

Security and privacy

RAPPORT ANALYSE STATIQUE

Mohamed BOUCHENGUOUR
Hamadi DAGHAR

MASTER 2 CYBERSECURITE
2024-2025



Table des matières

Exercice 1	2
Exercice 2	2
Exercice 3	4
Exercice 4	5
Exécutable 1	9
Exécutable 2	10
Spark.exe.....	11
SYMBOL TREE	11
RECHERCHE DE CHAINE	12
Les programmes qui pourraient interagir avec notre malware	12
Des noms d'utilisateurs ou de machines	12
Recherche de dll	13
Fichier sys.....	13
Clés de registre.....	13
Requête php	14
OSINT	14
Analyse de rk.sys	16
Conclusion	17

Exercice 1

On commence facile... Essayez de lancer le programme dans un terminal pour voir ce qu'il fait...

On met les droits pour pouvoir exécuter exo1, on l'exécute et le flag est fourni directement.

```
(root@kali)-[/home/.../TP1/TP1/Part-1/CTF-IDA]
# ./exo1
flag{not_that_kind_of_elf}
```

Exercice 2

Une fois le programme lancé, essayez de l'utiliser. Ensuite, posez-vous la question suivante : Comment fonctionne algorithmiquement le programme / Comment feriez-vous si vous deviez l'implémenter ? Une fois cette réflexion faite, essayez de trouver une commande, un outil ou une méthode permettant de récupérer l'information. Comment feriez-vous pour rendre ce programme plus robuste (de manière spéculative, pas besoin d'implémenter vos réflexions) ?

On lance une première fois le programme et on voit qu'un mot de passe est demandé.

```
(kali@kali)-[~/.../TP1/TP1/Part-1/CTF-IDA]
$ ./exo2
Usage: ./exo2 password
```

On saisit un mot de passe aléatoire pour voir la réaction du programme.

```
(kali@kali)-[~/.../TP1/TP1/Part-1/CTF-IDA]
$ ./exo2 test
Access denied.
```

Ici, on s'aperçoit que le programme attend un mot de passe spécifique. Algorithmiquement, nous aurions fait un programme qui récupère la saisie de l'utilisateur, puis le compare dans un if avec la valeur attendu (stockée dans une variable). Si la valeur saisie et attendu sont identique, on affiche le flag, sinon, on affiche un message de refus. Un des outil/programme pour récupérer l'information est strings. En exécutant string sur exo2, ces informations sont affichées :

```

(kali@kali)-[~/.../TP1/TP1/Part-1/CTF-IDA]
└─$ strings ./exo2
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
puts
printf
memset
strcmp
__libc_start_main
/usr/local/lib:$ORIGIN
_gmon_start__
GLIBC_2.0
PTRh
j3jA
[^_]
UWVS
t$,U
[^_]
Usage: %s password
super_secret_password
Access denied.
Access granted.
;+2$(
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
crtstuff.c
__JCR_LIST__

```

On voit apparaître plusieurs informations intéressantes dans l'exécutable, notamment la chaîne **super_secret_password**, ce qui indique que le mot de passe est stocké en clair.

Les messages **Access granted.** et **Access denied.** confirment une logique binaire : si le bon mot de passe est entré, l'accès est accordé, sinon, il est refusé.

On teste le mot de passe, et si celui-ci est correct, le flag s'affiche, confirmant que l'extraction était juste.

```

(kali@kali)-[~/.../TP1/TP1/Part-1/CTF-IDA]
└─$ ./exo2 super_secret_password
Access granted.
flag{if_i_submit_this_flag_then_i_will_get_points}

```

De plus, le mot de passe aurait pu être obtenu avec une attaque par bruteforce, car il n'y a aucune limite d'essais ni de délai entre les tentatives, même si une telle attaque aurait pris beaucoup plus de temps en raison de la longueur de la chaîne.

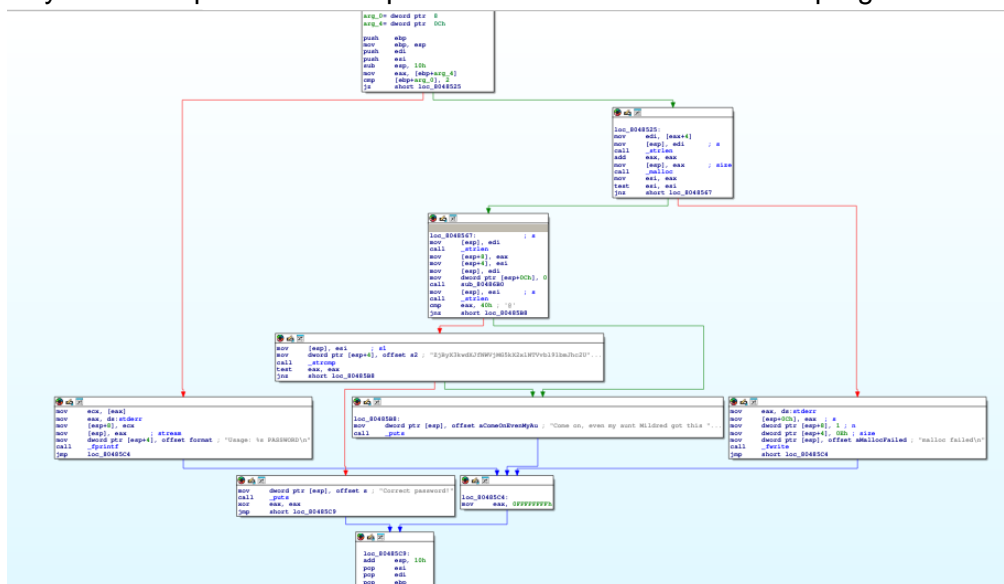
Pour rendre ce programme plus robuste, il faudrait éviter de stocker le mot de passe en clair en utilisant un hash plutôt qu'une simple comparaison, empêchant son extraction via strings. Ajouter un délai entre chaque tentative limiterait le bruteforce, tout comme un nombre maximum d'essais avant blocage. Enfin, déplacer la vérification sur un serveur compliquerait le reverse engineering. En combinant ces approches, on renforcerait la sécurité du programme face aux attaques courantes.

Exercice 3

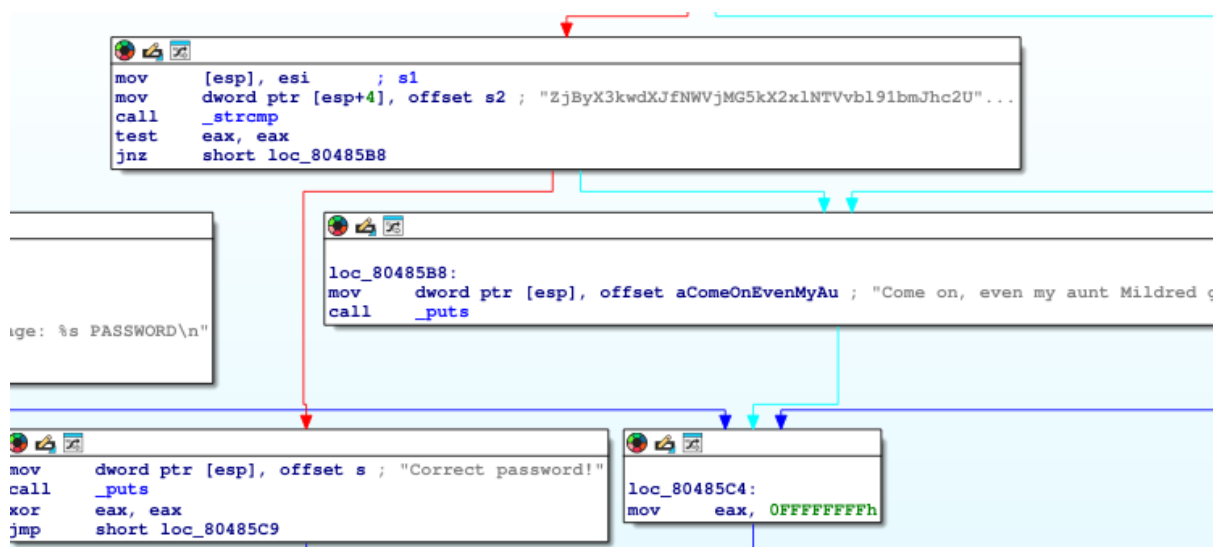
Les outils de RSE sont vos amis. Essayez de récupérer le mot de passe. Note : Il est bien sûr possible de récupérer l'information avec d'autres outils ou commandes (notamment celle que vous auriez pu utiliser dans l'exercice précédent). De plus, si vous avez des soucis avec l'installation d'IDA et que vous voulez utiliser autre chose que Ghidra, vous pouvez utiliser Radare2 (commande r2 déjà présente dans Kali).

Comment feriez-vous pour rendre ce programme plus robuste (de manière spéculative, pas besoin d'implémenter vos réflexions) ?

Nous avons analysé le programme avec IDA pour examiner sa fonction **main()** (voir capture). Cette analyse nous a permis de comprendre le fonctionnement de ce programme.



Ce programme utilise la fonction native strcmp pour comparer 2 chaînes, s1 et s2. On peut facilement supposer que s1 est la chaîne que l'utilisateur rentre et s2 est un string mis en brut dans le programme
 ZjByX3kwdXJfNWVjMG5kX2x1NTVvbl91bmJhc2U2NF80bGxfN2gzXzd0MW5nNQ==.



Nous avons donc testé avec la valeur de s2 en tant que mot de passe, le flag n'est toujours pas récupéré. Néanmoins cette chaîne de caractère ressemble fortement à une chaîne de caractère qui a été encodé en base64.

```
(root@kali)-[/home/.../TP1/TP1/Part-1/CTF-IDA]
# ./exo3 ZjByX3kwdXJfNWVjMG5kX2x1NTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==
```

En utilisant un simple décodeur de base 64, nous trouvons la chaîne de caractères f0r_y0ur_5ec0nd_le55on_unbase64_4ll_7h3_7h1ng5 qui est bien le mot de passe qu'on doit rentrer.

```
(root@kali)-[/home/.../TP1/TP1/Part-1/CTF-IDA]
# ./exo3 f0r_y0ur_5ec0nd_le55on_unbase64_4ll_7h3_7h1ng5
Correct password!
```

Pour améliorer la sécurité du programme, il existe plusieurs façons :

- Exécution en plusieurs étapes : Fragmenter la vérification du mot de passe sur plusieurs fonctions et conditions pour compliquer l'analyse du flux d'exécution.
- Utilisation de valeurs dynamiques : Générer des valeurs de comparaison à l'exécution au lieu de les stocker en dur dans le binaire.
- Obfuscation du code : Utiliser des techniques comme le chiffrement des chaînes sensibles en mémoire et leur déchiffrement à l'exécution pour éviter leur extraction via IDA.

Exercice 4

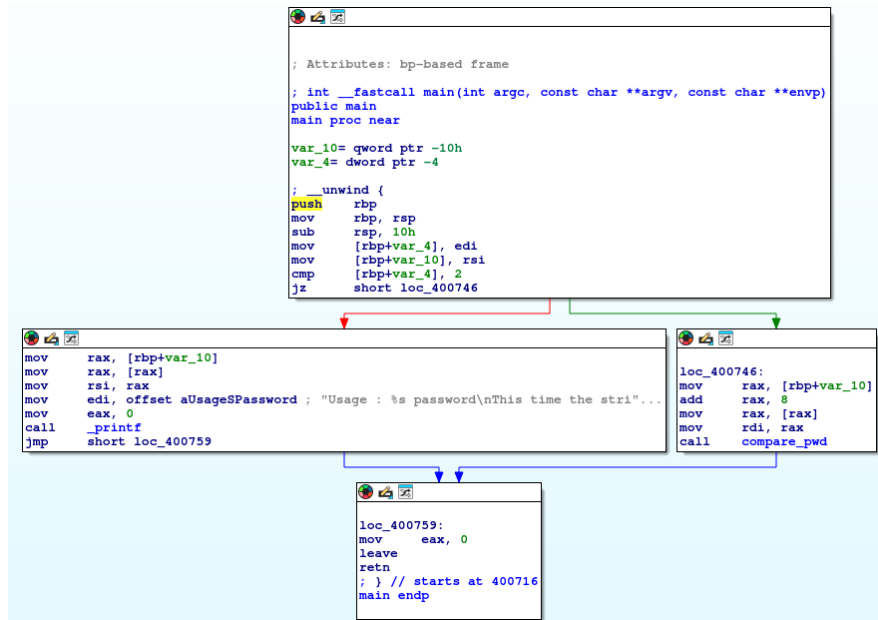
On lance une première fois l'exo sans paramètre et on remarque qu'il y a le message "This time the string is hidden and we used strcmp".

```
(kali@kali)-[~/Desktop/TP1/Part-1/CTF-IDA]
$ ./exo4
Usage : ./exo4 password
This time the string is hidden and we used strcmp
```

On relance l'exo mais cette fois mais avec un paramètre (test) et on on voit le message "password "test" not ok".

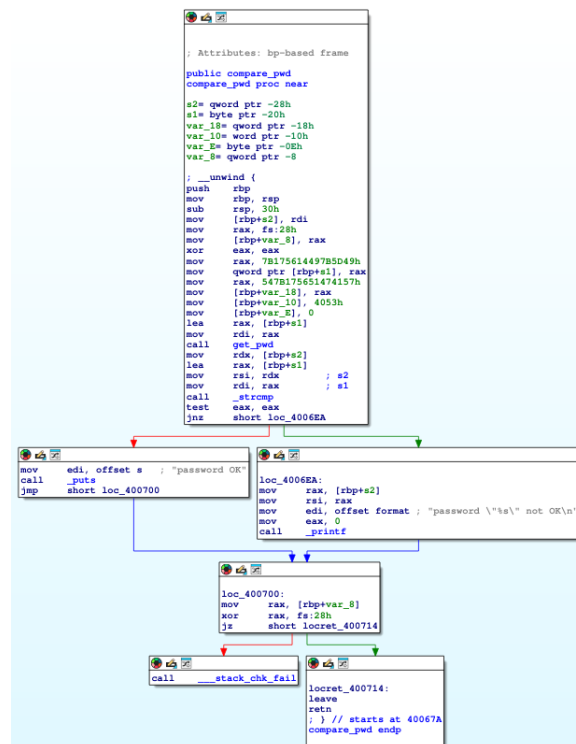
```
(kali@kali)-[~/Desktop/TP1/Part-1/CTF-IDA]
$ ./exo4 test
password "test" not OK
```

Nous avons analysé le programme avec **IDA** afin de voir si le mot de passe est stocké en clair ou en base64 comme pour les exercices précédents mais ce n'est pas le cas. Nous partons donc dans l'analyse de main.



Dans la fonction **main()**, on observe une structure conditionnelle (if). On retrouve également le message "This time the string is hidden and we used strcmp", ce qui nous donne une première indication sur le fonctionnement du programme.

En effet, cela nous permet de comprendre que la partie gauche du graphe correspond au cas où aucun paramètre n'est fourni, ce qui entraîne l'affichage du message "This time the string is hidden and we used strcmp" et l'arrêt du programme. Cependant, si un paramètre est saisi, l'exécution suit le chemin de droite et appelle la fonction **compare_pwd**. C'est donc dans cette fonction que la vérification du mot de passe est effectuée.



En entrant dans **compare_pwd**, on remarque une double logique conditionnelle : deux branches distinctes mènent soit à l'affichage de "password OK", soit à "password <input> not ok". Cela signifie que c'est dans cette fonction que se trouve la vérification du mot de passe.

En regardant plus en détail, on voit un appel à **get_pwd**. On peut supposer que cette fonction récupère le mot de passe attendu de manière "sécurisée", plutôt que de le stocker directement en clair. Cela explique pourquoi une recherche avec strings ne suffit pas cette fois-ci.

Après l'appel à **get_pwd**, le programme effectue ensuite un **strcmp**, ce qui confirme que la comparaison entre le mot de passe saisi et le mot de passe stocké est effectuée à cet endroit précis.

Sachant que **strcmp** prend deux arguments (**s1** et **s2**), nous pouvons poser un **breakpoint** juste avant son exécution pour examiner les valeurs qui lui sont passées en paramètre.

À ce stade, nous ne savons pas encore si le mot de passe attendu sera stocké dans **s1** ou **s2**. Pour le vérifier, nous lançons le programme avec un **breakpoint** sur **strcmp**, ce qui nous permettra d'afficher les valeurs des registres **RDI** et **RSI** au moment de la comparaison.

On s'intéresse à **RDI** et **RSI** car, par convention d'appel en **x86-64**, les deux premiers arguments d'une fonction sont toujours passés via ces registres. **RDI** reçoit le premier argument, qui correspond à **s1** dans **strcmp**, tandis que **RSI** reçoit le second argument, qui correspond à **s2**.

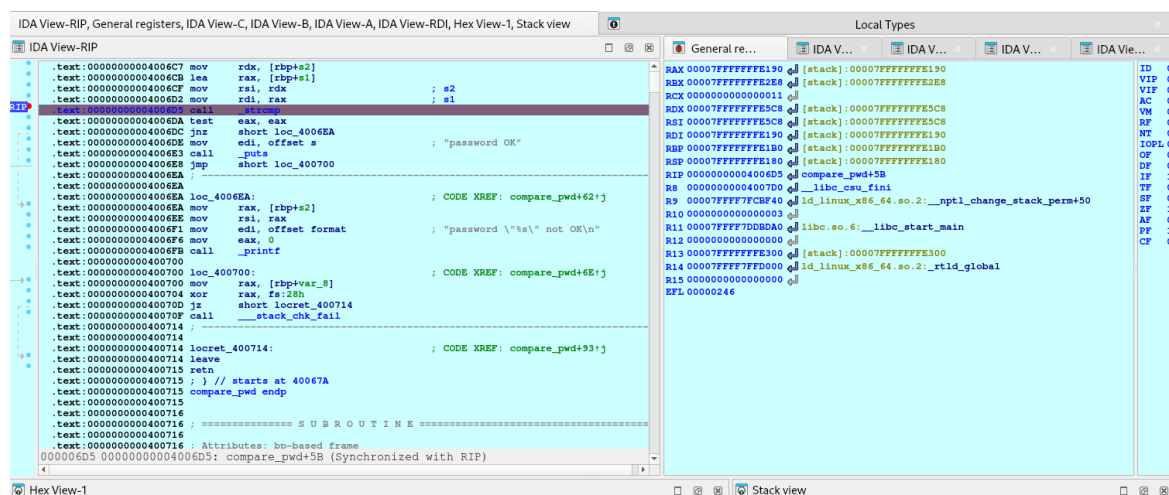
```

; Attributes: bp-based frame
public compare_pwd
compare_pwd proc near

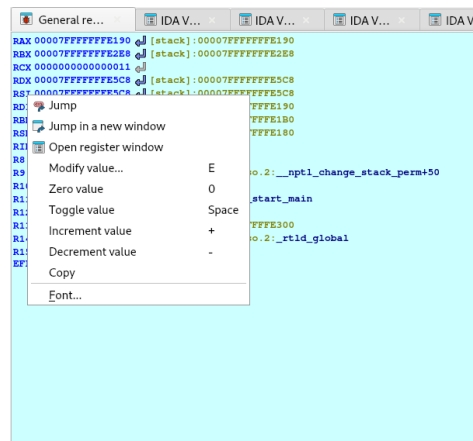
s2= qword ptr -28h
s1= byte ptr -20h
var_18= qword ptr -18h
var_10= word ptr -10h
var_E= byte ptr -0Eh
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     [rbp+s2], rdi
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
mov     rax, 7B175614497B5D49h
mov     qword ptr [rbp+s1], rax
mov     rax, 547B175651474157h
mov     [rbp+var_18], rax
mov     [rbp+var_10], 4053h
mov     [rbp+var_E], 0
lea     rax, [rbp+s1]
mov     rdi, rax
call    get_pwd
mov     rdx, [rbp+s2]
lea     rax, [rbp+s1]
mov     rsi, rdx
mov     rdi, rax
call    strcmp
test    eax, eax
jnz     short loc_4006EA

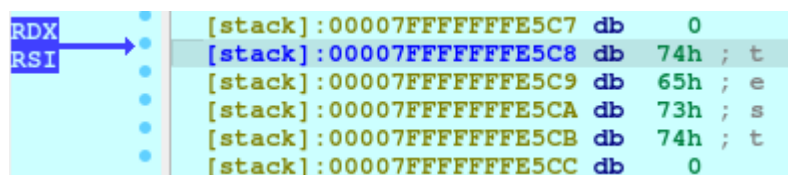
```



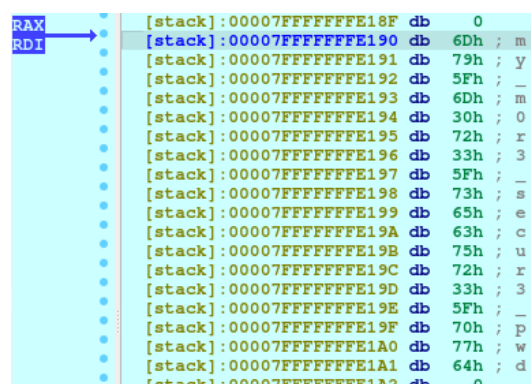
Après avoir lancé le programme avec le mot **"test"** en paramètre et atteint le breakpoint sur **strcmp**, nous pouvons examiner les valeurs stockées dans **RDI** et **RSI** (à droite de l'application, dans la fenêtre **"General Registers"**) pour déterminer le mot de passe attendu.



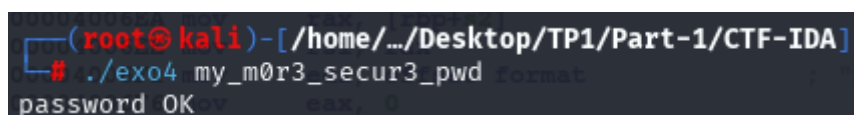
En affichant les registres, nous constatons que **RSI** pointe vers la chaîne **"test"**, ce qui confirme que c'est bien notre saisie.



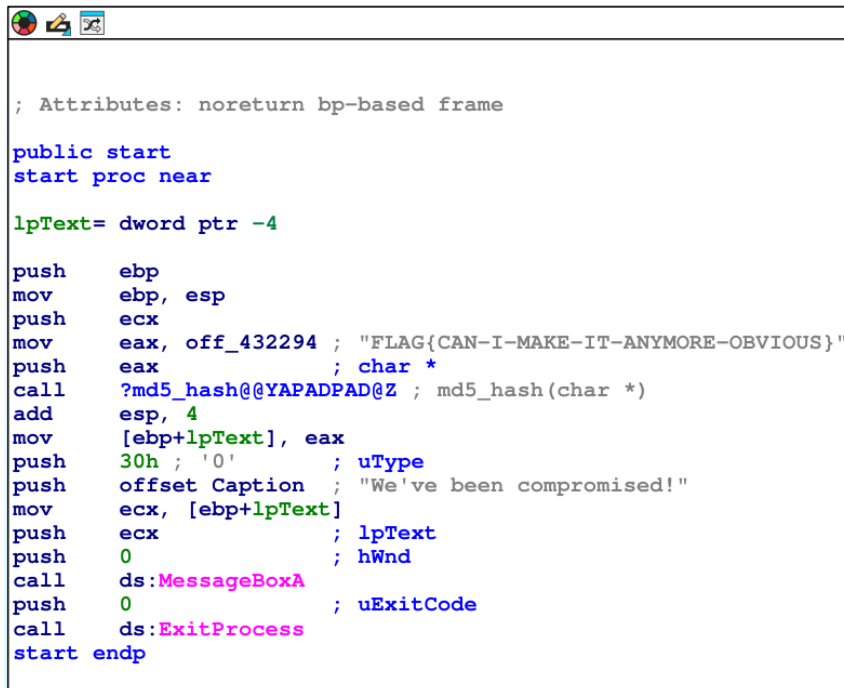
RDI pointe vers une chaîne **"my_m0r3_secur3_pwd"**, ce qui correspond au mot de passe attendu.



On lance le programme avec ce mot de passe et cela confirme bien qu'on a trouvé le bon mot de passe.



Exécutable 1



```
; Attributes: noreturn bp-based frame

public start
start proc near

lpText= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     eax, off_432294 ; "FLAG{CAN-I-MAKE-IT-ANYMORE-OBVIOUS}"
push    eax                ; char *
call    ?md5_hash@@YAPADPAD@Z ; md5_hash(char *)
add     esp, 4
mov     [ebp+lpText], eax
push    30h ; '0'          ; uType
push    offset Caption ; "We've been compromised!"
mov     ecx, [ebp+lpText]
push    ecx                ; lpText
push    0                  ; hWnd
call    ds:MessageBoxA
push    0                  ; uExitCode
call    ds:ExitProcess
start endp
```

On ouvre **Executable1** avec IDA et on voit directement le flag dans le code, sans avoir d'autre manipulation à faire.

Sous Linux, on devait d'abord entrer un mot de passe avant que le flag ne s'affiche. Ici, il est en clair dans le binaire et utilisé sans vérification.

Cela démontre qu'il est possible d'extraire des informations directement depuis un exécutable, sans le lancer, ce qui peut être particulièrement utile pour analyser un programme inconnu en toute sécurité, même s'il s'agit d'un **.exe Windows exécuté sous Linux**. Si le programme sous Linux cachait le flag derrière une vérification, ici, il suffit juste de regarder le code pour l'obtenir.

Exécutable 2

On ouvre **Executable2** avec IDA et cette fois, on ne voit pas directement le flag sous forme de chaîne de caractères comme dans **Executable1**.

À la place, le programme charge chaque caractère un par un en mémoire avec plusieurs instructions **mov**. Plutôt que d'être stocké en une seule fois, le flag est reconstruit dans le programme.

```
push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_28], 46h ; 'F'
mov     [ebp+var_27], 4Ch ; 'L'
mov     [ebp+var_26], 41h ; 'A'
mov     [ebp+var_25], 47h ; 'G'
mov     [ebp+var_24], 7Bh ; '{'
mov     [ebp+var_23], 53h ; 'S'
mov     [ebp+var_22], 54h ; 'T'
mov     [ebp+var_21], 41h ; 'A'
mov     [ebp+var_20], 43h ; 'C'
mov     [ebp+var_1F], 4Bh ; 'K'
mov     [ebp+var_1E], 2Dh ; '-'
mov     [ebp+var_1D], 53h ; 'S'
mov     [ebp+var_1C], 54h ; 'T'
mov     [ebp+var_1B], 52h ; 'R'
mov     [ebp+var_1A], 49h ; 'I'
mov     [ebp+var_19], 4Eh ; 'N'
mov     [ebp+var_18], 47h ; 'G'
mov     [ebp+var_17], 53h ; 'S'
mov     [ebp+var_16], 2Dh ; '-'
mov     [ebp+var_15], 41h ; 'A'
mov     [ebp+var_14], 52h ; 'R'
mov     [ebp+var_13], 45h ; 'E'
mov     [ebp+var_12], 2Dh ; '-'
mov     [ebp+var_11], 42h ; 'B'
mov     [ebp+var_10], 45h ; 'E'
mov     [ebp+var_F], 53h ; 'S'
mov     [ebp+var_E], 54h ; 'T'
mov     [ebp+var_D], 2Dh ; '-'
mov     [ebp+var_C], 53h ; 'S'
mov     [ebp+var_B], 54h ; 'T'
mov     [ebp+var_A], 52h ; 'R'
mov     [ebp+var_9], 49h ; 'I'
mov     [ebp+var_8], 4Eh ; 'N'
mov     [ebp+var_7], 47h ; 'G'
mov     [ebp+var_6], 53h ; 'S'
mov     [ebp+var_5], 7Dh ; '}'
lea     eax, [ebp+var_28]
push    eax ; char *
call    7md5_hash@@YAPADPAD@Z ; md5_hash(char *)
add     esp, 4
mov     [ebp+var_1], 0
```

Cela ne complique pas vraiment l'analyse, car il suffit de lire les valeurs chargées pour retrouver la chaîne complète. On récupère donc le flag sans exécuter l'exécutable, juste en observant le désassemblage.

Spark.exe

SYMBOL TREE

On ouvre **Spark.exe** avec Ghidra et on regarde les **DLL utilisées** ainsi que les **fonctions référencées**.



Dans les imports, on retrouve des **DLL système importantes** comme **ADVAPI32.DLL**, **KERNEL32.DLL**, **SHELL32.DLL**, **URLMON.DLL**, et **WININET.DLL**. Cela nous donne des indices sur les capacités du programme.

- **ADVAPI32.DLL** → Accès au **registre Windows**, potentiellement pour modifier des paramètres système ou établir une persistance.
- **KERNEL32.DLL** → Gestion **mémoire, fichiers et processus**, ce qui peut lui permettre de masquer sa présence ou d'exécuter du code arbitraire.
- **SHELL32.DLL** → Interaction avec **l'explorateur Windows**, possiblement pour manipuler des fichiers ou s'intégrer discrètement au système.
- **URLMON.DLL / WININET.DLL** → **Connexion réseau**, ce qui suggère un téléchargement de fichiers, une communication avec un serveur distant ou même une exfiltration de données.

Ces imports montrent que le malware a des capacités de persistance, d'exécution de commandes et de communication réseau, ce qui peut être utilisé pour du vol d'informations, du téléchargement de charges utiles ou du contrôle à distance.

On regarde les **méthodes visibles dans Ghidra** pour comprendre les capacités du programme.

- **GetPdbDll** → Suggère que le programme **charge dynamiquement des DLLs**. Cela peut être utilisé pour **exécuter du code externe, masquer ses dépendances** jusqu'à l'exécution ou charger des modules malveillants en fonction du contexte.
- **failwithmessage** → Indique que le malware **affiche des messages d'erreur**. Cela peut servir à **tromper l'utilisateur**, mais aussi être une **mécanique anti-debug**, en affichant un faux message pour cacher son vrai comportement.
- **ExFilterRethrow** → Présence d'un **mécanisme de gestion des exceptions**, qui peut être utilisé pour **dissimuler des erreurs** ou **perturber l'analyse**. Certains malwares l'utilisent pour éviter les crashes sous certaines conditions et empêcher leur détection.

- **regex_error** → Suggère l'usage d'**expressions régulières**, potentiellement pour **analyser du texte, extraire des données sensibles**, ou même rechercher des signatures spécifiques sur le système (comme des fichiers, des processus ou des logs ciblés).

RECHERCHE DE CHAÎNE

Les programmes qui pourraient interagir avec notre malware

String View
"\\installed\\windefender.exe"
"adobe\\flash.exe"
"AKW.exe"
"alg.exe"
"chrome.exe"
"csrss.exe"
"desktop.exe"
"devenv.exe"
"dllhost.exe"
"dwm.exe"
"explorer.exe"
"firefox.exe"
"iexplore.exe"
"java.exe"
"jucheck.exe"
"jused.exe"
"lsass.exe"
"ntoskrnl.exe"
"pidgin.exe"
"QML.exe"
"services.exe"
"skype.exe"
"smss.exe"
"spoolsv.exe"
"steam.exe"
"svchost.exe"
"taskmgr.exe"
"thunderbird.exe"
"win-firewall.exe"
"windefender.exe"
"wininit.exe"
"winlogon.exe"
"wsntfy.exe"
u"windefender.exe"

La présence de **firefox.exe**, **chrome.exe** et **iexplore.exe** est intéressante car cela peut indiquer que le malware cible les **navigateurs web**. Il pourrait chercher à **intercepter des données**, comme des identifiants enregistrés, des cookies ou des sessions actives, voire injecter du code malveillant dans les pages visitées.

L'inclusion de **explorer.exe** peut aussi suggérer une interaction avec l'**explorateur Windows**, soit pour manipuler des fichiers, soit pour se cacher au sein d'un processus légitime et éviter d'être détecté.


En parallèle, la présence de **taskmgr.exe** et **windefender.exe** renforce l'idée que le malware veut **empêcher l'utilisateur de le fermer et désactiver les protections Windows**.

Les logiciels de messagerie comme **skype.exe**, **pidgin.exe** et **thunderbird.exe** peuvent être ciblés pour **extraire des informations sensibles** ou **surveiller des communications**.

La présence de **steam.exe** peut indiquer que le malware s'intéresse à la plateforme Steam, potentiellement pour **voler des identifiants, intercepter des transactions** ou **injecter du code** dans le processus.

Le fait que le fichier "**\\installed\\windefender.exe**" apparaisse peut suggérer que le malware **se déguise sous ce nom** pour paraître légitime et rester discret.














Des noms d'utilisateurs ou de machines

Defined	Location	Label	Code Unit	...	Stri...	L...	Is...
	004f17a0		ds "C:\\Users\\Benson\\Desktop\\ALLIN\\Source working\\Debug\\Spark.pdb" (... "C:\\... string 60 true				

On repère une chaîne contenant **C:\\Users\\Benson\\Desktop\\ALLIN\\Source working\\Debug\\Spark.pdb**.

On a donc un **nom d'utilisateur : Benson**, ce qui peut être une trace laissée par le développeur. Soit l'attaquant a mal nettoyé son programme, laissant une info qui pourrait aider à le remonter, soit c'est une fausse piste volontaire.

Recherche de dll

Defined	Location	Label	Code Unit
	004fdc02		ds "ADVAPI32.dll"
	004fdae4		ds "KERNEL32.dll"
	004f6495		ds "shell32.dll"
	004fdc3e		ds "SHELL32.dll"
	004fdcf4		ds "urlmon.dll"
	004fdcd2		ds "WININET.dll"
	004e5214	u_ADVAPI32.D...	unicode u"ADVAPI32.DLL"
	004e5084	u_kernel32.dll...	unicode u"kernel32.dll"
	004e08d8	u_mscoree.dll...	unicode u"mscoree.dll"
	004eaa5c	u_MSPDB110....	unicode u"MSPDB110.DLL"
	004eaa78	u_MSVCRL10....	unicode u"MSVCRL10D.dll"
	004e2790	u_user32.dll...	unicode u"user32.dll"
	004e7068	u_USER32.DL...	unicode u"USER32.DLL"

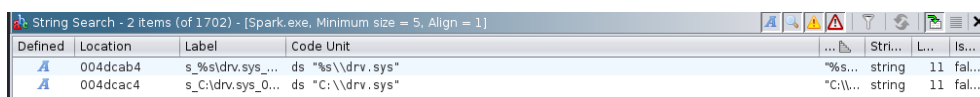
En comparant avec les **imports**, on retrouve bien **ADVAPI32.DLL**, **KERNEL32.DLL**, **SHELL32.DLL**, **URLMON.DLL** et **WININET.DLL**, ce qui confirme que le malware interagit avec le système et le réseau.



Cependant, **USER32.DLL** n'était pas listée avant, ce qui suggère qu'il peut manipuler l'interface graphique, peut-être pour afficher des fenêtres trompeuses ou intercepter des interactions. **MSCOREE.DLL** pourrait indiquer une dépendance au framework .NET, ce qui peut compliquer l'analyse.

Les DLL **MSPDB110.DLL** et **MSVCR110.DLL** confirment un build sous **Visual Studio 2012**, ce qui donne un indice sur son développement. Leur présence pourrait aussi être un résidu du débogage, signe d'un nettoyage bâclé.

Certaines de ces DLL ne figurant pas dans les imports initiaux, le malware pourrait les **charger dynamiquement** pour éviter d'être repéré immédiatement.




Fichier sys



Defined	Location	Label	Code Unit	...	Stri...	L...	Is...
	004dcab4	s_%s\drv.sys_...	ds \"%s\\drv.sys\"	%s...	string	11	fal...
	004dcac4	s_C:\drv.sys_0...	ds \"C:\\drv.sys\"	C:\\...	string	11	fal...

La présence de **C:\drv.sys** suggère que le malware installe un **driver malveillant**. Si ce driver est chargé, il pourrait **fonctionner en mode noyau**, donnant au malware un accès privilégié pour **cacher des processus**, **intercepter des appels système** ou **contourner les protections antivirus**.

Clés de registre

Defined	Location	Label	Code Unit	...	Stri...	L...	Is...
	004dc420	s_SOFTWARE\...	ds "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\"	"SO...	string	43	true
	004dc488	s_Software\Mi...	ds "Software\\Microsoft\\Windows\\CurrentVersion\\Run"	"Sof...	string	46	true
	004eaad8	u_SOFTWARE\...	unicode u"SOFTWARE\\Microsoft\\VisualStudio\\11.0\\Setup\\VS"	u"S...	unic...	92	true

On repère des références à des clés de registre Windows. Ces clés suggèrent que le malware pourrait modifier des paramètres système, s'ajouter au démarrage de Windows pour assurer

sa persistance, ou adapter son comportement en fonction de la version du système d'exploitation.

Requête php

Sachant que le malware récupère sûrement des informations, on cherche dans les chaînes des requêtes vers des serveurs (php ici).

Defined	Location	Label	Code Unit	...	Stri...	Len...	Is W...
...	004de924	s_wordpress/...	ds "/wordpress/post.php"	...	"/wo...	string	20 true

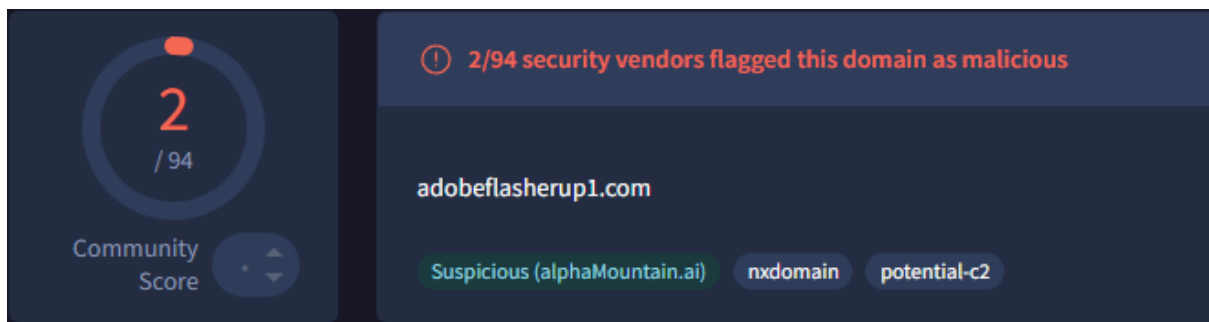
La présence de **"/wordpress/post.php"** suggère que le malware envoie bien des données via une **requête POST** à un serveur distant.

OSINT

On commence par chercher des url dans les chaînes.

Defined	Location	Label	Code Unit	...	Stri...	L...	Is...
...	004de93c	s_adobeflash...	ds "adobeflasherup1.com"	...	"ad...	string	20 true

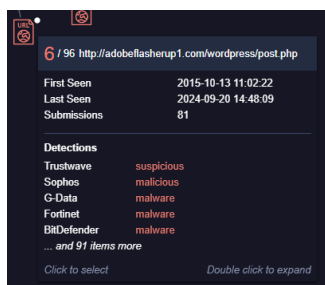
On remarque ainsi la présence de l'url "adobeflasherup1.com". On le passe sur virustotal pour avoir plus d'informations.



On voit que ce lien est malveillant. Pour en savoir plus, on récupère l'analyse hybride disponible dans les commentaires.

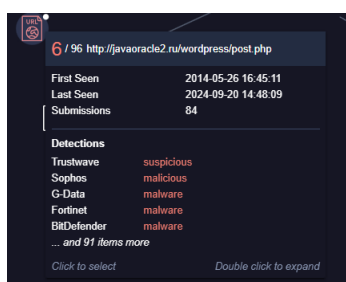


On remarque la présence de **windefender.exe**, ce qui confirme que le malware s'installe sous ce nom pour se camoufler. Ce fichier est déjà connu et relié à **deux domaines malveillants, javaoracle2.ru et adobeflasherup1.com**, ce dernier ayant été retrouvé dans les chaînes de Ghidra.



On retrouve également l'URL **adobeflasherup1.com/wordpress/post.php**, ce qui valide que le malware **exfiltre des données** en effectuant des requêtes **POST** vers ce serveur distant.

Le site **javaoracle2.ru** utilise la même URL **POST**, ce qui suggère que le malware **envoie les données vers deux serveurs distincts**. Cette redondance permet aux attaquants de **maintenir la collecte d'informations**, même si l'un des domaines devient inaccessible. Initialement, nous n'avons pas remarqué ce site dans ghidra lors de l'analyse mais après vérification, nous le retrouvons bien.



Defined	Location	Label	Code Unit	Str...	Len...	Is W...
004de954	s_javaoracle2...	ds "javaoracle2.ru"	"jav...	string	15	true

Enfin, les localisations des résolutions DNS pointent principalement vers la **Russie et l'Ukraine**, ce qui peut donner un indice sur l'origine des attaquants ou des infrastructures utilisées.



Analyse de rk.sys

Après ouverture et analyse de rk.sys, on récupère les adresses de départ et de fin de ce fichier qui sont **0x00010000** pour le début et **0xFFDFFFFF** pour la fin.

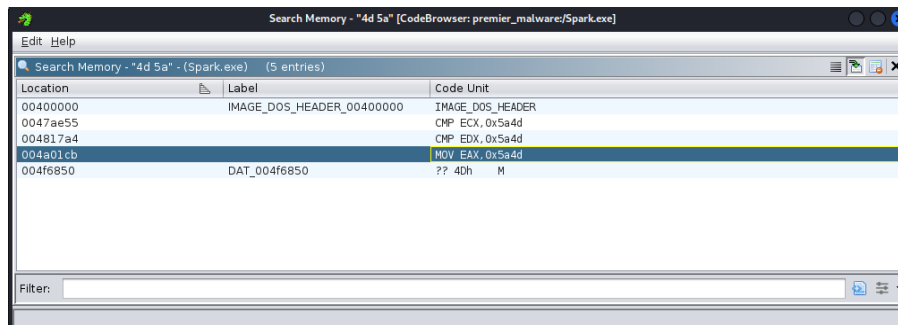
Start: 00010000 End: ffdfffff

En regardant l'entête du fichier, on remarque qu'il commence par **4D 5A**. Après recherche, on voit que cela correspond à une signature **MZ**, indiquant qu'il s'agit d'un **fichier exécutable Windows**.

4D 5A	MZ	0	exe dll mui sys scr cpl ocx ax iec ime rs tsp fon efi	DOS MZ executable and its descendants (including NE and PE)
-------	----	---	----------------------------------------------------------------------------------------------	-------------------------------------------------------------

Les fichiers MZ incluent les **.exe**, **.dll**, **.sys**, entre autres. Ici, comme l'extension est **.sys**, cela confirme que **rk.sys est un driver Windows**, ce qui peut lui permettre de s'exécuter en **mode noyau** et d'avoir des **privilèges élevés sur le système**.

Après recherche des occurrences 4d 5a sur le programme spark.exe, on en trouve 5.



La première occurrence correspond à **spark.exe** directement. En effet, nous avons vu que **4D 5A** correspond à l'entête **MZ**, utilisé pour identifier les exécutables Windows, et pas seulement les fichiers **.sys**. Cette occurrence indique simplement dans le header que **spark.exe** est bien un fichier exécutable.

La deuxième occurrence se réalise dans **_check_managed_app** et vérifie si un fichier commence bien par "MZ". Cela peut être un contrôle pour savoir si un fichier est un exécutable ou un driver avant de le charger ou l'exécuter.

La troisième occurrence se trouve dans **_ValidateImageBase** et effectue une vérification similaire à la précédente. Ici, le programme semble s'assurer qu'un fichier possède bien l'entête "MZ", sûrement pour valider un exécutable avant de l'utiliser.

La quatrième occurrence est un **MOV EAX, 0x5A4D**, ce qui signifie que le programme place directement la valeur "MZ" dans un registre. Cela peut être utilisé plus tard pour une comparaison ou une manipulation en mémoire.

La dernière occurrence est une donnée brute stockée en mémoire, contenant la valeur "M". Cela pourrait être un résidu de manipulation de fichiers ou une référence partielle à un exécutable.

La recherche montre que **spark.exe** manipule activement des fichiers exécutables au format MZ, notamment en vérifiant leurs entêtes et en stockant cette signature en mémoire. Ces vérifications suggèrent qu'il cherche à interagir avec des exécutables ou des drivers, probablement pour les charger, valider ou modifier. Plus tôt, nous avons repéré **drv.sys**, un driver potentiellement malveillant. Il est donc probable que le programme effectue ces manipulations pour gérer ou exploiter ce fichier en particulier, ce qui renforce l'idée d'une infection via un driver système pour obtenir des privilèges élevés.

Conclusion

L'analyse de **spark.exe** montre un malware avec des capacités variées : modification du registre, interaction avec des processus Windows et communication avec des serveurs distants. L'utilisation de **drv.sys** indique qu'il exploite un driver système, probablement pour obtenir des privilèges élevés. Il envoie des données vers plusieurs serveurs et se camoufle sous des noms légitimes comme **windefender.exe**, confirmant son intention de rester actif sur la machine infectée.