoi correspond cette selection dans une autre (c-à-d asm vs. pseudocode)

PSEUDOCODE

N’oubliez pas de retyper les variables et les renommer selon vos déductions

PSEUDOCODE

N’oubliez pas de retyper les variables et les renommer selon vos déductions

Si un bout de code fait pas mal de maths avec des constantes, cherchez ces constantes sur votre moteur de recherche/GitHub

PSEUDOCODE

N’oubliez pas de retyper les variables et les renommer selon vos déductions

Si un bout de code fait pas mal de maths avec des constantes, cherchez ces constantes sur votre moteur de recherche/GitHub

Le pseucode n’est pas toujours fiable, n’hésitez pas à comparer avec l’assembleur si vous ne comprenez pas exactement une partie

PSEUDOCODE

Si vous avez une variable pas mal utilisée comme X+{1,2,3,4,8,...} alors c'est surement soit un array soit une structure. Les structures peuvent être créés automatiquement sous ghira, et vous pouvez retyper la variable en array. Gardez ce qui fait le plus sens !

```
case 0x41:
  *(undefined4 *)(enemy + 0xe60) = *(undefined4 *)(ecl_instr + 0xc);
  *(undefined4 *)(enemy + 0xe64) = *(undefined4 *)(ecl_instr + 0x10);
  *(undefined4 *)(enemy + 0xe68) = *(undefined4 *)(ecl_instr + 0x14);
  *(undefined4 *)(enemy + 0xe6c) = *(undefined4 *)(ecl_instr + 0x18);
  *(byte *)(enemy + 0xe52) = *(byte *)(enemy + 0xe52) | 1;
  break;
```

```
case 0x41:
  enemy->screen_box[0] = *(float *)ecl_instr->params;
  enemy->screen_box[1] = *(float *)(ecl_instr->params + 4);
  enemy->screen_box[2] = *(float *)(ecl_instr->params + 8);
  enemy->screen_box[3] = *(float *)(ecl_instr->params + 0xc);
  enemy->flags[2] = enemy->flags[2] | 1;
  break;
```

LES XREFS

LES XREFS

Les XRefs vous permettent de trouver comment et ou une fonction, constante ou valeur globale est utilisée

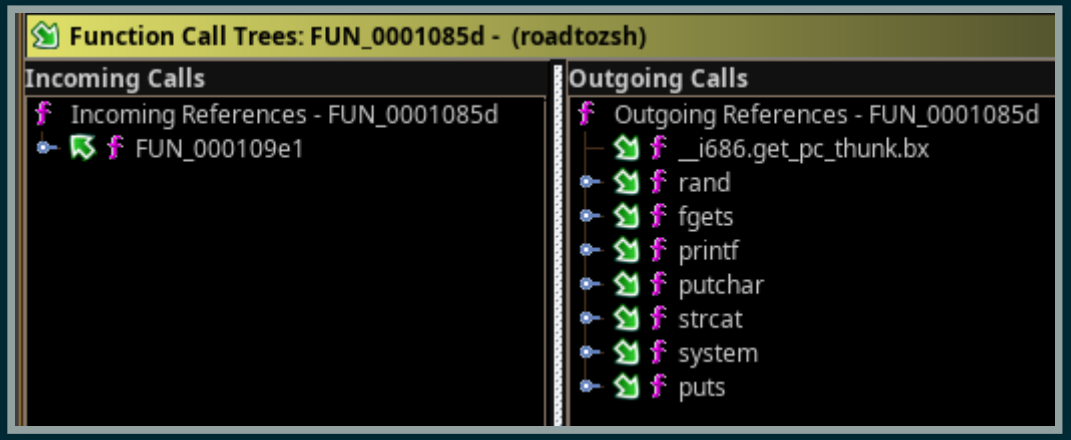
LES XREFS

Les XRefs vous permettent de trouver comment et ou une fonction, constante ou valeur globale est utilisée

Lexique entre paranthèse : c = call, R = read, W = write, * = inconnu

LES XREFS

Vous pouvez aussi trouver tous les appels et utilisations de fonction dans une fonction sélectionnée avec Windows → Function Call Trees



REVERSE DYNAMIQUE

REVERSE DYNAMIQUE

Utilisez gdb pour voir dynamiquement comment certaines fonctions fonctionnent

REVERSE DYNAMIQUE

Utilisez gdb pour voir dynamiquement comment certaines fonctions fonctionnent

Vous pouvez patcher temporairement les valeurs de retour dans Ghidra pour voir si le résultat est plus lisible dans le pseudocode right click → Patch Instruction

MITIGATIONS

NX (No eXecute) :
rend la stack non exécutable. Empêche l'exécution de shellcode

ASLR (Address Space Layout Randomization) :
rend les adresses de librairies, stack et heap aléatoire à chaque exécution du programme. Rends, entre autre, le ROP plus difficile

PIE (Position Independant Executable) :
met le code assembleur du programme a une région aléatoire de la mémoire à chaque exécution du processus. Rends le ROP plus difficile

Stack Cookie :
met toutes les variables locales avant les variables de type buffer dans la stack et ajoute une valeur aléatoire avant la valeur de retour et la vérifie avant chaque exécution. Rends le ROP plus difficile

ROP

Return Oriented Programming. C’est une technique d’exploitation réutilisant du code déjà existant. Indispensable si le NX est présent

ROP

Return Oriented Programming. C’est une technique d’exploitation réutilisant du code déjà existant. Indispensable si le NX est présent

Peut être exploité quand on peut réécrire l’adresse de retour dans la stack (grace à, par exemple, un buffer overflow)

BUFFER OVERFLOW

On écrit par-delà un buffer ce qui réécrit l'adresse de retour dans la stack

BUFFER OVERFLOW

On écrit par-delà un buffer ce qui réécrit l'adresse de retour dans la stack

On peut savoir où elle est réécrite en créant un pattern de brujin et regardant quelle valeur contient `eip`

BUFFER OVERFLOW

On écrit par-delà un buffer ce qui réécrit l'adresse de retour dans la stack

On peut savoir où elle est réécrite en créant un pattern de brujin et regardant quelle valeur contient `eip`

Si le programme segfault avant de réécrire cette adresse regardez la première valeur sur la stack dans gdb au moment du `ret` fatidique

GADGETS

Un gadget est une partie de code assembleur finissant par un ret, on peut en trouver avec ROPgadget

GADGETS

Un gadget est une partie de code assembleur finissant par un ret, on peut en trouver avec ROPgadget

Ainsi le code s'exécute puis exécute la prochaine adresse sur la ropchain

GADGETS

Un gadget est une partie de code assembleur finissant par un ret, on peut en trouver avec ROPgadget

Ainsi le code s'exécute puis exécute la prochaine adresse sur la ropchain

Dans les prochaines slides, si le registre dans lequel on pop n'est pas précisé il n'est pas important, sinon il est crucial

ARGUMENTS

Chaque fonction prends des arguments soit sur la stack soit sur un registre

ARGUMENTS

Chaque fonction prends des arguments soit sur la stack soit sur un registre

On peut le savoir pour une fonction en particulier en regardant ghidra

ARGUMENTS

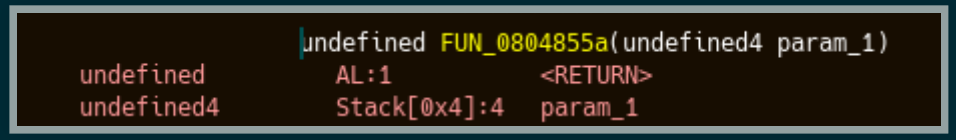
Chaque fonction prends des arguments soit sur la stack soit sur un registre

On peut le savoir pour une fonction en particulier en regardant ghidra

Si la fonction n’est pas présente, faites un binaire de test sur la même plateforme avec la même architecture ! (32 vs. 64bits)

ARGUMENTS

Exemple stack



ARGUMENTS

Exemple stack

```
                    undefined FUN_0804855a(undefined4 param_1)
undefined             AL:1           <RETURN>
undefined4            Stack[0x4]:4   param_1
```

Exemple registre

```
                    thunk int system(char * __command)
                      Thunked-Function: <EXTERNAL>::system
int                   EAX:4          <RETURN>
char *                RDI:8          __command
```

ARGUMENTS

Imaginons que function_reg prenne en argument le registre rdi et qu'on veuille faire l'appel `function_reg(12)`

ARGUMENTS

`pop rdi; ret`	`12`	`fonction_reg`

rdi=inconnu

ARGUMENTS

rdi=inconnu

pop est exécuté, 12 est enlevé de la stack, rdi=12

ARGUMENTS

rdi=inconnu
pop est exécuté, 12 est enlevé de la stack, rdi=12
ret est exécuté, on continue

ARGUMENTS

pop est exécuté, 12 est enlevé de la stack, rdi=12
ret est exécuté, on continue
`function_reg` est exécuté

ARGUMENTS

Imaginons que function_stack prenne en argument le Stack[+0x4] et qu'on veuille faire l'appel `function_stack(12)`

ARGUMENTS

Stack[0x0]	Stack[0x4]	Stack[0x8]
fonction_stack	pop; ret	12

function_stack n'est pas encore appellé

ARGUMENTS

Stack[-0x4]	Stack[0x0]	Stack[0x4]
fonction_stack	pop; ret	12

function_stack n'est pas encore appellé

function_stack est appelé, la stack se met à jour, Stack[0x4] est utilisé comme argument

ARGUMENTS

Stack[-0x8]	Stack[-0x4]	Stack[0x0]
fonction_stack	pop; ret	12

function_stack est appelé, la stack se met à jour, Stack[0x4] est utilisé comme argument

Le stack est màj, pop est appellé et enlève l'argument de la stack

ARGUMENTS

Stack[-0xC]	Stack[-0x8]	Stack[-0x4]
fonction_stack	pop; ret	12

Le stack est màj, pop est appellé et enlève l'argument de la stack

Le stack est nettoyé, on peut continuer l'exécution

ARGUMENTS

P.S: Si on a plus d'un argument sur la stack il faudra trouver un gadget avec autant de pop que d'arguments suivi d'un ret.

RET2LIBC

On veux lancer un /bin/sh mais notre programme n'a aucun gadget utile

RET2LIBC

On veux lancer un /bin/sh mais notre programme n'a aucun gadget utile

- On obtiens l'adresse de la libc dans la mémoire (infoleak)

RET2LIBC

On veux lancer un /bin/sh mais notre programme n'a aucun gadget utile

- On obtiens l'adresse de la libc dans la mémoire (infoleak)
- On appelle `system("/bin/sh")`!

RET2LIBC

On veux lancer un /bin/sh mais notre programme n'a aucun gadget utile

- On obtiens l'adresse de la libc dans la mémoire (infoleak)
- On appelle `system("/bin/sh")`!

Permet de palier au manque de gadgets utiles

INFOLEAK

Deux cas possible pour du ROP/ret2libc :

INFOLEAK

Deux cas possible pour du ROP/ret2libc :

- Sans PIE et stack cookie : on peut faire une ropchain qui va imprimer l'adresse de __libc_start_main directement

INFOLEAK

Deux cas possible pour du ROP/ret2libc :

- Sans PIE et stack cookie : on peut faire une ropchain qui va imprimer l'adresse de __libc_start_main directement
- Avec PIE et stack cookie : on a besoin d'un infoleak nous permettant de lire au minimum la stack. Un string format peut faire l'affaire

INFOLEAK

Deux cas possible pour du ROP/ret2libc :

- Sans PIE et stack cookie : on peut faire une ropchain qui va imprimer l'adresse de __libc_start_main directement
- Avec PIE et stack cookie : on a besoin d'un infoleak nous permettant de lire au minimum la stack. Un string format peut faire l'affaire

Un infoleak peut être utile dans bien d'autre cas !

RACE CONDITION

Un bug logique dans lequel on peut modifier une variable pendant que le programme l’utilise et ne la vérifie pas

RACE CONDITION

Un bug logique dans lequel on peut modifier une variable pendant que le programme l'utilise et ne la vérifie pas

Aussi appellée TOCTOU (Time Of Check Time Of Use) pour les fous

RACE CONDITION

Un bug logique dans lequel on peut modifier une variable pendant que le programme l'utilise et ne la vérifie pas

Aussi appellée TOCTOU (Time Of Check Time Of Use) pour les fous

Exemple : le programme écrit dans un fichier temporaire et le réutilise plus tard, on peut réécrire ce fichier

COMMAND INJECTION

Un bug logique possible lorsqu'on laisse l'utilisateur renseigner des arguments d'une commande.

COMMAND INJECTION

Un bug logique possible lorsqu'on laisse l'utilisateur renseigner des arguments d'une commande.

Si non-sanitisés, l'utilisateur peut escape et exécuter les commandes qu'il veux, exemple : "cat test;/bin/sh"

COMMAND INJECTION

Un bug logique possible lorsqu'on laisse l'utilisateur renseigner des arguments d'une commande.

Si non-sanitisés, l'utilisateur peut escape et exécuter les commandes qu'il veux, exemple : "cat test;/bin/sh"

Un autre exemple connu : les SQLi (injections SQL)

FORMAT STRING

Un bug présent lorsqu'un programme printf un input que l'utilisateur controle

FORMAT STRING

Un bug présent lorsqu'un programme printf un input que l'utilisateur controle

Le printf charge ses arguments dans la stack. Il est possible d'abuser cette technique afin de jouer avec la stack

- %x : Affiche le paramètre en hexadécimal

- %x : Affiche le paramètre en hexadécimal
- %n : Écrit le nombre de charactères imprimés dans l'argument traité en tant que pointeur

- %x : Affiche le paramètre en hexadécimal
- %n : Écrit le nombre de charactères imprimés dans l'argument traité en tant que pointeur
- %.33d : Écrit 33 charactères

- %x : Affiche le paramètre en hexadécimal
- %n : Écrit le nombre de charactères imprimés dans l'argument traité en tant que pointeur
- %.33d : Écrit 33 charactères
- %2$x : Affiche le second paramètre dans la stack. Fonctionne avec n'importe quels charactères spéciaux de printf

- %x : Affiche le paramètre en hexadécimal
- %n : Écrit le nombre de charactères imprimés dans l'argument traité en tant que pointeur
- %.33d : Écrit 33 charactères
- %2$x : Affiche le second paramètre dans la stack. Fonctionne avec n'importe quels charactères spéciaux de printf

Exemple : %.65536d%35$n écrit la valeur 65535 (0x10000) sur l'élément pointé par la 35ème valeur sur la stack