



• [Baleful] •

. Introducción .[Solve](#) [Hint](#) [Review](#)

This program seems to be rather delightfully twisted! Can you get it to accept a password? We need it to get access to some Daedalus Corp files.

. Descripción de la tarea .

Arriba podéis ver la descripción de la tarea. Creo recordar que la pista decía algo sobre que el programa estaba empaquetado y el uso del propio *packer* para obtener el código original.

Parece que será una tarea fácil

. Primer Contacto .

Comenzamos usando un editor hexadecimal, con el cual vamos a inspeccionar el contenido del archivo binario. Ejecuta “*radare2*” (desde ahora *r2*) y escribe “*px 400*”, este comando nos mostrara un volcado hexadecimal del archivo.

```
root@ymkalix86 ~/picoctf# r2 baleful
Warning: Cannot initialize section headers
-- Hold on, this should never happen!
[0x00c02038]> px 400
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00c02038 e80f 0200 0060 8b74 2424 8b7c 242c 83cd .....t$$.$,..
0x00c02048 ffeb 0f90 9090 9090 8a06 4688 0747 01db .....F..G..
0x00c02058 7507 8b1e 83ee fc11 db8a 0772 ebb8 0100 u.....r...
0x00c02068 0000 01db 7507 8b1e 83ee fc11 db11 c001 ...u.....
0x00c02078 db73 ef75 098b 1e83 eefc 11db 73e4 31c9 .s.u.....s.l
0x00c02088 83e8 0372 0dc1 e008 8a06 4683 f0ff 7476 ...r.....F...tv
0x00c02098 89c5 01db 7507 8b1e 83ee fc11 db11 c901 ...u.....
0x00c020a8 db75 078b 1e83 eefc 11db 11c9 7520 4101 .u.....u A.
0x00c020b8 db75 078b 1e83 eefc 11db 11c9 01db 73ef .u.....s...S.
0x00c020c8 7509 8b1e 83ee fc11 db73 e483 c102 81fd u.....s.....
0x00c020d8 00f3 ffff 83d1 018d 142f 83fd fc8a 040f ...../.....
0x00c020e8 760e 8a02 4288 0747 4975 f7e9 5eff ffff v...B..GIu..^...
0x00c020f8 8b02 83c2 0489 0783 c704 83e9 0477 f101 .....w...
0x00c02108 cfe9 48ff ffff 8b54 2424 0354 2428 39d6 ..H....T$$$.T$(9.
0x00c02118 7401 482b 7c24 2c8b 5424 3089 3a89 4424 t.H+|$.T$0.:.D$
0x00c02128 1c61 c361 c30a 0024 496e 666f 3a20 5468 .a.a...$Info: Th
0x00c02138 6973 2066 696c 6520 6973 2070 6163 6b65 is file is packe
0x00c02148 6420 7769 7468 2074 6865 2055 5058 2065 d with the UPX e
0x00c02158 7865 6375 7461 626c 6520 7061 636b 6572 xecutable packer
0x00c02168 2068 7474 703a 2f2f 7570 782e 7366 2e6e http://upx.sf.n
0x00c02178 6574 2024 0a00 2449 643a 2055 5058 2033 et $.Id: UPX 3
0x00c02188 2e39 3120 436f 7079 7269 6768 7420 2843 .91 Copyright (C
0x00c02198 2920 3139 3936 2d32 3031 3320 7468 6520 ) 1996-2013 the
0x00c021a8 5550 5820 5465 616d 2e20 416c 6c20 5269 UPX Team. All Ri
0x00c021b8 6768 7473 2052 6573 6572 7665 642e 2024 ghts Reserved. $
[0x00c02038]>
```

Podemos observar cómo el programa ha sido empaquetado usando *UPX*. Este compresor es bastante conocido y el propio *UPX* tiene una opción “-d” que puede ser usada para obtener el archivo original sin compresión. Por lo tanto usando el comando “-d” obtener el archivo original es una tarea trivial.

. Análisis .

Vayamos a inspeccionar el contenido mediante el desensamblado de código.

Carga el archivo desempaquetado en *r2* y usa el comando “aa” para ejecutar el auto análisis, “s entry0” para indicar que queremos que el bloque apunte a la entrada del programa y “pd 14@entry0” para obtener un desensamblado de las 14 primeras instrucciones.

```
root@vmkalix86:~/picocf/dbgsrv# r2 baleful
-- Thank you for using radare2. Have a nice night!
[0x08048540]> aa
[0x08048540]> s entry0
[0x08048540]> pd 14@entry0
; [12] va=0x08048540 pa=0x00000540 sz=6236 vsz=6236 rwx=-r-x .text
/ (fcn) entry0
|
| section..text:
| 0x08048540 31ed      xor ebp  ebp
| 0x08048542 5e        pop esi
| 0x08048543 89e1      mov ecx  esp
| 0x08048545 83e4f0    and esp  0xffffffff
| 0x08048548 50        push eax
| 0x08048549 54        push esp
| 0x0804854a 52        push edx
| 0x0804854b 68609d0408 push 0x8049d60
| 0x08048550 68f09c0408 push fcn.08049cf0
| 0x08048555 51        push ecx
| 0x08048556 56        push esi
| 0x08048557 68829c0408 push main ; "U..WS....." @ 0x8049c82
| 0x0804855c e88fffff  call sym.imp.__libc_start_main
| 0x08048561 f4        hlt
[0x08048540]>
```

Podríamos simplemente cargar el archivo dentro de *r2* y directamente hacer un “px 14”, pero en este ejemplo hemos empleado 3 comandos. Dejame que te explique que hace cada uno

- “aa”: Auto análisis. Intenta localizar funciones, strings, imports, exports ...
- “s”: Comando Seek. Indica a *r2* la dirección del bloque de trabajo.
- “pd”: Perform a disassemble. Este comando realiza un desensamblado. Si usamos este comando sin parámetros, realizará un desensamblado del bloque actual, el bloque actual siempre lo puedes ver indicado en el número que aparece en el propio *prompt* de *r2* “[0x08048540]>”. Si añadimos un parámetro al comando, estaremos indicando que queremos desensamblar ese número de instrucciones “pd 10”. Si queremos mostrar el desensamblado de una dirección sin tener que cambiar el bloque actual podemos hacerlo usando “@” y a continuación la dirección a mostrar “pd @0x401000”. Por último recuerda que puedes combinar ambos parámetros para indicar que muestre *N* instrucciones a partir de un *offset* “pd 20@401000”.

Introducimos el comando “pd 14@entry0” el cual nos muestra el siguiente código desensamblado:

```

/ (fcn) entry0 34
|      ;-- section..text:
|      0x08048540  31ed      xor ebp, ebp
|      0x08048542  5e        pop esi
|      0x08048543  89e1      mov ecx, esp
|      0x08048545  83e4f0    and esp, 0xffffffff
|      0x08048548  50        push eax
|      0x08048549  54        push esp
|      0x0804854a  52        push edx
|      0x0804854b  68609d0408 push 0x8049d60
|      0x08048550  68f09c0408 push fcn.08049cf0
|      0x08048555  51        push ecx
|      0x08048556  56        push esi
|      0x08048557  68829c0408 push main      ; "U..WS....." @ 0x8049c82
|      0x0804855c  e88fffff  call sym.imp.__libc_start_main
|      sym.imp.__libc_start_main(unk, unk, unk, unk, unk, unk, unk, unk)
\      0x08048561  f4        hlt

```

Podemos ver cómo se realiza una llamada a “sym.imp.__libc_start_main”. Si inspeccionamos los valores que se le pasan a esta llamada, podremos localizar donde se encuentra la función “main” del programa. Recuerda que el primer parámetro pasado a esta llamada suele ser la dirección de main(). Por lo tanto la función main deberíamos localizarla en 0x8049C82. Vamos a realizar el desensamblado de esta dirección “pd 29@0x8049c82” o “pd 29@main”:

```

0x08049c82  55        push ebp
0x08049c83  89e5      mov ebp, esp
0x08049c85  57        push edi
0x08049c86  53        push ebx
0x08049c87  83e4f0    and esp, 0xffffffff
0x08049c8a  81ec90000000 sub esp, 0x90
0x08049c90  65a114000000 mov eax, dword gs:[0x14]
0x08049c96  8984248c000. mov dword [esp + 0x8c], eax
0x08049c9d  31c0      xor eax, eax
0x08049c9f  8d442410  lea eax, dword [esp + 0x10]
0x08049ca3  89c3      mov ebx, eax
0x08049ca5  b800000000 mov eax, 0
0x08049caa  balf000000 mov edx, 0x1f
0x08049caf  89df      mov edi, ebx
0x08049cb1  89d1      mov ecx, edx
0x08049cb3  f3ab      rep stosd dword es:[edi],
0x08049cb5  8d442410  lea eax, dword [esp + 0x10]
0x08049cb9  890424    mov dword [esp], eax
0x08049cbc  e8caecffff call 0x804898b
0x0804898b(unk, unk, unk) ; entry0
0x08049cc1  b800000000 mov eax, 0
0x08049cc6  8b94248c000. mov edx, dword [esp + 0x8c]
0x08049ccd  65331514000. xor edx, dword gs:[0x14]
,=< 0x08049cd4  7405      je 0x8049cdb
| 0x08049cd6  e8e5e7ffff call sym.imp.__stack_chk_fail
| 0x080484c0() ; sym.imp.__stack_chk_fail
`-> 0x08049cdb  8d65f8    lea esp, dword [ebp - 8]
0x08049cde  5b        pop ebx
0x08049cdf  5f        pop edi
0x08049ce0  5d        pop ebp
0x08049ce1  c3        ret

```

Podemos ver cómo en “0x08049cbc call 0x804898b”, realiza una llamada a otra función. Veamos qué contiene : “pd 38@0x804898b”:

```

|           ; CALL XREF from 0x08049cbc (unk)
| / (fcn) fcn.0804898b 4855
|           0x0804898b 55          push ebp
|           0x0804898c 89e5          mov  ebp, esp
|           0x0804898e 81ecc8000000 sub  esp, 0xc8
|           0x08048994 c745cc00100. mov  dword [ebp - 0x34], 0x1000 ;
| [0x1000:4]=0xc080b60f
|           0x0804899b 837d0800    cmp  dword [ebp + 8], 0          ;
| [0x8:4]=0
|           ,=< 0x0804899f 742a          je  0x80489cb
|           | 0x080489a1 c745d000000. mov  dword [ebp - 0x30], 0
|           ,==< 0x080489a8 eb19          jmp  0x80489c3          ; (fcn.0804898b)
|           ; JMP XREF from 0x080489c7 (fcn.0804898b)
|           .----> 0x080489aa 8b45d0    mov  eax, dword [ebp - 0x30]
|           ||| 0x080489ad c1e002    shl  eax, 2
|           ||| 0x080489b0 034508    add  eax, dword [ebp + 8]
|           ||| 0x080489b3 8b10      mov  edx, dword [eax]
|           ||| 0x080489b5 8b45d0    mov  eax, dword [ebp - 0x30]
|           ||| 0x080489b8 8994854cfff. mov  dword [ebp + eax*4 - 0xb4], edx
|           ||| 0x080489bf 8345d001  add  dword [ebp - 0x30], 1
|           || ; JMP XREF from 0x080489a8 (fcn.0804898b)
|           |`--> 0x080489c3 837dd01e    cmp  dword [ebp - 0x30], 0x1e
|           `====< 0x080489c7 7ee1          jle  0x80489aa
|           ,====< 0x080489c9 eb21          jmp  0x80489ec          ; (fcn.0804898b)
|           | | ; JMP XREF from 0x0804899f (fcn.0804898b)
|           | |`-> 0x080489cb c745d400000. mov  dword [ebp - 0x2c], 0
|           ,====< 0x080489d2 eb12          jmp  0x80489e6          ; (fcn.0804898b)
|           ; JMP XREF from 0x080489ea (fcn.0804898b)
|           .-----> 0x080489d4 8b45d4    mov  eax, dword [ebp - 0x2c]
|           ||| 0x080489d7 c784854cfff. mov  dword [ebp + eax*4 - 0xb4], 0
|           ||| 0x080489e2 8345d401  add  dword [ebp - 0x2c], 1
|           || ; JMP XREF from 0x080489d2 (fcn.0804898b)
|           |`-----> 0x080489e6 837dd41e    cmp  dword [ebp - 0x2c], 0x1e
|           `====< 0x080489ea 7ee8          jle  0x80489d4
|           | ; JMP XREF from 0x080489c9 (fcn.0804898b)
|           |`-----> 0x080489ec c745c800f00. mov  dword [ebp - 0x38], 0xf000
|           | 0x080489f3 c745d800000. mov  dword [ebp - 0x28], 0
|           | 0x080489fa c745ec00000. mov  dword [ebp - 0x14], 0
|           | 0x08048a01 c745f000000. mov  dword [ebp - 0x10], 0
|           | 0x08048a08 c745f400000. mov  dword [ebp - 0xc], 0
|           | 0x08048a0f c745e800000. mov  dword [ebp - 0x18], 0
|           | 0x08048a16 8b45e8    mov  eax, dword [ebp - 0x18]
|           | 0x08048a19 8945e4    mov  dword [ebp - 0x1c], eax
|           | 0x08048a1c 8b45e4    mov  eax, dword [ebp - 0x1c]
|           | 0x08048a1f 8945e0    mov  dword [ebp - 0x20], eax
|           | 0x08048a22 8b45e0    mov  eax, dword [ebp - 0x20]
|           | 0x08048a25 8945dc    mov  dword [ebp - 0x24], eax
|           ,====< 0x08048a28 e93a120000 jmp  0x8049c67          ; (fcn.0804898b)
|           | ; JMP XREF from 0x08049c74 (fcn.0804898b)
|           |
|

```

Aquí tenemos una función bastante grande y sin lugar a dudas diría que es donde empieza el verdadero reto.

Realizando un análisis rápido podemos ver cómo las primeras instrucciones se encargan de inicializar varias variables dentro de la pila. Según el código tras reservar espacio en pila bastantes variables, estas pueden ser inicializadas a 0, o usando un puntero el cual debe apuntar a un *array* de *dwords* con el valor de cada una de estas variables. Este puntero se pasa en el argumento 0 de la función.

Básicamente el código está realizando un bucle de 0 a 0x1e, para inicializar las variables, cuando ha terminado la inicialización de la variables salta a 0x80489ec, donde continua inicializando valores y finalmente continua la ejecución en 0x8049c67.

Inspeccionemos esta dirección “pd 8@0x8049c67”:

```
| ; JMP XREF from 0x08048a28 (fcn.0804898b)
| 0x08049c67 8b45cc mov eax, dword [ebp - 0x34]
| 0x08049c6a 05c0c00408 add eax, 0x804c0c0
| 0x08049c6f 0fb600 movzx eax, byte [eax]
| 0x08049c72 3c1d cmp al, 0x1d
|=< 0x08049c74 0f85b3edffff jne 0x8048a2d
| 0x08049c7a 8b854cffffff mov eax, dword [ebp - 0xb4]
| 0x08049c80 c9 leave
| 0x08049c81 c3 ret
```

El trozo de código de arriba está obteniendo el byte que se encuentra en la dirección 0x0804c0c0 + [ebp-0x34] y lo compara con el valor 0x1d. Si el valor no coincide continua saltando a otra dirección de lo contrario salimos de esta rutina.

Veamos ahora que ocurre cuando la función continua, para ello examinamos la dirección de destino del jne, “pd 8@0x8048a2d”:

```
| ; JMP XREF from 0x08049c74 (fcn.0804898b)
| 0x08048a2d 8b45cc mov eax, dword [ebp - 0x34]
| 0x08048a30 05c0c00408 add eax, 0x804c0c0
| 0x08048a35 0fb600 movzx eax, byte [eax]
| 0x08048a38 0fbec0 movsx eax, al
| 0x08048a3b 83f820 cmp eax, 0x20
|=< 0x08048a3e 0f871e120000 ja 0x8049c62
| 0x08048a44 8b0485d49d0. mov eax, dword [eax*4 + 0x8049dd4]
| 0x08048a4b ffe0 jmp eax
```

El código que estamos viendo arriba, tiene la forma de un *switch/case* de C, tiene una tabla de saltos “[eax*4 + 0x8049dd4]” y la comprobación “cmp eax, 0x20; ja 0x8049c62” para el “case default”.

Si recopilamos todo lo encontrado hasta el momento tenemos:

- Una inicialización de 0x1e variables que pueden iniciarse a 0 o con valores determinados de un puntero a *dwords*.
- Una comparación con el valor 0x1d, para determinar la salida de esta función.
- Un gran switch que selecciona que hacer en base a un byte extraído de una posición de terminada de memoria.

Si miramos el destino de cada salto de la tabla 0x8049dd4, podemos observar cómo cada caso retorna siempre a la misma dirección 0x8049c67, donde se extrae de nuevo un byte y en base a este se determina que hacer.

Sin lugar a dudas podríamos decir que cumple con el comportamiento de una máquina virtual.

. Análisis de una máquina virtual .

. Qué se denomina "Máquina Virtual" .

“En informática una *máquina virtual* es un software que emula a una computadora y puede ejecutar programas como si fuese una computadora real.”

En nuestro caso es un software que emula un determinado juego de instrucciones.

Todo procesador necesita unos componentes básicos:

- Juego de instrucciones
- Registros
- Memoria
- Dispositivos de E/S

Durante nuestro análisis deberemos encontrar todos estos componentes.

. Análisis General .

Con la información recopilada hasta ahora podemos inferir que la rutina en la dirección “0x0804898B” es realmente lo que se denomina interprete de la máquina virtual.

Volvamos a revisar el código que vimos en nuestro análisis inicial en el cual se extraía un byte y se comparaba con 0x1d. “pd 8@0x8049c67”:

```
| ; JMP XREF from 0x08048a28 (fcn.0804898b)
| 0x08049c67 8b45cc mov eax, dword [ebp - 0x34]
| 0x08049c6a 05c0c00408 add eax, 0x804c0c0
| 0x08049c6f 0fb600 movzx eax, byte [eax]
| 0x08049c72 3c1d cmp al, 0x1d
|=< 0x08049c74 0f85b3edffff jne 0x8048a2d
| 0x08049c7a 8b854cffffff mov eax, dword [ebp - 0xb4]
| 0x08049c80 c9 leave
| 0x08049c81 c3 ret
```

Como vimos anteriormente simplemente obtiene un byte y lo compara con 0x1d y si no coincide salta a la dirección 0x8048a2d. Veamos de nuevo el código de esta dirección. “pd 8@0x8048a2d”:

```
| ; JMP XREF from 0x08049c74 (fcn.0804898b)
| 0x08048a2d 8b45cc mov eax, dword [ebp - 0x34]
| 0x08048a30 05c0c00408 add eax, 0x804c0c0
| 0x08048a35 0fb600 movzx eax, byte [eax]
| 0x08048a38 0fbec0 movsx eax, al
| 0x08048a3b 83f820 cmp eax, 0x20
|=< 0x08048a3e 0f871e120000 ja 0x8049c62
| 0x08048a44 8b0485d49d0. mov eax, dword [eax*4 + 0x8049dd4]
| 0x08048a4b ffe0 jmp eax
```

Realiza operaciones muy similares, pero ahora el byte que se extrajo es usado para determinar que caso de *switch* debe realizar.

Sin lugar a dudas el primer trozo de código está revisando si el byte extraído es el que corresponde para finalizar la ejecución de la VM (0x1d). Si no es así se ejecuta el *switch/case* que podemos ver cómo contiene 0x20 casos y además comprueba si sobrepasamos este número para ejecutar el “case default”, esto nos indica que la VM sabe manejar 0x20 instrucciones u *op-codes*.

Ahora que sabemos que estamos ante una máquina virtual, con el poco análisis realizado podemos determinar lo siguiente:

- El número total de instrucciones de la máquina virtual, 0x20.
- El valor de la instrucción que finaliza el código virtual, 0x1d.
- La tabla con todas las instrucciones/*op-codes*. 0x8049dd4
- Puntero a la instrucción actual. [ebp-var_34] + 0x804c0c0.
- La base de la VM, 0x804c0c0.

Veamos de nuevo el código de inicialización de la VM (0x804898b):

```

/ (fcn) fcn.0804898b 4855
|      0x0804898b 55      push ebp
|      0x0804898c 89e5      mov ebp, esp
|      0x0804898e 81ecc8000000 sub esp, 0xc8
|      0x08048994 c745cc00100. mov dword [ebp - 0x34], 0x1000 ;
|      0x0804899b 837d0800      cmp dword [ebp + 8], 0 ;
|      ,=< 0x0804899f 742a      je 0x80489cb
|      | 0x080489a1 c745d000000. mov dword [ebp - 0x30], 0
|      ,==< 0x080489a8 eb19      jmp 0x80489c3 ; (fcn.0804898b)
|      ; JMP XREF from 0x080489c7 (fcn.0804898b)
|      .----> 0x080489aa 8b45d0      mov eax, dword [ebp - 0x30]
|      ||| 0x080489ad c1e002      shl eax, 2
|      ||| 0x080489b0 034508      add eax, dword [ebp + 8]
|      ||| 0x080489b3 8b10      mov edx, dword [eax]
|      ||| 0x080489b5 8b45d0      mov eax, dword [ebp - 0x30]
|      ||| 0x080489b8 8994854cfff. mov dword [ebp + eax*4 - 0xb4], edx
|      ||| 0x080489bf 8345d001      add dword [ebp - 0x30], 1
|      || ; JMP XREF from 0x080489a8 (fcn.0804898b)
|      |`--> 0x080489c3 837dd01e      cmp dword [ebp - 0x30], 0x1e
|      `====< 0x080489c7 7ee1      jle 0x80489aa
|      ,====< 0x080489c9 eb21      jmp 0x80489ec ; (fcn.0804898b)
|      | | ; JMP XREF from 0x0804899f (fcn.0804898b)
|      | `-> 0x080489cb c745d400000. mov dword [ebp - 0x2c], 0
|      ,====< 0x080489d2 eb12      jmp 0x80489e6 ; (fcn.0804898b)
|      ; JMP XREF from 0x080489ea (fcn.0804898b)
|      .-----> 0x080489d4 8b45d4      mov eax, dword [ebp - 0x2c]
|      ||| 0x080489d7 c784854cfff. mov dword [ebp + eax*4 - 0xb4], 0
|      ||| 0x080489e2 8345d401      add dword [ebp - 0x2c], 1
|      || ; JMP XREF from 0x080489d2 (fcn.0804898b)
|      |`-----> 0x080489e6 837dd41e      cmp dword [ebp - 0x2c], 0x1e
|      `=====< 0x080489ea 7ee8      jle 0x80489d4
|      | ; JMP XREF from 0x080489c9 (fcn.0804898b)
|      `-----> 0x080489ec c745c800f00. mov dword [ebp - 0x38], 0xf000

```

Como vimos anteriormente la rutina reserva varias variables locales en la pila, 0x1e de estas variables serán usadas como registros de la máquina virtual. Y si

recuerdas el análisis inicial vimos que estas variables podían ser inicializadas de 2 formas distintas determinadas por el `arg0`

- Si se pasa un *null*, todos los registros son inicializados a 0.
- Si pasamos un valor, este debe ser un puntero a `0x1e dwords`. Los valores de este puntero son usados para inicializar las variables de la VM. Esto se hace para poder pasar valores desde el *host* a la VM, de esta forma la máquina virtual puede interactuar con su *host* o con el resultado de una llamada anterior a la propia máquina virtual.

Genial, ahora mismo sabemos el número total de registros que usa la máquina virtual y donde los guarda. Son un total de `0x1e` registros alojados en el *stack* de la propia función que interpreta el código virtual.

Dentro de los registros siempre existen al menos 3 registros “especiales”:

- Registro de Pila (*Stack Register*)
- Registro de banderas (*Eflags Register*)
- Puntero a Instrucción (*Instruction Pointer Register*)

De momento solo nos interesa saberlo. Mas adelante esta información será muy útil para analizar cada uno de los *op-codes* de la máquina virtual.

Actualmente sabemos cuál es el *Puntero a las instrucciones* gracias al análisis inicial en el que vimos cómo extraía los bytes en base a `[ebp-var_34] + 0x804c0c0`. Por lo tanto `[ebp-var_34]`, es nuestro registro *IP virtual* (a partir de ahora *vIP*).

El siguiente paso a seguir debería ser analizar la tabla de saltos del *switch/case* para poder determinar cómo manejan los parámetros cada uno de los *op-codes*. En una máquina virtual cada instrucción del conjunto de instrucciones es llamado “instrucción”, sin más, u “op-code”. Cada op-code operara de forma distinta y necesitará de distintos parámetros; algunos manejaran registros, otros manejaran *dwords*, otros bytes, e incluso algunos manejarán registros, *dwords* o bytes todo a la vez. Por lo tanto debe existir algo que indique que tipo de parámetros usa cada op-code.

Para ilustrar este concepto dejame mostrarte cómo se hace en *x86*:

<u>instrucción</u>	<u>Tipo de direccionamiento</u>
<code>mov eax , 1</code>	<code>reg imm</code>
<code>mov eax , ebx</code>	<code>reg reg</code>
<code>mov [eax], 1</code>	<code>[reg] imm</code>
<code>mov [eax], ebx</code>	<code>[reg] reg</code>

Fijate cómo el direccionamiento `mov [reg],[reg]` no existe en el x86. Sin embargo en una máquina virtual no hay nada que nos impida implementar este tipo de direccionamiento.

SkUaTeR

Puedes ver cómo el opcode “`mov`” gestiona sus distintos direccionamientos (esto es un ejemplo simple, el `mov` real tiene muchos más direccionamientos que los que he puesto). Dentro de una máquina virtual el concepto es el mismo. Si el op-code `mov` tiene que mover valores de un registro a otro, la forma de codificarlo debe ser distinta a si el `mov` realiza movimientos de un valor dentro de un registros. Lo que significa la necesidad de un byte adicional o usar los propios bits de la instrucción para manejar los distintos tipos de direccionamiento.

Cómo realizar esta codificación es algo que debe ser pensado en la fase de diseño de la máquina virtual y según las necesidades intentaremos optimizar la codificación lo máximo posible. La máquina virtual analizada siempre usa un byte extra para indicar el tipo de direccionamiento usado por el op-code, así como los parámetros que obtendrá del *Buffer de ejecución*.

Tras el análisis total de la máquina virtual nos daremos cuenta que los op-codes podrían haberse optimizado mucho más ya que está utilizando un byte entero para el direccionamiento y podría haberse codificado en el mismo byte de la instrucción. Los chicos que han diseñado este reto no han querido complicarnos la vida aún más.

SkUaTeR

. Análisis de los Op-codes y sus parámetros .

Ahora que tenemos unas nociones básicas de que es una máquina virtual y cómo funcionan algunos de sus componentes, es el momento de inspeccionar la tabla de saltos que nos va desvelar cómo funcionan los *op-codes*. Recuerda que la tabla la teníamos en la dirección 0x8049dd4, y esta tabla contendrá los punteros al código que maneja cada una de las instrucciones. Usa el siguiente comando para visualizar dicha tabla:

```
"pxw 0x24*4@0x8049dd4":
```

```
0x08049dd4 0x08048a4d 0x08048a56 0x08048a8f 0x08048bc4 M...V.....
0x08049de4 0x08048cf9 0x08048e2f 0x08048f91 0x080495f5 .... / .....
0x08049df4 0x08049649 0x080490c6 0x080491fb 0x0804959e I.....
0x08049e04 0x08049330 0x08049467 0x080496d1 0x0804969d 0...g.....
0x08049e14 0x080496ec 0x08049715 0x0804973e 0x08049767 .....>...g...
0x08049e24 0x08049790 0x080497b9 0x080497e2 0x080498f0 .....
0x08049e34 0x08049a02 0x08049a86 0x08049ab9 0x08049aec .....
0x08049e44 0x08049b43 0x08049c62 0x08049b92 0x08049bf8 C...b.....
0x08049e54 0x08049c2e 0x3b031b01 0x000000e0 0x0000001b .....f.....
```

Usemos el primer valor de la tabla y obtengamos el código para lo que sería el op-code 0, "pd 2@0x8048a4d":

```
0x08048a4d 8345cc01 add dword [ebp - 0x34], 1
,=< 0x08048a51 e911120000 jmp 0x8049c67
```

Este es de los fáciles, mas que nada porque se compone de 2 instrucciones, un "add" y el salto al código para obtener el próximo *op-code*. Como vimos antes [ebp-0x34] es el puntero de instrucciones virtual y según vemos el código lo único que hace es añadirle 1 y continuar. Por lo tanto este *op-code* se podría denominar "NOP" ya que no hace nada salvo incrementar el *vIP*.

Si te fijas este *op-code* no necesita ningún tipo de direccionamiento ya que afecta directamente al registro *vIP* ([ebp-34]), como se comenté anteriormente el registro *IP* es un registro especial, y en la arquitectura *x86* no puedes acceder directamente a él, parece ser que en esta máquina virtual han respetado esto pero podría existir perfectamente un acceso normal a este registro especial, recuerda que todo depende de cómo se haya diseñado la máquina virtual.

Continuamos inspeccionando el siguiente valor de la tabla "0x08048a56". Para ello hacemos un desensamblado de esta dirección, "pd 15@0x8048a56":

```
0x08048a56 8b45c8 mov eax, dword [ebp - 0x38]
0x08048a59 05c0c00408 add eax, 0x804c0c0
0x08048a5e 8b00 mov eax, dword [eax]
0x08048a60 8945ec mov dword [ebp - 0x14], eax
0x08048a63 837dec00 cmp dword [ebp - 0x14], 0
,=< 0x08048a67 750b jne 0x8048a74
| 0x08048a69 8b854cffffff mov eax, dword [ebp - 0xb4]
,==< 0x08048a6f e90c120000 jmp 0x8049c80
```

```

|`-> 0x08048a74      8b45c8      mov eax, dword [ebp - 0x38]
|      0x08048a77      83c004      add eax, 4
|      0x08048a7a      8945c8      mov dword [ebp - 0x38], eax
|      0x08048a7d      8b45ec      mov eax, dword [ebp - 0x14]
|      0x08048a80      8945cc      mov dword [ebp - 0x34], eax
|      0x08048a83      c745d80000. mov dword [ebp - 0x28], 0
,====< 0x08048a8a      e9d8110000. jmp 0x8049c67

```

Este código está obteniendo un valor apuntado por la base de la máquina virtual y [ebp-38] y lo guarda en una variable temporal (en este momento no sabemos para qué es usado ebp-38 en la máquina virtual). Si el valor obtenido no es cero incrementa el contenido de ebp-38 en 4 y directamente mete el valor obtenido en [ebp-34] que es el *vIP*.

Básicamente este código obtiene un valor del contexto de la VM apuntado por ebp-38 incrementa este puntero y finalmente introduce este valor en el *vIP*. Esto tiene toda la pinta de ser un “RET” y gracias a esta instrucción ahora sabemos que [ebp-38] es el puntero virtual a pila.

Seguimos el análisis con el siguiente valor de la tabla “0x08048a8f”.

“pd 200@0x8048a8f”

```

||      0x08048a8f      8b45cc      mov eax, dword [ebp - 0x34]
||      0x08048a92      83c001      add eax, 1
||      0x08048a95      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
||      0x08048a9c      0fbec0      movsx eax, al
||      0x08048a9f      8945f4      mov dword [ebp - 0xc], eax
||      0x08048aa2      8b45cc      mov eax, dword [ebp - 0x34]
||      0x08048aa5      83c002      add eax, 2
||      0x08048aa8      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
||      0x08048aaf      0fbec0      movsx eax, al
||      0x08048ab2      8945dc      mov dword [ebp - 0x24], eax
||      0x08048ab5      8b45f4      mov eax, dword [ebp - 0xc]
||      0x08048ab8      83f801      cmp eax, 1
,====< 0x08048abb      745e      je 0x8048b1b
||      0x08048abd      83f801      cmp eax, 1
,====< 0x08048ac0      7f09      jg 0x8048acb
||||     0x08048ac2      85c0      test eax, eax
,====< 0x08048ac4      7418      je 0x8048ade
,====< 0x08048ac6      e9d5000000. jmp 0x8048ba0
||`-----> 0x08048acb      83f802      cmp eax, 2
,====< 0x08048ace      747b      je 0x8048b4b
||||     0x08048ad0      83f804      cmp eax, 4
,====< 0x08048ad3      0fb4a2000000. je 0x8048b7b
,====< 0x08048ad9      e9c2000000. jmp 0x8048ba0
||`-----> 0x08048ade      8b45cc      mov eax, dword [ebp - 0x34]
| |||     0x08048ae1      83c003      add eax, 3
| |||     0x08048ae4      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
| |||     0x08048aeb      0fbec0      movsx eax, al
| |||     0x08048aee      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
| |||     0x08048af5      8945e0      mov dword [ebp - 0x20], eax
| |||     0x08048af8      8b45cc      mov eax, dword [ebp - 0x34]
| |||     0x08048afb      83c004      add eax, 4
| |||     0x08048afe      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
| |||     0x08048b05      0fbec0      movsx eax, al
| |||     0x08048b08      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
| |||     0x08048b0f      8945e4      mov dword [ebp - 0x1c], eax
| |||     0x08048b12      8345cc05      add dword [ebp - 0x34], 5
,====< 0x08048b16      e985000000. jmp 0x8048ba0
|`-----> 0x08048b1b      8b45cc      mov eax, dword [ebp - 0x34]
| ||      0x08048b1e      83c003      add eax, 3
| ||      0x08048b21      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;

```

```

|  || 0x08048b28 0fbec0 movsx eax, al
|  || 0x08048b2b 8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
|  || 0x08048b32 8945e0 mov dword [ebp - 0x20], eax
|  || 0x08048b35 8b45cc mov eax, dword [ebp - 0x34]
|  || 0x08048b38 83c004 add eax, 4
|  || 0x08048b3b 05c0c00408 add eax, 0x804c0c0
|  || 0x08048b40 8b00 mov eax, dword [eax]
|  || 0x08048b42 8945e4 mov dword [ebp - 0x1c], eax
|  || 0x08048b45 8345cc08 add dword [ebp - 0x34], 8
=====< 0x08048b49 eb55 jmp 0x8048ba0
-----> 0x08048b4b 8b45cc mov eax, dword [ebp - 0x34]
|  || 0x08048b4e 83c003 add eax, 3
|  || 0x08048b51 05c0c00408 add eax, 0x804c0c0
|  || 0x08048b56 8b00 mov eax, dword [eax]
|  || 0x08048b58 8945e0 mov dword [ebp - 0x20], eax
|  || 0x08048b5b 8b45cc mov eax, dword [ebp - 0x34]
|  || 0x08048b5e 83c007 add eax, 7
|  || 0x08048b61 0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
|  || 0x08048b68 0fbec0 movsx eax, al
|  || 0x08048b6b 8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
|  || 0x08048b72 8945e4 mov dword [ebp - 0x1c], eax
|  || 0x08048b75 8345cc08 add dword [ebp - 0x34], 8
=====< 0x08048b79 eb25 jmp 0x8048ba0
-----> 0x08048b7b 8b45cc mov eax, dword [ebp - 0x34]
|  || 0x08048b7e 83c003 add eax, 3
|  || 0x08048b81 05c0c00408 add eax, 0x804c0c0
|  || 0x08048b86 8b00 mov eax, dword [eax]
|  || 0x08048b88 8945e0 mov dword [ebp - 0x20], eax
|  || 0x08048b8b 8b45cc mov eax, dword [ebp - 0x34]
|  || 0x08048b8e 83c007 add eax, 7
|  || 0x08048b91 05c0c00408 add eax, 0x804c0c0
|  || 0x08048b96 8b00 mov eax, dword [eax]
|  || 0x08048b98 8945e4 mov dword [ebp - 0x1c], eax
|  || 0x08048b9b 8345cc0b add dword [ebp - 0x34], 0xb
|  || 0x08048b9f 90 nop
-----> 0x08048ba0 8b45e4 mov eax, dword [ebp - 0x1c]
|  || 0x08048ba3 8b55e0 mov edx, dword [ebp - 0x20]
|  || 0x08048ba6 01c2 add edx, eax
|  || 0x08048ba8 8b45dc mov eax, dword [ebp - 0x24]
|  || 0x08048bab 8994854cfff. mov dword [ebp + eax*4 - 0xb4], edx
|  || 0x08048bb2 8b45dc mov eax, dword [ebp - 0x24]
|  || 0x08048bb5 8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
|  || 0x08048bbc 8945d8 mov dword [ebp - 0x28], eax
=====< 0x08048bbf e9a3100000 jmp 0x8049c67

```

!!!SANTO DIOS QUE DEMONIO ES ESTO!!! Mantén la calma y tómate el análisis de este código con filosofía. Como se suele decir “perro ladrador poco mordedor”.

Seguramente estamos ante un op-code que maneja distintos direccionamientos, esto explicaría el gran tamaño de este pedazo de código. Pues nada, manos a la obra, para continuar voy a ir partiendo este código en trocitos mas pequeños que analizaremos por separado.

```

|  || 0x08048a8f 8b45cc mov eax, dword [ebp - 0x34]
|  || 0x08048a92 83c001 add eax, 1
|  || 0x08048a95 0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
|  || 0x08048a9c 0fbec0 movsx eax, al
|  || 0x08048a9f 8945f4 mov dword [ebp - 0xc], eax
|  || 0x08048aa2 8b45cc mov eax, dword [ebp - 0x34]
|  || 0x08048aa5 83c002 add eax, 2
|  || 0x08048aa8 0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ;
|  || 0x08048aaf 0fbec0 movsx eax, al
|  || 0x08048ab2 8945dc mov dword [ebp - 0x24], eax
|  || 0x08048ab5 8b45f4 mov eax, dword [ebp - 0xc]

```

```

    || 0x08048ab8 83f801    cmp eax, 1
    ,====< 0x08048abb 745e      je 0x08048b1b
    ||| 0x08048abd 83f801    cmp eax, 1
    ,====< 0x08048ac0 7f09      jg 0x08048acb
    |||| 0x08048ac2 85c0      test eax, eax
    ,====< 0x08048ac4 7418      je 0x08048ade
    ,====< 0x08048ac6 e9d5000000 jmp 0x8048ba0
    ||`-----> 0x08048acb 83f802    cmp eax, 2
    =====< 0x08048ace 747b      je 0x08048b4b
    || ||| 0x08048ad0 83f804    cmp eax, 4
    =====< 0x08048ad3 0f84a2000000 je 0x08048b7b
    =====< 0x08048ad9 e9c2000000 jmp 0x8048ba0

```

Este trozo inicial de código se encarga de descartar el byte del *op-code* y obtiene el siguiente byte en el *buffer* de ejecución y lo guarda en `ebp-0xc`, a continuación obtiene el siguiente byte del *buffer* de ejecución y lo guarda en `ebp-24`. Una vez a guardado estos valores recupera el primero de ellos e inicia una serie de comprobaciones para determinar cómo debe continuar.

Este comportamiento sin lugar a dudas es para determinar el tipo de direccionamiento que usa esta instrucción así como los parámetros implicados en ella, por el momento podríamos visualizarlo de esta forma:

[Op-code] [direccionamiento] ...

Pues podemos determinar sin lugar a dudas que el código inicial está obteniendo el byte de direccionamiento y en base a este selecciona cómo continuar.

Veamos que es lo que hace en cada caso.

. Direcccionamiento = 0: * REG,REG *.

```

; Obtiene el byte 3 del buffer de ejecucion actual.
0x08048ade mov eax, dword [ebp - 0x34] ; load into eax vIP
0x08048ae1 add eax, 3 ; add 3 to vIP
0x08048ae4 movzx eax, byte [eax + 0x804c0c0] ; load into eax the byte ptr at vIP
0x08048aeb movsx eax, al

; Carga en eax un valor que extrae de la pila en base al registro
0xb4 que resulta ser la base de los registros de la VM. Obtiene
el registro indicado por el byte extraido anteriormente.
0x08048aee mov eax, dword [ebp + eax*4 - 0xb4];
0x08048af5 mov dword [ebp - 0x20], eax

; extract from execution buffer four byte
0x08048af8 mov eax, dword [ebp - 0x34] ;
0x08048afb add eax, 4 ;
0x08048afe movzx eax, byte [eax + 0x804c0c0] ;
0x08048b05 movsx eax, al

; Obtiene el registro indicado por el byte extraido
anteriormente.
0x08048b08 mov eax, dword [ebp + eax*4 - 0xb4]
0x08048b0f mov dword [ebp - 0x1c], eax

; Incrementa el vIP en 5
0x08048b12 add dword [ebp - 0x34], 5 ; increment the vIP in 5
0x08048b16 jmp 0x8048ba0

```

El código que vemos arriba está obteniendo valores de los registros de la máquina virtual. Se determina que registro va a obtener usando la información que saca del *buffer* de ejecución, exactamente extrae el valor de dos registros de la máquina virtual lo guarda en dos variables temporales finalmente incrementa el *vIP* en 5 y continua. Podemos representar este op-code gráficamente de la siguiente forma:

[op-code] [p1] [p2] [p3] [p4]

El código que hemos visto extrae *p3* y *p4* y usa los valores obtenidos como índices del *array* de registros virtuales el cual se encuentra en *ebp-0xb4*. Y como puedes ver el tamaño de este op-code con este direccionamiento es de 5 bytes que es exactamente el número de bytes en los que *vIP* es incrementado.

Por lo tanto este direccionamiento podríamos representarlo así, pero recuerda que en este caso direccionamiento es igual a 0:

1 1 1 1 1
 [op-code] [direccionamiento] [unknown] [reg] [reg]

. Direcccionamiento = 1 * REG,IMM32 *.

```
; extract from execution buffer third byte
0x08048b1b mov eax, dword [ebp - 0x34]
0x08048b1e add eax, 3
0x08048b21 movzx eax, byte [eax + 0x804c0c0] ;
0x08048b28 movsx eax, al

; Load into eax a value from the vm reg context ebp-0xb4 is the
base of VM registers and use as index the third byte of execution
buffer.
0x08048b2b mov eax, dword [ebp + eax*4 - 0xb4]
; Save the retrieve value in temporal stack var.
0x08048b32 mov dword [ebp - 0x20], eax

; point eax to four byte of the execution buffer
0x08048b35 mov eax, dword [ebp - 0x34]
0x08048b38 add eax, 4
0x08048b3b add eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b40 mov eax, dword [eax]
; Save the retrieve value in temporal stack var.
0x08048b42 mov dword [ebp - 0x1c], eax

; Increment the value of virtual instruction pointer by 8
0x08048b45 add dword [ebp - 0x34], 8
0x08048b49 jmp 0x8048ba0
```

Aquí se está obteniendo un registro de la máquina virtual usando el tercer byte del *buffer* de ejecución como índice y un valor inmediato de 32 bits que se encuentra a partir del cuarto byte del *buffer* de ejecución y finalmente incrementa *vIP* en ocho. Su representación sería la siguiente cuando direccionamiento sea 1:

1	1	1	4	1
[op-code]	[direccionamiento]	[unknown]	[reg]	[dword]

. Direcccionamiento = 2 * IMM32,REG * .

```
; point eax to third byte of the execution buffer
0x08048b4b  mov  eax, dword [ebp - 0x34]
0x08048b4e  add  eax, 3
0x08048b51  add  eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b56  mov  eax, dword [eax]
; Save the retrieve value in temporal stack var.
0x08048b58  mov  dword [ebp - 0x20], eax

; point eax to seven byte of the execution buffer
0x08048b5b  mov  eax, dword [ebp - 0x34]
0x08048b5e  add  eax, 7
; load into EAX the byte at execution buffer
0x08048b61  movzx eax, byte [eax + 0x804c0c0] ;
0x08048b68  movsx eax, al

; Load into eax a value from the vm reg context ebp-0xb4 is the
base of VM registers and use as index the seven byte of execution
buffer.
0x08048b6b  mov  eax, dword [ebp + eax*4 - 0xb4]
0x08048b72  mov  dword [ebp - 0x1c], eax

; Increment the value of virtual instruction pointer by 8
0x08048b75  add  dword [ebp - 0x34], 8
0x08048b79  jmp  0x8048ba0
```

Es igual que el caso anterior pero con los parámetros cambiados y con valor 2 en direccionamiento.

1	1	1	4	1
[op-code]	[direccionamiento]	[desconocido]	[dword]	[reg]

. Direcccionamiento = 4 * IMM32,IMM32 * .

```
; point eax to third byte of the execution buffer
0x08048b7b  mov  eax, dword [ebp - 0x34]
0x08048b7e  add  eax, 3
0x08048b81  add  eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b86  mov  eax, dword [eax]

; Save the retrieve value in temporal stack var.
0x08048b88  mov  dword [ebp - 0x20], eax

; point eax to seven byte of the execution buffer
0x08048b8b  mov  eax, dword [ebp - 0x34]
0x08048b8e  add  eax, 7
0x08048b91  add  eax, 0x804c0c0

; load into EAX the dword at execution buffer
0x08048b96  mov  eax, dword [eax]

; Save the retrieve value in temporal stack var.
0x08048b98  mov  dword [ebp - 0x1c], eax

; Increment the value of virtual instruction pointer by 8
0x08048b9b  add  dword [ebp - 0x34], 0xb
0x08048b9f  nop
```

Este código obtiene el *dword* que se encuentre en el *buffer* de ejecución empezando por el tercer byte y el *dword* empezando por séptimo byte, por ultimo incrementa *vIP* en 11. Este caso lo podemos representar de la siguiente forma donde su direccionamiento tiene el valor 4:

1	1	1	4	4
[op-code]	[direccionamiento]	[desconocido]	[DWORD]	[DWORD]

. La operación real del op-code .

```

;Load into eax the saved value3
0x08048ba0  mov  eax, dword [ebp - 0x1c]

;load into edx the save value2
0x08048ba3  mov  edx, dword [ebp - 0x20]

; do the operation ADD
0x08048ba6  add  edx, eax

; Load into eax the save value1
0x08048ba8  mov  eax, dword [ebp - 0x24]

; Write into vm reg at index selected by the previous
instruction, the result of the add operation (edx)
0x08048bab  mov  dword [ebp + eax*4 - 0xb4], edx

0x08048bb2  mov  eax, dword [ebp - 0x24]
0x08048bb5  mov  eax, dword [ebp + eax*4 - 0xb4]
0x08048bbc  mov  dword [ebp - 0x28], eax
; Process next opcode
0x08048bbf  jmp  0x8049c67

```

Este último trozo de código es donde realmente se realiza la operación de este op-code y se guarda su resultado en el contexto de la máquina virtual. Fijaos cómo el código lo único que hace es obtener los valores que se guardaron anteriormente en variables temporales y opera con ellos, en este caso realiza una suma “add edx, eax” y guarda el resultado dentro de otro registro de la VM el cual se determina usando el parámetro *p2* desconocido hasta ahora, por lo tanto donde en un principio teníamos esta representación:

[op-code] [p1] [p2] [p3] [p4]

Ahora tenemos esta:

[ADD] [direccionamiento] [dest_reg] [x] [y]

Los valores de x e y dependerán del tipo de direccionamiento que indique este byte. Y nos indicaran si lo que tenemos que obtener es el índice a algún registro o un valor inmediato.

SkUaTeR

Esta instrucción puede representarse de la siguiente forma:

- si direccionamiento=0: reg_des = reg[x] + reg[y]
- si direccionamiento=1: reg_des = reg[x] + imm32
- si direccionamiento=2: reg_des = imm32 + reg[y]
- si direccionamiento=4: reg_des = imm32 + imm32

Ahora ya conocemos dos op-codes de esta máquina virtual. Dos de ellos son simples y no usan ningún tipo de direccionamiento y el ultimo que hemos analizado es algo mas complejo y si que tiene direccionamiento. Dejame mostrártelo en una tabla para simplificar las cosas:

Nº Op-code	Nombre	Longitud	0	1	2	4
0	NOP	1				
1	RET	1				
2	ADD	5/8/8/11	r0=r1+r2	r0=r1+i32_2	r0=i32_1+r2	r0=i32_1+i32_2

En esta tabla estoy nombrando los registros como r0,r1,.. y los valores inmediatos como i32_1,i32_2,...

SkUaTeR

Si en el *buffer* de ejecución tenemos un byte “0” esto significará que estamos ante una instrucción “nop” y su tamaño es de un byte.

Si en el *buffer* de ejecución tenemos un byte “1” esto significará que estamos ante una instrucción “ret” y de nuevo su tamaño será de un byte.

Si en el *buffer* de ejecución tenemos un byte “2” esto significará que estamos ante un “add” y en este caso tendremos que inspeccionar el siguiente byte del *buffer* de ejecución, para poder determinar que direccionamiento, tamaño usa este *op-code*.

- Si el byte posterior al *op-code* es “0” la longitud será de 5 bytes y necesitaremos obtener los 3 bytes siguientes al direccionamiento para obtener los indices a *r0,r1* y *r2*.
- Si el byte posterior al *op-code* es “1” la longitud será de 8 bytes y necesitaremos extraer 2 bytes para determinar el *r0* y el *r1* y un *dword* para el valor del *i32_2*.
- Y así para el resto de direccionamientos según nos indique la tabla.

Si pudiéramos construir una tabla con todos los *op-codes*, esto nos daría el poder necesario para obtener todo el código virtual y poder resolver este reto.

Por lo tanto vamos a continuar analizando la tabla de *op-codes* para descubrir el resto de instrucciones usadas por esta máquina virtual. Tras el análisis del *op-code* “add” tenemos una imagen de cómo es el código que gestiona los direccionamientos y gracias a este análisis obtener el resto de *op-codes* va a ser bastante rápido.

Veamos el siguiente valor de la tabla “0x08048bc4” para continuar con el análisis. “pd 20000x8048bc4”:

```

0x08048bc4      8b45cc      mov eax, dword [ebp - 0x34]
0x08048bc7      83c001      add eax, 1
0x08048bca      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048bd1      0fbec0      movsx eax, al
0x08048bd4      8945f4      mov dword [ebp - 0xc], eax
0x08048bd7      8b45cc      mov eax, dword [ebp - 0x34]
0x08048bda      83c002      add eax, 2
0x08048bdd      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048be4      0fbec0      movsx eax, al
0x08048be7      8945dc      mov dword [ebp - 0x24], eax
0x08048bea      8b45f4      mov eax, dword [ebp - 0xc]
0x08048bed      83f801      cmp eax, 1
0x08048bf0      745e      je 0x8048c50
0x08048bf2      83f801      cmp eax, 1
0x08048bf5      7f09      jg 0x8048c00
0x08048bf7      85c0      test eax, eax
0x08048bf9      7418      je 0x8048c13
0x08048bfb      e9d5000000 jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c00      83f802      cmp eax, 2
0x08048c03      747b      je 0x8048c80
0x08048c05      83f804      cmp eax, 4
0x08048c08      0f84a2000000 je 0x8048cb0
0x08048c0e      e9c2000000 jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c13      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c16      83c003      add eax, 3
0x08048c19      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c20      0fbec0      movsx eax, al
0x08048c23      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048c2a      8945e0      mov dword [ebp - 0x20], eax
0x08048c2d      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c30      83c004      add eax, 4
0x08048c33      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c3a      0fbec0      movsx eax, al
0x08048c3d      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048c44      8945e4      mov dword [ebp - 0x1c], eax
0x08048c47      8345cc05      add dword [ebp - 0x34], 5
0x08048c4b      e985000000 jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c50      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c53      83c003      add eax, 3
0x08048c56      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c5d      0fbec0      movsx eax, al
0x08048c60      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048c67      8945e0      mov dword [ebp - 0x20], eax
0x08048c6a      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c6d      83c004      add eax, 4
0x08048c70      05c0c00408 add eax, 0x804c0c0
0x08048c75      8b00      mov eax, dword [eax]
0x08048c77      8945e4      mov dword [ebp - 0x1c], eax
0x08048c7a      8345cc08      add dword [ebp - 0x34], 8
0x08048c7e      eb55      jmp 0x8048cd5 ; (fcn.0804898b)
0x08048c80      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c83      83c003      add eax, 3
0x08048c86      05c0c00408 add eax, 0x804c0c0
0x08048c8b      8b00      mov eax, dword [eax]
0x08048c8d      8945e0      mov dword [ebp - 0x20], eax
0x08048c90      8b45cc      mov eax, dword [ebp - 0x34]
0x08048c93      83c007      add eax, 7
0x08048c96      0fb680c0c0. movzx eax, byte [eax + 0x804c0c0] ; [0x804c0c0:1]=0
0x08048c9d      0fbec0      movsx eax, al
0x08048ca0      8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048ca7      8945e4      mov dword [ebp - 0x1c], eax
0x08048caa      8345cc08      add dword [ebp - 0x34], 8
0x08048cae      eb25      jmp 0x8048cd5 ; (fcn.0804898b)
0x08048cb0      8b45cc      mov eax, dword [ebp - 0x34]
0x08048cb3      83c003      add eax, 3
0x08048cb6      05c0c00408 add eax, 0x804c0c0
0x08048cbb      8b00      mov eax, dword [eax]

```

```

0x08048cbd  8945e0    mov dword [ebp - 0x20], eax
0x08048cc0  8b45cc    mov eax, dword [ebp - 0x34]
0x08048cc3  83c007    add eax, 7
0x08048cc6  05c0c00408 add eax, 0x804c0c0
0x08048ccb  8b00      mov eax, dword [eax]
0x08048ccd  8945e4    mov dword [ebp - 0x1c], eax
0x08048cd0  8345cc0b  add dword [ebp - 0x34], 0xb
0x08048cd4  90        nop
0x08048cd5  8b45e4    mov eax, dword [ebp - 0x1c]
0x08048cd8  8b55e0    mov edx, dword [ebp - 0x20]
0x08048cdb  29c2      sub edx, eax
0x08048cdd  8b45dc    mov eax, dword [ebp - 0x24]
0x08048ce0  8994854cfff. mov dword [ebp + eax*4 - 0xb4], edx
0x08048ce7  8b45dc    mov eax, dword [ebp - 0x24]
0x08048cea  8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048cf1  8945d8    mov dword [ebp - 0x28], eax
0x08048cf4  e96e0f0000 jmp 0x8049c67 ; (fcn.0804898b)

```

Como puedes ver este código es muy parecido al que hemos analizado anteriormente. Ignoremos todo el tema de direccionamiento y obtención de parámetros y vamos a buscar dónde se realiza la verdadera operación:

```

0x08048cd4  90        nop
0x08048cd5  8b45e4    mov eax, dword [ebp - 0x1c]
0x08048cd8  8b55e0    mov edx, dword [ebp - 0x20]
0x08048cdb  29c2      sub edx, eax
0x08048cdd  8b45dc    mov eax, dword [ebp - 0x24]
0x08048ce0  8994854cfff. mov dword [ebp + eax*4 - 0xb4], edx
0x08048ce7  8b45dc    mov eax, dword [ebp - 0x24]
0x08048cea  8b84854cfff. mov eax, dword [ebp + eax*4 - 0xb4]
0x08048cf1  8945d8    mov dword [ebp - 0x28], eax
0x08048cf4  e96e0f0000 jmp 0x8049c67 ; (fcn.0804898b)

```

En “0x08048cdb sub edx, eax” es donde se realiza la verdadera operación la cual podemos ver cómo es un “SUB”.

Todos los *op-codes* que tiene direccionamiento son gestionados casi de la misma forma, existe una excepción que es la instrucción “DIV”, para esta instrucción se utiliza un parámetro mas con el cual se indica en que registro de la máquina virtual se guardara el resto de la operación de división. Pero esto no nos debe preocupar ya que si realizamos un análisis concienzudo, nos daremos cuenta de esto sin problemas.

Una vez que tengamos identificados todos los *op-codes*, debería ser bastante fácil identificar variables especiales como los *Eflags*. Recuerda cómo la instrucción “ret” nos desveló donde se alojaba el registro de pila.

. Saltos Condicionales & Registro de Eflags.

Por último voy a intentar explicar cómo funcionan los saltos condicionales y cómo estos *op-codes* nos revelan el registro de *Eflags*. “pd 12@0x080496ec”

```

; Load vIP and increment by 1
0x080496ec  8b45cc    mov eax, dword [ebp - 0x34]
0x080496ef  83c001    add eax, 1

```

```
0x080496f2  05c0c00408  add eax, 0x804c0c0
; Load into eax a dword from instruction buffer and save into tmp var
0x080496f7  8b00        mov eax, dword [eax]
0x080496f9  8945f0      mov dword [ebp - 0x10], eax
; Test the var at [ebp-0x28] with 0, this var its the Eflags
0x080496fc  837dd800    cmp dword [ebp - 0x28], 0
; This je is the "op-code"
0x08049700  7408        je 0x804970a
; if value at ebp-28 dont is 0 increment the vIP by 5.
0x08049702  8b45cc      mov eax, dword [ebp - 0x34]
0x08049705  83c005      add eax, 5
0x08049708  eb03        jmp 0x804970d
; if the value was 0 the load the temp var with destination address
0x0804970a  8b45f0      mov eax, dword [ebp - 0x10]
; set te eax value into vIP
0x0804970d  8945cc      mov dword [ebp - 0x34], eax
```

En este código podemos observar cómo se extrae el siguiente *dword* del *buffer* de instrucciones y se guarda en una variable temporal, después se realiza un “`cmp dword [ebp - 0x28], 0`” y un “`je 0x804970a`”, estas instrucciones están leyendo el valor alojado en `[ebp-0x28]` y si es 0, actualiza el *puntero de instrucción virtual* (`[ebp-0x34]`) con el valor guardado en `[ebp-0x10]` (este valor es el extraído del *buffer* de ejecución). Si no es cero se incrementa el *puntero de instrucción virtual* en 5 y se devuelve el control a la rutina principal.

Ahora podemos ver para qué es usado `[ebp-0x28]`, sin lugar a dudas está siendo usado por la VM para guardar el resultado de las operaciones lógicas. En otras palabras es el *Registro Virtual de Eflags*.

Veamos ahora la instrucción “`cmp`” para asegurarnos de que todo lo visto es correcto. Si existe una instrucción implicada en los *Eflags* sin lugar a dudas será esta. “`pd 8@0x080499ed`”:

```
0x080499ed  90          nop
0x080499ee  8b45e4      mov eax, dword [ebp - 0x1c]
0x080499f1  8b55e0      mov edx, dword [ebp - 0x20]
0x080499f4  89d1        mov ecx, edx
0x080499f6  29c1        sub ecx, eax
0x080499f8  89c8        mov eax, ecx
0x080499fa  8945d8      mov dword [ebp - 0x28], eax
0x080499fd  e965020000 jmp 0x8049c67
```

Como los demás *op-codes* éste usa distintos direccionamientos, aquí solo vamos a mostrar el código principal de la instrucción. Podemos ver cómo este *op-code* realmente realiza un “`sub`”. En la arquitectura *x86* la instrucción “`sub`” está muy implicada en la manipulación de *Eflags*. Básicamente un “`cmp`” y un “`sub`” actúan de la misma forma, de hecho en *x86* la única diferencia es que el “`cmp`” no guarda el resultado, simplemente actualiza los *Eflags*. Esta máquina virtual usa un método bastante interesante, realiza la resta de dos valores seleccionados en base a su direccionamiento, y guarda el resultado en `[ebp-0x28]`, de esta forma más tarde solo

tendrá que mirar este valor para tomar la decisión correcta ante un “jne”, “je”, “ja”, “jb” ...

Bien, pues hasta aquí la forma de analizar una máquina virtual. Espero que hayáis entendido todo este tocho. Y bueno, sólo queda analizar el resto op-codes de la tabla y podremos inferir completamente el comportamiento de esta VM e incluso crear un interprete para ella.

. Virtual Machine Instruction Set .

Nº	Nombre	Representación	Longitud	0	1	2	4
00	NOP	nop	1				
01	RET	ret	1				
02	ADD	addressing1	5/8/8/11	r0=r1+r2	r0=r1+i32_2	r0=i32_1+r2	r0=i32_1+i32_2
03	SUB	addressing1	5/8/8/11	r0=r1-r2	r0=r1-i32_2	r0=i32_1-r2	r0=i32_1-i32_2
04	MUL	addressing1	5/8/8/11	r0=r1*r2	r0=r1*i32_2	r0=i32_1*r2	r0=i32_1*i32_2
05	DIV	addressing5	6/9/9/12				
06	XOR	addressing1	5/8/8/11	r0=r1^r2	r0=r1^i32_2	r0=i32_1^r2	r0=i32_1^i32_2
07	NEG	r0=neg r1	3				
08	NOT	r0=not r1	3				
09	AND	addressing1	5/8/8/11	r0=r1&r2	r0=r1&i32_2	r0=i32_1&r2	r0=i32_1&i32_2
10	OR	addressing1	5/8/8/11	r0=r1 r2	r0=r1 i32_2	r0=i32_1 r2	r0=i32_1 i32_2
11	SETZ	r0=r1==0	3				
12	ROL	addressing1	5/8/8/11	r0=r1<<r2	r0=r1<<i32_2	r0=i32_1<<r2	r0=i32_1<<i32_2
13	ROR	addressing1	5/8/8/11	r0=r1>>r2	r0=r1>>i32_2	r0=i32_1>>r2	r0=i32_1>>i32_2
14	JMP	jmp i32_0	5				
15	CALL	call i32_0	5				
16	JZ	jz i32_0	5				
17	JS	js i32_0	5				
18	JBE	jbe i32_0	5				
19	JG	jg i32_0	5				
20	JNS	jns i32_0	5				
21	JNZ	jnz i32_0	5				
22	AND	addressing2	4/7/7/10	r1 & r2	r1 & i32_2	i32_1 & r2	i32_1 & i32_2
23	CMP	addressing2	4/7/7/10	cmp r1,r2	cmp r1,i32_2	cmp i32_1,r2	Cmp i32_1,i32_1
24	MOV	addressing3	4/7	mov r1,r2	mov r1,i32_2		
25	++	r0++	2				
26	--	r0--	2				
27	MOV	mov r1,[r2]	3	M r,[r]			
28	MOV	Mov [r1],r2	3	M [r],r			
29	ENDVM		1				
30	PUSH	addressing4	3/6	push r1	Push i32_1		
31	POP	pop r0	2				
32	NCALL	call idx	2				

Fijaos en la columna “*Representación*”, después de terminar el análisis me di cuenta de que son usados cinco tipos distintos de direccionamientos con sus correspondientes tamaños:

- Addressing1: 5/8/8/11, este tipo de direccionamiento es el que hemos analizado anteriormente, siempre usa un registro de destino el cual se obtiene a partir del byte siguiente al direccionamiento.
- Addressing2: 4/7/7/10, es idéntico al anterior pero no se guarda el resultado en un registro especificado por parámetro, en su lugar el resultado es guardado en registro *Eflags* de la máquina virtual.
- Addressing3: 4/7, Este direccionamiento solo es usado por un tipo de “mov”. Y maneja parámetros del tipo *r,r* y *r,im32*
- Addressing4: 3/6, Solo es usado por “push”, maneja los parámetros de tipo *r* e *im32*.
- Addressing5: 6/9/9/12. Idéntico a Addressing1, pero usa un byte más para indicar en que registro guardar el resto de la operación “div”. (solo usado por div)

Recordemos cómo se ve el un op-code en el *buffer* de ejecución:

[op-code] (direccionamiento)

Como puedes ver, direccionamiento aparece entre paréntesis porque es opcional, no todo los *op-codes* lo usan.

Veamos ahora el *buffer* real de ejecución para las primeras instrucciones de este código virtual. “px 256@0x0804d0c0”:

```
[0x0804cfd4]> px 256@0x0804d0c0
- offset -    0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x0804d0c0  1801 003a 1000 0018 0101 db1e 0000 1800  ....:.....
0x0804d0d0  0300 1801 0542 cf74 011b 0403 0600 0404  ....B.t.....
0x0804d0e0  051c 0304 0201 0303 0400 0000 1700 0103  ....:.....
0x0804d0f0  1419 1000 000e 3a10 0000 4c0f 6f01 42ef  ....:...L.o.B.
0x0804d100  7400 62ce 7521 40ce 5402 43ef 7000 62ca  t.b.u!@.T.C.p.b.
0x0804d110  7521 44ce 5406 43ef 7c00 62c6 7521 48ce  u!D.T.C.|.b.u!H.
0x0804d120  540a 43ef 7800 62c2 7521 4cce 540e 43ef  T.C.x.b.u!L.T.C.
0x0804d130  6400 55cf 7401 52b3 6401 42ce 6921 53ce  d.U.t.R.d.B.i!S.
0x0804d140  6a01 5cd1 741c 5acf 6a00 46cf 741f 42cd  j.\.t.Z.j.F.t.B.
0x0804d150  7501 42c7 7401 42d7 741c 42ef 6517 42cf  u.B.t.B.t.B.e.B.
0x0804d160  7411 80df 7401 5acf 701c 4fce 7005 41cf  t...t.Z.p.O.p.A.
0x0804d170  7401 5ecf 7003 43cf 7409 42cf 741e 5fd0  t.^.p.C.t.B.t._.
0x0804d180  6a00 5fd7 7501 01cf 7401 4df0 6401 42d7  j._.u...t.M.d.B.
0x0804d190  7501 2dcf 7401 4df0 6401 42d7 7501 2ccf  u.-.t.M.d.B.u.,.
0x0804d1a0  7401 4df0 6401 42d7 7501 25cf 7401 4df0  t.M.d.B.u.%.t.M.
0x0804d1b0  6401 42d7 7501 30cf 7401 4df0 6401 42d7  d.B.u.0.t.M.d.B.
```

Recuerda que los valores que estamos viendo están en hexadecimal y así evitaremos confusiones al tratar de identificar el *op-code*.

El primer valor es 0x18 (24) si lo buscamos en la tabla vemos lo siguiente:

24	MOV	addressing3	4/7	mov r1,r2	mov r1,i32_2
----	-----	-------------	-----	-----------	--------------

Podemos ver cómo es de tipo “addressing3”, lo que significa que necesitamos mirar el siguiente byte para determinar el tipo de direccionamiento y los parámetros que extrae, el valor del siguiente byte es 0x01 lo que nos dice que la longitud es de 7 y que los parámetros a extraer son 2 índices a registros de la VM. De la tabla anterior podemos extraer el tamaño del op-code y su representación. Si la instrucción es de tipo “addressingX”, cogeremos el byte de direccionamiento y buscaremos su valor en las columnas numérica “0,1,2,4” esto nos dará la forma de representarlo. Luego vamos a la columna “Longitud” y obtenemos el tamaño también en base al byte de direccionamiento, es decir si Longitud es “4/7” y el byte de direccionamiento es “0” la longitud será de 4 ya que estamos cogiendo el primer Longitud, si el valor de direccionamiento fuera 1 la longitud sería 7.

Vamos a verlo usando colorines:

```
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x0804d0c0 1801 003a 1000 0018 0101 db1e 0000 1800 .....
0x0804d0d0 0300 1801 0542 cf74 011b 0403 0600 0404 .....B.t.....
```

Rojo: op-code. MOV.

Amarillo: direccionamiento, r1, i32_1


Verde: índice del registro a obtener del contexto de registros de la máquina virtual, reg_0.

Azul: i32_1, es el valor inmediato a extraer como segundo parámetro del *op-code*, 0x0000103a.

La instrucción decodificada es “mov r0,0x0000103a”, a la que le sigue otro “mov r1,0x00001edb” y otro más pero con un direccionamiento diferente 0: “mov r3,r0”.

Espero que con este ejemplo haya quedado claro cómo usar la tabla para decodificar las instrucciones de esta máquina virtual. Usando la tabla no sería difícil obtener todo el código pero sería mucho mejor crear una utilidad que lo obtenga por nosotros y nos ahorre algo de tiempo.

Implementando un Intérprete .

Es el momento de codificar una herramienta que nos ayude a obtener todo el Código Virtual. Aquí tenéis mi código un simple interprete: [\[decoder.c\]](#) 

El código es muy sencillote, sólo contiene una función que se usa para extraer y formatear los parámetros del *buffer* de ejecución y finalmente devolver el tamaño de op-code en base a su direccionamiento.

Y un enorme *switch/case* usado para manejar cada *op-code*, y llamar a la función de la que hablábamos arriba, la cual obtiene los *paramentros/tamaño* y finalmente imprime la salida formateada.

Una vez tenemos el interprete obviamente este necesita algo que interpretar, por lo tanto es el momento de extraer “*el buffer de ejecución*” que se encuentra dentro del archivo binario. Cuando realicé el reto *ripeé* todo el contexto de la máquina virtual y el código del `decoder.c` necesita el contexto entero. Procedamos a ello, necesitamos crear un archivo “*vm.code*”. Nada mejor para realizar el trabajo que *r2* ya que también podemos usarlo como editor hexadecimal. Recordemos cómo el contexto de la máquina virtual se encontraba en la dirección “0x804C0C0” por lo tanto tendremos que realizar el volcado a partir de esta dirección.

Ejecuta *r2* y escribe los siguientes comandos “s 0x804C0C0” y “v” para entrar en modo visual:

```
[0x0804c0c0 368 baleful]> x @ section..data+128 # 0x804c0c0
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x0804c0c0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c0d0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c0e0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c0f0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c100  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c110  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c120  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c130  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c140  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c150  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c160  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c170  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c180  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c190  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c1a0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c1b0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c1c0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c1d0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c1e0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c1f0  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0804c200  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Iremos usando las tecla “AvPag Key” para inspeccionar todos estos bytes y determinar el tamaño de la VM (las primeras instrucciones de la máquina virtual se

encuentra a 0x1000 bytes de desplazamiento desde el inicio de su contexto). Por lo tanto vamos a ir avanzado para poder determinar donde acabaría la máquina virtual.

No importa si volcamos bytes de mas, en la imagen de abajo podéis ver ya donde acaba el *buffer de ejecución*.

```
[0x0804dec0 322 baleful]> x @ section..data+7808 # 0x804dec0
- offset -    0 1  2 3  4 5  6 7  8 9  A B  C D  0123456789ABCD
0x0804dec0    7401 42c0 4b11 42cf 6c00 42a0 7401 42c0  t.B.K.B.l.B.t.
0x0804dece    42c0 4b11 42cf 6c00 42a1 7401 42c0  B.K.B.l.B.t.B.
0x0804dedc    4b11 42cf 6c00 42a8 7401 42c0 4b11  K.B.l.B.t.B.K.
0x0804deea    42cf 6c00 42ef 7401 42c0 4b11 42cf  B.l.B.t.B.K.B.
0x0804def8    6c00 42bf 7401 42c0 4b11 42cf 6c00  l.B.t.B.K.B.l.
0x0804df06    42ae 7401 42c0 4b11 42cf 6c00 42bc  B.t.B.K.B.l.B.
0x0804df14    7401 42c0 4b11 42cf 6c00 42bc 7401  t.B.K.B.l.B.t.
0x0804df22    42c0 4b11 42cf 6c00 42b8 7401 42c0  B.K.B.l.B.t.B.
0x0804df30    4b11 42cf 6c00 42a0 7401 42c0 4b11  K.B.l.B.t.B.K.
0x0804df3e    42cf 6c00 42bd 7401 42c0 4b11 42cf  B.l.B.t.B.K.B.
0x0804df4c    6c00 42ab 7401 42c0 4b11 42cf 6c00  l.B.t.B.K.B.l.
0x0804df5a    42ee 7401 42c0 4b11 42cf 6c00 42c5  B.t.B.K.B.l.B.
0x0804df68    7401 42c0 4b11 42cf 6c00 42cf 7401  t.B.K.B.l.B.t.
0x0804df76    42d0 7f1e 48d0 7d00 5acf 740a 4d66  B...H.}.Z.t.Mf
0x0804df84    6601 42d7 7501 42cf 7401 5dc4 6b0b  f.B.u.B.t.].k.
0x0804df92    5dc6 751c 5fd7 6c19 0000 0000 0000  ].u._.l.....
0x0804dfa0    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfa6    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfb4    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfc2    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfd0    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfd8    0000 0000 0000 0000 0000 0000 0000  .....
```

Ahora podemos determinar el tamaño del bloque a volcar que es desde 0x804c0c0 hasta 0x804dfd8, un total de 0x1f18 bytes.

Usando el siguiente comando de *r2* podemos volcar directamente a fichero:
“s 0x804c0c0” and “wt vm.code.r2 0x1f18”

```
0x0804df8a    42cf 7401 5dc4 6b0b 5dc6 751c 5fd7  B.t.].k.].u._.
0x0804df98    6c19 0000 0000 0000 0000 0000 0000  l.....
0x0804dfa6    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfb4    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfc2    0000 0000 0000 0000 0000 0000 0000  .....
0x0804dfd0    0000 0000 0000 0000 0000 0000 0000  .....
[0x0804c0c0]> wt vm.code 0x1f18
dumped 0x1f18 bytes
Dumped 7960 bytes from 0x0804c0c0 into vm.code
[0x0804c0c0]>
```

Finalmente con *el contexto de la vm* y nuestra utilidad *encode.exe* podemos obtener la representación del código virtual con el cual finalizaremos este reto.

*Las primeras instrucciones de la máquina virtual se encargan de
descifrar el resto de las instrucciones a ejecutar.
El archivo vm.code final deberá ser descifrado pero esto lo explicará mas
adelante mi compañero thEpOpE*

SkUaTeR

Bonus Stage: Plug-in para r2.

Radare2 es una herramienta imprescindible para el análisis de binarios pero además tiene cientos de funcionalidades, además de un magnífico sistema de *plugins* con el cual podemos añadir nuevas arquitecturas. A continuación vamos a explicar cómo implementar un *plugin* de tipo “asm” con el cual podamos interpretar el código de esta máquina virtual. Esto será casi un chiste ya que podemos reutilizar la mayor parte del interprete en C que hicimos anteriormente.

R2 solo necesita un archivo para definir un *plug-in*. Pero será necesario que conozcamos cómo es la estructura básica de un *plugin* “asm”.

La mejor forma de documentarnos sobre *r2* es visitar su blog periódicamente e inspeccionar su propio código fuente. Os dejo aquí la dirección: <http://radare.today>.

Precisamente hace menos de un mes un artículo sobre cómo extender las funcionalidades de *r2* con *plugins*, fue publicado “<http://radare.today/extending-r2-with-new-plugins>”. Si necesitas más información no dudes en visitarlo, aquí Pancake explica cómo funciona el sistema y nos muestra un simple ejemplo:

```
static int disassemble(RAsm *a, RAsmOp *op, ut8 *buf, ut64 len) {
    /* TODO: Implement disassemble code here,
     * give a look to the other plugins */
}

RAsmPlugin r_asm_plugin_myarch = {
    .name = "MyArch",
    .desc = "disassembly plugin for MyArch",
    .arch = "myarch",
    .bits = (int[]){ 32, 64, 0 }, /* supported wordsizes */
    .init = NULL,
    .fini = NULL,
    .disassemble = &disassemble,
    .modify = NULL,
    .assemble = NULL,
};

#ifdef CORELIB
struct r_lib_struct_t radare_plugin = {
    .type = R_LIB_TYPE_ASM,
    .data = &r_asm_plugin_myarch
};
#endif
```

Por lo tanto si reciclamos nuestro código en C y le añadimos unas pequeñas modificaciones obtenemos el siguiente archivo: [[código plug-in r2](#)]



Ahora sólo deberíamos compilar, pero en este caso vamos a crear un *plug-in* dinámico que se cargue cuando *r2* sea invocado. De esta forma sólo deberemos copiar el *plug-in* en el directorio “*plug-ins*” de *r2* y podremos obtener un desensamblado completo de la máquina virtual.

Para poder realizar la compilación será necesario que nuestro *r2* haya sido instalado del repositorio *GIT*, así podremos enlazar nuestro código contra las librerías de *r2*. Usa el siguiente comando para realizar la compilación:

```
gcc -shared -fpic asm_baleful.c -o asm_baleful.so $(pkg-config --cflags --libs r_asm)
```

será necesario tener instalado el paquete “*pkg-tools*” en nuestro sistema. Este paquete se encargara de indicarle a *gcc* las opciones del *linker*. El proceso de compilación es muy rápido y nos dará como resultado “*asm_baleful.so*”, ahora sólo necesitamos copiar este archivo en el directorio de *plug-ins* de *r2*. Podemos averiguar este directorio ejecutando *radare2* con el siguiente comando, “*r2 -hh*”:

```
...
Plugins:
plugins  /usr/lib/radare2/last
user     ~/.config/radare2/plugins
LIBR_PLUGINS /usr/lib/radare2/0.9.9-git
```

Ahora que conocemos el directorio sólo tenemos que realizar la copia del archivo *asm_baleful.so* dentro del directorio “*~/.config/radare2/plugins*” (si no existe el directorio solo tendremos que crearlo).

Esto es todo, ya tenemos nuestro *r2* preparado para realizar un desensamblado de la máquina virtual de “*baleful*”.

Ejecutemos el siguiente comando, “*r2 vm.code*”:

```
root@vmkalix86:~/picocft/dbgsrv# r2 vm.code
-- Ask not what r2 can do for you - ask what you can do for r2
[0x00000000]> s 0x1000
[0x00001000]> e asm.arch=baleful
[0x00001000]> pd 10
0x00001000 1801003a100. mov r_00 0x103a
0x00001007 180101db1e0. mov r_01 0x1edb
0x0000100e 18000300    mov r_03 r_00
0x00001012 18010542cf7. mov r_05 0x174cf42
0x00001019 1b0403      mov r_04 r_03
0x0000101c 0600040405  r_04 = r_04 ^ r_05
0x00001021 1c0304      mov r_03 r_04
0x00001024 02010303040. r_03 = r_03 0x0004
0x0000102c 17000103    r_01 cmp r_03
0x00001030 1419100000  jns 0x1019
[0x00001000]>
```

Usamos el comando “*s*” para decirle a *r2* que vaya a ese *offset*, y luego mediante el comando “*e*” le vamos a indicar qué arquitectura de asm queremos cargar:

```
e asm.arch=baleful
```

Finalmente con nuestra arquitectura cargada, solo tenemos que realizar el comando de desensamblado “*pd 10*”.

Pues esto es todo sobre el análisis de una VM y de cómo implementar un desensamblador para ella.

Espero que hayáis disfrutado leyendo este documento y que continúes disfrutando con mi compañero “thEpOpE”. Con el cual os dejo para que continúe explicando cómo interpretó el código de la VM.

Ha sido un placer nos vemos pronto!!!

*Agradecimientos especiales a Pancacke (@trufae @radareorg)
por todos sus consejos y por el soporte que está dando a esta
genial herramienta r2.*


[SkUaTeR](#)

Punk
Not
Dead.

. Código Virtual – Análisis de Código – Análisis Virtual .

. Elige: Susto o mu3rt3 .

Mientras SkUaTeR se dedicaba a desarrollar una herramienta para poder decodificar todo el código virtual, yo me dediqué a analizar la función que gestiona el *opcode* `ncall`. Este *opcode* tiene un parámetro que se usa para buscar en una tabla con funciones. Principalmente es una instrucción usada para añadir nuevas funcionalidades a la VM. En este, en concreto, hay 17 funciones nativas que pueden ser llamadas.

Una vez que SkUaTeR ya tuvo todo el código virtual en un formato parecido al assembler, me lo envió en un archivo de texto plano, sí sin colorines en la sintaxis. Aquí está el código originalmente usado. [[código virtual](#)] 

A partir de aquí había dos opciones: Susto o Mu3rt3.

Susto : Una vez tenemos identificado el *Instruction Pointer*, podríamos ejecutar todo el código de la máquina virtual, y detenerla cuando se llegue a una dirección concreta del código virtual. De este modo sería como poner un *breakpoint* en el código virtual. Bastaría poner un *breakpoint* condicional en la dirección donde la VM carga el siguiente *op-code*. También podríamos poner una condición con un valor determinado de *op-code*, y así se detendría cuando se fuera a ejecutar ese *op-code* (podríamos detenernos, por ejemplo, cada vez que se vaya a ejecutar un salto condicional).

Mu3rt3 : La otra opción es no ejecutar para nada la VM, y tratar de entender qué demonios hace el código virtual. Como nos gustan los retos duros... este fue el camino que tomamos: analizar *d3ad c0d3*.

Sabemos que el código virtual de la máquina se inicia en el dirección virtual `0x1000`, y que en un estado inicial desde `0x0000` hasta `0x0FFF` tiene la memoria puesta a `0x00`. (A partir de este punto todas las referencias a direcciones son direcciones virtuales, es decir, de la memoria virtual de la VM).

Así es cómo comienza el programa virtual:

```
1000:  mov r_00,0x103a
1007:  mov r_01,0x1edb
100e:  mov r_03,r_00
1012:  mov r_05,0x174cf42
1019:  mov r_04,[r_03]
101c:  r_04=r_04 ^ r_05
1024:  mov [r_03],r_04
1028:  r_03=r_03 + 0004
102c:  cmp r_01,r_03
1030:  jns 0x1019
1035:  jmp 0x103a
```

Se trata de un bucle, en 1030 existe un salto condicional a 1019, y de hecho hay un registro al que se le suma 4. En concreto este bucle hace una decodificación usando `xor`. El registro `r_00` es un puntero hacia el código cifrado, que se copia a `r_03` y que más tarde es incrementado; `r_01` es la última dirección que tiene código cifrado. De hecho `r_05` tiene un valor de 32 bits que se usa para hacer el `xor` con los valores cifrados que se leen de la memoria en 1019, y que después son escritos en la memoria de nuevo después de hacerles `xor` usando `r_04`, en la dirección 1028.

Genial, el viejo truco de un cifrado `xor` justo al inicio de un fragmento de código. Cuando todo el bucle termina se salta al inicio del código que acaba de ser descifrado, justo a 103a. La primera instrucción en 103a es un `jmp`, (`jmp 0x1bc0`) que salta a 0x1bc0.

Esto es lo que hay a partir de 0x1bc0:

```
1bc0:  push r_09
1bc3:  push r_0a
1bc6:  push r_0b
1bc9:  call 0x1a6d
1bce:  mov r_01,0x001e
1bd5:  mov r_00,0x0004
1bdc:  call 0x1080
1be1:  mov r_0b,r_00
1be5:  jmp 0x1bef
```

Parece bastante familiar. Unos cuantos `push` al principio, y algún `call`. A decir verdad en todo el código apenas se usan unos cuantos registros, pero es importante resaltar que `r_00`, `r_01`, y `r_1e` constantemente son usados como registros temporales para otras operaciones.

Veamos lo que hay en la dirección que llama el primer `call`, a 1a6d:

```
1a6d:  mov r_00,0x0050
1a74:  call 0x103f
1a79:  mov r_00,0x006c
1a80:  call 0x103f
1a85:  mov r_00,0x0065
1a8c:  call 0x103f
1a91:  mov r_00,0x0061
1a98:  call 0x103f
1a9d:  mov r_00,0x0073
1aa4:  call 0x103f
1aa9:  mov r_00,0x0065
1ab0:  call 0x103f
1ab5:  mov r_00,0x0020
1abc:  call 0x103f
1ac1:  mov r_00,0x0065
[...]
```

```
1ba5:  mov r_00,0x003a
1bac:  call 0x103f
1bb1:  mov r_00,0x0020
```

```
1bb8: call 0x103f
1bbd: ret
```

Bueno, se trata de un fragmento con apenas dos instrucciones repetidas una y otra vez, y que finalmente termina, como es lógico, con la instrucción `ret`. Si nos fijamos bien, todos los valores que se pasan a `r_00`, antes de la llamada a `103f`, son caracteres ASCII dentro del rango alfabético. De hecho estos valores conforman la cadena: "Please enter your password: ". Sin detenernos demasiado en lo que haya en `103f` podríamos averiguar que es una función que imprimirá ese carácter. De hecho en esa dirección hay una `ncall`, y un `ret`.

```
103f: ncall 0x00 ;putchar
1041: ret
```

En el análisis de este *opcode*, hemos identificado y puesto nombre a cada una de las funciones nativas que en la tabla interna tiene programada la VM. En concreto la función `0x00` de `ncall` es `Putchar`. Es muy curioso ver cómo en el código virtual hay como una pequeña tabla de importación, donde se hace referencia a las 17 funciones nativas implementadas.

```
1042: ncall 0x01
1044: ret
1045: ncall 0x02
1047: ret
1048: ncall 0x03
104a: ret
104b: ncall 0x04
104d: ret
104e: ncall 0x05
1050: ret
[...]
```

He aquí la tabla de todas las funciones implementadas por medio de `ncall` en esta VM:

N° <i>ncall</i>	Nombre	Operación	Tipo
0x00	Putchar	putchar(r_00)	Salida
0x01	PrintInt	fprintf ("%d", r_00)	Salida
0x02	PrintHex	fprintf ("%x", r_00)	Salida
0x03	PrintFloat	fprintf ("%f", r_00)	Salida
0x04	Getchar	r_00 = getchar()	Entrada
0x05	ScanInt	r_00 = fscanf ("%d")	Entrada
0x06	ScanHex	r_00 = fscanf ("%x")	Entrada
0x07	ScanFloat	r_00 = fscanf ("%f")	Entrada
0x08	Eof	r_00 = (getchar == EOF ? 1 : 0)	Entrada

0x09	Int2Float	r_00 (Bin32) → r_00 (IEEE 754)	Fpu
0x0a	Float2Int	r_00 (IEEE 754) → r_00 (Bin32)	Fpu
0x0b	FAdd	r_00 = r_00 + r_01	Fpu
0x0c	FSub	r_00 = r_00 - r_01	Fpu
0x0d	FMul	r_00 = r_00 * r_01	Fpu
0x0e	FDiv	r_00 = r_00 / r_01	Fpu
0x0f	NNop	r_00 = r_00	Miscelánea
0x10	IsNotZero	R_00 = (r_00 != 0 ? 1 : 0)	Miscelánea
0x11	SwapAndAdd	R_00 ↔ RamBaseFree : RamBaseFree += r_00	Miscelánea

Una vez se imprime la cadena podemos imaginar que de algún modo se debe pedir y leer la contraseña del usuario. Sigamos por donde lo habíamos dejado antes...

```

1bc6:  push r_0b
1bc9:  call 0x1a6d ;Prints "Please enter your password: "
1bce:  mov r_01,0x001e
1bd5:  mov r_00,0x0004
1bdc:  call 0x1080
1be1:  mov r_0b,r_00
1be5:  jmp 0x1bef

```

Se realiza una nueva llamada, en esta ocasión a 1080, usando dos parámetros que se pasan en r_00 y r_01.

```

1080:  push r_1e
1083:  push r_1d
1086:  mov r_1e,r_01
108a:  r_00 = r_1e * r_00
108f:  r_00=r_00 +0008
1097:  mov r_1d,r_00
109b:  ncall 0x11
10a1:  jz 0x10c2
10a6:  mov r_04,r_1d
10aa:  r_04 = r_04 >> 0003
10b5:  mov [r_00],r_04
10b5:  r_00=r_00 +0008
10bd:  pop r_1d
10bf:  pop r_1e
10c1:  ret
10c2:  endvm

```

Este fragmento inicialmente fue un poco duro a la hora de analizarlo sin ver su ejecución en la VM. La `ncall 0x11` es una función nativa que hemos llamado “SwapAndAdd”. De hecho usa una variable inicial que vale 0x10000. Esta variable se intercambia con r_00, y después le añade r_00 al nuevo valor de la variable. Por tanto, cada vez que se ejecuta `ncall 0x11` la variable se incrementa con el valor de r_00. Por ejemplo, si la variable vale 0x1010 y r_00 es 0x30, `ncall 0x11` hará que r_00 sea 0x1010 y la variable pase a valer 0x1040. Si volvemos a hacer `ncall 0x11` y r_00 es 0x60, hará que r_00 sea 0x1040 y la variable sea 0x10a0.

El `jz` no va a ocurrir nunca ya que `r_00` y `r_01` no son cero (al menos en este momento, si por alguna razón fallara, se salta a `10c2`, donde `endvm` cerraría la ejecución de la máquina virtual). Este código en concreto es algo así como un cálculo del espacio necesario, escribiendo un tamaño calculado. Vamos a pensar en esta subrutina como un “alojamiento de memoria”, de hecho la llamaremos “AllocateMemory”. Los argumentos pasados serán el número de elementos a reservar, y el tamaño de cada uno de esos elementos. Lo mas importante a tener en cuenta en esta función es que `r_00` adquiere un nuevo valor (sea el que sea), antes de volver.

De hecho, una vez termina la llamada a esta función, el código sigue en `1be1`, donde el registro `r_0b` se carga con el valor de `r_00` (justo el valor devuelto por `1080`). Atención aquí, porque `r_0b` va a ser importante más tarde. Después continúa la ejecución con un salto a `1bef`.

```
1bef: mov r_09,0x0000
1bf6: jmp 0x1d66
1bfb: mov r_1e,r_09
1bff: r_1e = r_1e * 0004
1c07: mov r_00,r_0b
1c0b: r_1e=r_1e + r_00
1c10: mov r_0a,r_1e
1c14: call 0x104b ; GetChar
1c19: mov r_01,r_00
1c20: mov [r_0a],r_01
1c20: mov r_01,[r_0a]
1c23: mov r_1e,r_01
1c27: mov r_00,0x000a
1c2e: cmp r_1e,r_00
1c32: jz 0x1c3c ; jumps to print sorry..
1c37: jmp 0x1d5e
1c3c: mov r_00,0x0053 ; "Sorry, wrong password!\n"
1c43: call 0x103f
1c48: mov r_00,0x006f
1c4f: call 0x103f
[...]
1d4b: call 0x103f
1d50: mov r_00,0x0000
1d57: pop r_0b
1d59: pop r_0a
1d5b: pop r_09
1d5d: ret
1d5e: r_09=r_09 +0001
1d66: mov r_1e,r_09
1d6a: mov r_00,0x001e
1d71: cmp r_1e,r_00
1d75: js 0x1bfb
```

Ante habíamos identificado una especie de tabla de importación con todos los `ncall`. Por eso sabemos que la llama a `104b` realmente es `ncall 0x04`, que en concreto es `getchar`. Desde `1c3c` hasta `1d5d` también es un fragmento que empieza a ser conocido, ya que es cómo se imprimía antes una cadena (en esta ocasión corresponde a la cadena “Sorry, wrong password!\n”); por eso lo he puesto como

comentario en el código. Todo lo demás es un bucle de un *while*. El registro `r_09` se inicializa a 0, y salta a `1d66`, que es donde está la comparación del bucle. Hay que tener cuidado de no confundir el registro `r_1e` con el valor `0x001e` :) La condición hace uso de `r_1e` y `r_00` como registros temporales para hacer el chequeo de la condición, pero realmente el chequeo consiste en ver que `r_09` sea menor que `0x001E`. Si se da la condición salta al cuerpo del bucle *while* (en `1bfb`).

La parte principal del bucle es sencilla: se llama a `getchar` usando la tabla de importación, y se guarda el valor devuelto en una zona de memoria apuntada por el registro `r_0a`. Este registro se calcula usando `r_09` y `r_0b` (usando de nuevo `r_1e` y `r_00` como registros temporales).

Tenemos por tanto que la posición donde se guarda el valor devuelto por `getchar` es: `r_0b+(r_09*4)`. Esta multiplicación por 4 es necesaria porque los caracteres se están guardando como valores de 32 bits. Con esta información podemos dar por hecho que `r_0b` es el puntero base donde se almacena la cadena que se introduce, y que `r_09` es un contador usado como índice. Es más, el registro `r_09` se incrementa en `1d5e`, justo antes de llegar a la condición del bucle *while*. Recuerda que `r_0b` se calculó anteriormente, con la función que hay en `1080`, y que de hecho habíamos bautizado como “AllocateMemory”.

Dentro del bucle, se hace un chequeo en `1c27`, que verifica que el valor devuelto por `getchar` no sea `0x0a` (“\n”). Así, si la contraseña introducida tiene una longitud menor de `0x1e` (30) aparecerá el mensaje de “Sorry, wrong password!\n”.

Cuando se termina el bucle del *while* el registro `r_0b` conserva la dirección inicial (*puntero base*) donde se guarda la contraseña introducida. Justo después de la salida del *while* nos encontramos esto:

```
1d7a: call 0x104b ; GetChar
1d7f: mov r_01,r_00
1d83: mov r_1e,r_01
1d87: mov r_00,0x000a
1d8e: cmp r_1e,r_00
1d92: jnz 0x1d9c ; prints sorry
1d97: jmp 0x1ebe
1d9c: mov r_00,0x0053 ; "Sorry, wrong password!\n"
1da3: call 0x103f
[...]
1ebd: ret
1ebe: mov r_00,r_0b
1ec2: call 0x12a9
1ec7: mov r_00,0x0000
1ece: pop r_0b
1ed0: pop r_0a
1ed2: pop r_09
1ed4: ret
```


Otra vez la función `getchar`; en esta ocasión se mira que el usuario introduzca el carácter `0x0a` (“\n”), pero no se guarda en ningún lado. Lo que hay en `1d9c`, donde se salta si el carácter no es `0x0a`, es de nuevo código para imprimir una cadena. Sí, de nuevo aparece la cadena “Sorry, wrong password!\n”. Todo esto es para verificar que la longitud de la cadena es justo de `0x1e` caracteres. Ni uno más, ni uno menos.

Cuando se pasan todos los chequeos para verificar la longitud de la contraseña, se salta en `1ebe`. Aquí `r_00` se carga con el valor que tiene `r_0b` (recuerda que este registro conserva el puntero de la contraseña tecleada), y se llama a `12a9`. Vamods, que a esa función se le pasa como argumento un puntero a la contraseña. Vaya vaya, esto tiene toda la pinta de ser la función *CheckPassword* :)

. Cómo llegar a odiar los registros temporales .

El inicio de la función, que ya hemos supuesto como *CheckPassword*, nos va a resultar muy familiar a estas alturas del análisis:

```
12a9: push r_09
12ac: push r_0a
12af: mov r_0a, r_00
12b3: mov r_01, 0x001e
12ba: mov r_00, 0x0004
12c1: call 0x1080 ; AllocateMem
12c6: mov r_09, r_00
12ca: mov r_01, 0x0004
12d1: mov r_00, 0x0004
12d8: call 0x1080 ; AllocateMem
12dd: mov r_05, r_00
```

Se guardan los registros `r_09` y `r_0a`, haciendo uso de la pila, y se carga `r_0a` con el argumento (en `r_00`) que se le ha pasado a la función. Por tato desde ahora `r_0a` es el puntero a la contraseña introducida. Anda, aquí aparece de nuevo “AllocateMem”, llamada en dos ocasiones, y con argumentos distintos cada vez. En `12c1` se le pasan `1e` y `04`, así que ya podemos asumir que está reservando `1e dwords`. La dirección a ese espacio reservado es devuelta en `r_00` y pasada a `r_09`. La segunda llamada se hace con `04` y `04` como argumentos. La dirección en esta ocasión es guardada en `r_05`. En seguida, de hecho, empieza a hacer uso del valor de `r_05`.

```
12e1: mov r_1e, 0x00fd
12eb: mov [r_05], r_1e
12eb: mov r_1e, 0x0001
12f2: r_1e = r_1e * 0004
12fa: mov r_00, r_05
12fe: r_1e = r_1e + r_00
1303: mov r_01, r_1e
1307: mov r_1e, 0x000e
1311: mov [r_01], r_1e
1311: mov r_1e, 0x0001
1318: r_1e = r_1e * 0008
```

```
1320:  mov r_00,r_05
1324:  r_1e=r_1e + r_00
1329:  mov r_01,r_1e
132d:  mov r_1e,0x0063
1337:  mov [r_01],r_1e
1337:  mov r_1e,0x0001
133e:  r_1e = r_1e * 000c
1346:  mov r_00,r_05
134a:  r_1e=r_1e + r_00
134f:  mov r_01,r_1e
1353:  mov r_1e,0x004f
135d:  mov [r_01],r_1e
```

En este fragmento se hace uso constantemente de `r_1e` como almacenamiento temporal. Para no equivocarnos, veamos lo que hace realmente. Primero se guarda `0xfd` en `[r_05]`. Después carga `r_1e` con `1`, y haciendo uso de `r_00` y `r_01` calcula: $(1*4)+r_05$. Esta nueva dirección de memoria la pasa a `r_01`, y usando `r_1e` como almacenamiento temporal, guarda el valor `0x0e` en la dirección calculada en `r_01`.

Todas las demás instrucciones que vienen después son muy parecidas a esto. Se usan `r_1e`, `r_01`, y `r_00` como registros temporales para hacer multiplicaciones similares, solo que en las otras ocasiones va a multiplicar el `1` por `08`, y por `0c`. Aquí es importante darse cuenta que siempre se usa el valor de `r_05` como un valor fijo que se suma.

Fácil de entender o_O El registro `r_05` es el puntero a un vector (*array*) de 4 elementos, de 32 bits cada uno; y la multiplicación realmente está calculando la dirección de memoria (*offset*) donde va cada uno de los elementos del vector. Realmente todo este código no hace otra cosa sino inicializar este vector con estos valores: `[fd, 0e, 63, 4f]`. Llamaremos a este vector “fourarray”.

Después de esta rallada, vienen un montón de instrucciones muy parecidas a esto que acabamos de analizar. Esta vez, en lugar de usar `r_05`, se usa `r_09` como valor fijo para la suma. Esto empieza a tener sentido, porque al principio de todo, se llamó dos veces a `AllocateMem`. La primera vez el valor devuelto por esta función en `r_00` se guardó en `r_09`. Además ahora sabemos que la función fue llamada con los parámetros `1e` y `04`. Veamos el aspecto real en el código:

```
135d:  mov r_1e,0x008d
1367:  mov [r_09],r_1e
1367:  mov r_1e,0x0001
136e:  r_1e = r_1e * 0004
1376:  mov r_00,r_09
137a:  r_1e=r_1e + r_00
137f:  mov r_01,r_1e
1383:  mov r_1e,0x006f
138d:  mov [r_01],r_1e
138d:  mov r_1e,0x0001
1394:  r_1e = r_1e * 0008
139c:  mov r_00,r_09
```

```
13a0:  r_1e=r_1e + r_00
13a5:  mov  r_01,r_1e
13a9:  mov  r_1e,0x0000
13b3:  mov  [r_01],r_1e
13b3:  mov  r_1e,0x0001
13ba:  r_1e = r_1e * 000c
13c2:  mov  r_00,r_09
13c6:  r_1e=r_1e + r_00
13cb:  mov  r_01,r_1e
13cf:  mov  r_1e,0x0024
13d9:  mov  [r_01],r_1e
13d9:  mov  r_1e,0x0001
13e0:  r_1e = r_1e * 0010
13e8:  mov  r_00,r_09
13ec:  r_1e=r_1e + r_00
13f1:  mov  r_01,r_1e
13f5:  mov  r_1e,0x0098
13ff:  mov  [r_01],r_1e
[...]
```

```
178f:  mov  r_1e,0x0001
1796:  r_1e = r_1e * 0074
179e:  mov  r_00,r_09
17a2:  r_1e=r_1e + r_00
17a7:  mov  r_01,r_1e
17ab:  mov  r_1e,0x0077
17b5:  mov  [r_01],r_1e
```

Sí, muy laaaargo, pero muy predecible (de hecho he omitido una parte central por ser repetitiva). Otra vez se inicializa un vector. En esta ocasión podemos imaginar que será de tamaño 1e elementos. Y por eso se multiplica por 4, 8, 0c, 10, 14, y así, de 4 en 4, hasta 74 ($(1e-1)*4=74$). Como ocurría antes, los valores se cargan en la memoria del vector, pero pasan antes por el registro r_1e. A este vector lo llamaremos “longarray”, y estos son los valores con los que se inicializa:

```
[8d, 6f, 00, 24, 98, 7c, 10, 10, 9c, 60, 07, 10, 8b, 63, 10,
10, 9c, 60, 07, 10, 85, 61, 11, 3c, a2, 61, 0b, 10, 90, 77]
```

En este momento tenemos tres registros con información importante: r_0a que apunta a la contraseña introducida; r_05 que apunta a un vector de 4 elementos; y r_09 que apunta a un vector de 1e elementos.

Las siguientes instrucciones ponen a cero tres registros, antes de empezar un bucle, que va a ser muy parecido al *while* que hemos analizado antes:

```
17b5:  mov  r_03,0x0000
17bc:  jmp  0x17c6
17c1:  jmp  0x1878
17c6:  mov  r_01,0x0000
17cd:  mov  r_02,0x0000
17d4:  jmp  0x1860
```

Podríamos asumir que los registros que se han puesto a cero vayan a ser usados como contadores o índices. En concreto son r_03, r_01, y r_02. Veamos 1860 y todo el bloque de instrucciones anteriores...

```
17d9:  mov r_1e,r_02
17dd:  r_1e = r_1e * 0004
17e5:  mov r_00,r_0a
17e9:  r_1e=r_1e + r_00
17ee:  mov r_03,r_1e
17f2:  mov r_04,[r_03]
17f5:  r_00 = r_02 / 0004
17f5:  r_03 = r_02 mod 0004
17fe:  mov r_1e,r_03
1802:  r_1e = r_1e * 0004
180a:  mov r_00,r_05
180e:  r_1e=r_1e + r_00
1813:  mov r_03,r_1e
1817:  mov r_03,[r_03]
181a:  r_04=r_04 ^ r_03
181f:  mov r_1e,r_02
1823:  r_1e = r_1e * 0004
182b:  mov r_00,r_09
182f:  r_1e=r_1e + r_00
1834:  mov r_03,r_1e
1838:  mov r_03,[r_03]
183b:  mov r_1e,r_04
183f:  mov r_00,r_03
1843:  cmp r_1e,r_00
1847:  jnz 0x1851
184c:  jmp 0x1858
1851:  mov r_01,0x0001
1858:  r_02=r_02 +0001
1860:  mov r_03,r_01
1864:  mov r_1e,r_02
1868:  mov r_00,0x001e
186f:  cmp r_1e,r_00
1873:  js 0x17d9
```

Desde 1860 hasta 1873 está la condición del *while*. El registro `r_02` se carga en `r_1e`, y se carga con un valor fijo de `1e`, después son comparados en 186f (sí, de nuevo `r_00` y `r_1e` se han usado como registros temporales).

Siempre que `r_02` sea menor de `1e` la condición será verdadera y el código saltará a 17d9. En 17d9 de nuevo `r_1e` se usa para hacer una serie de cálculos. Primero `r_02` se multiplica por 4, y después es sumado a `r_0a` (pasándolo por `r_00`); finalmente el resultado es cargado en `r_03`. En resumen cuando llegamos a 17ee tenemos: `r_03=r_0a+(r_02*4)`.

Sabemos que `r_0a` es el puntero al inicio de la contraseña introducida, así que podríamos asumir que `r_02` es el índice del carácter que queremos leer de la contraseña. En 17f2 el carácter es leído de la dirección desde la dirección de memoria calculada en `r_03`, cargando el valor de ese carácter en el registro `r_04`. Por tanto: `r_04=password[r_02]`.

El *opcode* que hay en 17f5 es especial, tal y como ha explicado ya SkUaTeR. La división tiene dos destinos: un registro donde se almacena el cociente de la operación, y otro registro donde se almacena el módulo. En este caso concreto nos interesa solamente el módulo, que se carga en `r_03`. Por tanto: `r_03=r_02 mod 4`.

Las instrucciones de 17fe a 1813 usan como registros temporales `r_00` y `r_1e`. De nuevo podemos resumir las operaciones en algo mucho más sencillo de entender: `r_03=r_05+((r_02 mod 4)*4)`.

Sabemos que `r_05` es el puntero al inicio del vector que hemos llamado “fourarray”, y que el módulo se ha hecho con el registro que se está usando como índice para leer caracteres de la contraseña, luego podemos intuir que es el modo de obtener valores consecutivos y de modo cíclico desde 0 hasta 3 (los índices válidos para el vector “fourarray”). Pensemos que el índice va desde 0 hasta 0x1e, si calculamos el módulo 4 de cada valor del índice obtenemos: 0,1,2,3,0,1,2,3,0,1... Finalmente, y como viene siendo habitual, es multiplicado por 4, ya que cada elemento en el vector “fourarray” tiene 32 bits.

Esto empieza a ser sencillo de entender, ya que `r_05` es el puntero del vector, y por tanto todo este cálculo sirve para obtener la dirección del elemento indicado por el índice que hay en `r_02`. De hecho la siguiente instrucción carga en `r_03` esa dirección de memoria, y carga desde esa dirección el valor del elemento (también en `r_03`). En este momento tenemos `r_04=password[r_02]` y `r_03=fourarray[r_02 mod 4]`.

Aquí llega el momento cumbre de todo el chequeo, la instrucción en 181a hace un *xor*: `r_04=r_04^r_03`, que podríamos substituir con los valores de los cálculos anteriores, obteniendo:

```
r_04 = password[r_02] ^ fourarray[r_02 mod 4]
```

Desde 181f hasta 1838 se hace de nuevo un cálculo para conseguir la dirección de memoria de un elemento concreto en un vector. Como son partes muy repetitivas de código ya visto, es más fácil darse cuenta de qué está haciendo. Esta vez usa como puntero base el registro `r_09`, que es el puntero a “longarray”. Y de nuevo se usa como índice `r_02`, que multiplicándolo por 4 saca la dirección concreta del elemento indicado por `r_02`. Finalmente se carga en `r_03` el valor del elemento del vector. Tendríamos por tanto: `r_03=longarray[r_02]`

Desde 183b hasta 1847 se compara `r_03` con `r_04`, y sí, de nuevo se hace usando `r_00` y `r_1e` como registros temporales (seguro que a estas alturas lo odias tanto como yo). Si `r_03` y `r_04` no son iguales, entonces se salta a 1051 donde `r_01` se pone a 1. Si son iguales, se salta a 1851, donde el registro `r_02` (el índice) se incrementa y el bucle vuelve a empezar.

Bueno, esto parece ya bien claro de entender. `r_01` va a servir de marca, que inicialmente está a 0, de tal modo que valdrá 1 si el chequeo falla, y 0 si se han pasado todos los chequeos dentro del bucle. Hay que fijarse bien, porque justo antes de la

condición del bucle, `r_01` se copia en `r_03`. El chequeo que se realiza es fácil (usando `i` como índice):

```
password[i] ^ fourarray[i mod 4] == longarray[i]
```

Al final hay un salto condicional en 187c. Esta condición se hace sobre el valor `r_03`, usando la instrucción lógica AND (no confundir con la operación binaria); de este modo se imprimirá “Sorry, wrong password!\n” o “Congratulations!\n”.

```
1878: r_03 and r_03
187c: jnz 0x1952
1881: mov r_00,0x0043 ; "C"
1888: call 0x103f
188d: mov r_00,0x006f ; "o"
1894: call 0x103f
1899: mov r_00,0x006e ; "n"
18a0: call 0x103f
18a5: mov r_00,0x0067 ; "g"
18ac: call 0x103f
18b1: mov r_00,0x0072 ; "r"
18b8: call 0x103f
18bd: mov r_00,0x0061 ; "a"
[...]
```

Visto lo visto, conseguir la contraseña válida va a ser ya coser y cantar. Tan solo tendremos que usar “fourarray” como contraseña (tendremos que repetirla una y otra vez) para hacer *xor* a “longarray”. Quedaría de este modo: 8d^fd, 6f^0e, 00^63, 24^4f, 98^fd, 7c^0e, 10^63, ...

De este modo obtenemos la tan preciada y buscada contraseña:

```
packers_and_vms_and_xors_oh_my
```

[thEpOpE](#)
crAck & prAy

Saludacos a toda la gentuza de amn3sla_team

Todos nuestros respetos para w0pr, w3b0n3s, int3pids, Insanity, ReSecurity, The DHARMA Initiative, y a los españoles de dcua y fail0verflow

Síguenos en [@amn3sla_team](#) | www.amn3sla.com

```
C:\> forget what we made _
```

• [Data] •

.decoder.c.

```
// Baleful VM Decoder by SkUaTeR
//
//   compile: i586-mingw32msvc-gcc -mconsole decoder.c -odecoder.exe
//
// Need a vm.code at same directory

#include <stdio.h>
#include <windows.h>

// Defines for r2 compatibility
#define ut8 char
#define ut32 unsigned int
#define r_strbuf_setf sprintf

int ae_load_file_to_memory(const char *filename, char **result) {
    int size = 0;
    FILE *f = fopen(filename, "rb");
    if (f == NULL) {
        *result = NULL;
        return -1;
    }
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    fseek(f, 0, SEEK_SET);
    *result = (char *)malloc(size+1);
    if (size != fread(*result, sizeof(char), size, f)) {
        free(*result);
        return -2;
    }
    fclose(f);
    (*result)[size] = 0;
    return size;
}

int anal_baleful_getregs(const ut8 *buf, ut8 * b, char * oper, int type) {
    const ut8 * c;
    const ut8 *r0;
    const ut8 *r1;
    const ut8 *r2;
    const ut8 *r3;
    const ut32 *imm;
    const ut32 *imm1;

    int size=0;
    c = buf+1;
    switch(type) {
    case 0: // 8 8 11 5
        r0 = buf + 2;
        switch(*c) {
        case 1:
            r1 = buf + 3;
            imm = buf + 4;
            r_strbuf_setf(b, "r_%02x = r_%02x %s 0x%04x", *r0, *r1, oper, *imm);
            size=8;
            break;
        case 2:
            imm = buf + 3;
            r1 = buf + 4;
            r_strbuf_setf(b, "r_%02x = 0x%04x %s r_%02x", *r0, *imm, oper, *r1);
            size=8;
            break;
        case 4:
            imm = buf + 3;

```

```
        imm1 = buf + 7;
        r_strbuf_setf(b, "r_%02x = 0x%04x %s 0x%04x", *r0, *imm, oper, *imm1);
        size=11;
        break;
    case 0:
        r1 = buf + 3;
        r2 = buf + 4;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=5;
        break;
    default:
        r1 = buf + 3;
        r2 = buf + 4;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=5;
        break;
    }
    break;

case 1: // 9 9 12 6
    r0 = buf + 2;
    r3 = buf + 3; // address to save reminder
    switch(*c) {
    case 1:
        r1 = buf + 4;
        imm = buf + 5;
        r_strbuf_setf(b, "r_%02x = r_%02x %s 0x%04x", *r0, *r1, oper, *imm);
        size=9;
        break;
    case 2:
        r1 = buf + 5;
        r_strbuf_setf(b, "r_%02x = 0x%04x %s r_%02x", *r0, *imm, oper, *r1);
        size=9;
        break;
    case 4:
        imm = buf + 4;
        imm1 = buf + 8;
        r_strbuf_setf(b, "r_%02x = 0x%04x %s 0x%04x", *r0, *imm, oper, *imm1);
        size=12;
        break;
    case 0:
        r1 = buf + 4;
        r2 = buf + 5;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=6;
        break;
    default:
        r1 = buf + 4;
        r2 = buf + 5;
        r_strbuf_setf(b, "r_%02x = r_%02x %s r_%02x", *r0, *r1, oper, *r2);
        size=6;
        break;
    }
    break;

case 2: // 7 7 10 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        r_strbuf_setf(b, "r_%02x %s 0x%04x", *r1, oper, *imm);
        size=7;
        break;
    case 2:
        imm = buf + 2;
        r1 = buf + 6;
        r_strbuf_setf(b, "0x%04x %s r_%02x", *imm, oper, *r1);
        size=7;
        break;
    case 4:
        imm = buf + 2;
        imm1 = buf + 6;
```



```
        r_strbuf_setf(b, "0x%04x %s 0x%04x",*imm,oper,*imm1);
        size=10;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "r_%02x %s r_%02x",*r1,oper,*r2);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "r_%02x %s r_%02x",*r1,oper,*r2);
        size=4;
        break;
    }
    break;

case 3:// 7 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        r_strbuf_setf(b, "%s r_%02x,0x%04x",oper,*r1,*imm);
        size=7;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "%s r_%02x,r_%02x",oper,*r1,*r2);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        r_strbuf_setf(b, "%s r_%02x,r_%02x",oper,*r1,*r2);
        size=4;
        break;
    }
    break;

case 4: // 6 3
    switch(*c) {
    case 1:
        imm = buf + 2;
        r_strbuf_setf(b, "%s 0x%04x",oper,*imm);
        size=6;
        break;
    case 0:
        r0 = buf + 2;
        r_strbuf_setf(b, "%s r_%02x",oper,*r0);
        size=3;
        break;
    default:
        r0 = buf + 2;
        r_strbuf_setf(b, "%s r_%02x",oper,*r0);
        size=3;
        break;
    }
    break;

case 5: //5
    imm = buf + 1;
    r_strbuf_setf(b, "%s 0x%04x",oper,*imm);
    size=5;
    break;

case 6://2
    r0 = buf + 1;
    r_strbuf_setf(b, "%s r_%02x",oper,*r0);
    size=2;
    break;

break;
```

```
    }
    return size;
}

int main(void)
{
    int size;
    int reip;
    int vIP;
    int tmp;
    char *vmMemoryBase=0;
    int buf;
    char salida[1024];
    const ut8 *r = 0;
    const ut8 *r0 = 0;
    const ut8 *r1 = 0;
    const ut8 *p = 0;
    const ut32 *imm = 0;
    const ut32 *imm1 = 0;
    // Set virtual ip to 0x1000, we have a full context and instruction
    // start at this offset into the context.
    vIP = 0x1000u;
    tmp=vIP;
    size = ae_load_file_to_memory("vm.code", &vmMemoryBase);
    while(vIP<size) {
        buf=&vmMemoryBase[vIP];
        tmp=vIP;
        switch (vmMemoryBase[vIP]) {
            case 2: // 8 8 11 5
                vIP+=anal_baleful_getregs(buf,&salida,"+",0);
                break;
            case 3: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"-",0);
                break;
            case 4: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"*",0);
                break;
            case 6: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"^",0);
                break;
            case 9: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"%",0);
                break;
            case 10: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"|",0);
                break;
            case 12: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,"<<",0);
                break;
            case 13: // 8 8 11 5
                vIP+= anal_baleful_getregs(buf,&salida,">>",0);
                break;
            case 5: // 9 9 12 6
                vIP+= anal_baleful_getregs(buf,&salida,"/",1);
                break;
            case 22: // 7 7 10 4
                vIP+= anal_baleful_getregs(buf,&salida,"and",2);
                break;
            case 23: // 7 7 10 4
                vIP+= anal_baleful_getregs(buf,&salida,"cmp",2);
                break;
            case 24: //7 4

                vIP+= anal_baleful_getregs(buf,&salida,"mov",3);
                break;
            case 30: //6 3
                p = buf + 1;
                vIP+= anal_baleful_getregs(buf,&salida,"push",4);
                break;
            case 15: //5
                imm = buf + 1;
                vIP+= anal_baleful_getregs(buf,&salida,"call",5);
```

```
        break;
case 14: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jmp",5);
    break;
case 16: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jz",5);
    break;
case 17: //5
    vIP+= anal_baleful_getregs(buf,&salida,"js",5);
    break;
case 18: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jbe",5);
    break;
case 19: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jg",5);
    break;
case 20: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jns",5);
    break;
case 21: //5
    vIP+= anal_baleful_getregs(buf,&salida,"jnz",5);
    break;

case 27:
    r = buf + 1;
    r1 = buf + 2;
    vIP+= 3;
    r_strbuf_setf (&salida, "mov r_%02x,[r_%02x]",*r,*r1);
    break;
case 28://0x1c
    r = buf + 1;
    r1 = buf + 2;
    vIP+= 3;
    r_strbuf_setf (&salida, "mov [r_%02x],r_%02x",*r,*r1);
    break;
case 11:
    r_strbuf_setf (&salida, "regX = regY==0");
    vIP+= 3;
    break;
case 7:
    r_strbuf_setf (&salida, "regX = NEG regY");
    vIP+= 3;
    break;
case 8:
    r_strbuf_setf (&salida, "regX = NOT regY");
    vIP+= 3;
    break;
case 25:
    vIP+= anal_baleful_getregs(buf,&salida,"++",6);
    break;
case 26:
    r = buf + 1;
    vIP+= anal_baleful_getregs(buf,&salida,"--",6);
    break;
case 31:
    vIP+= anal_baleful_getregs(buf,&salida,"pop",6);
    break;
case 32:
    p = buf + 1;
    vIP+= 2;
    if (*p==0)
        r_strbuf_setf (&salida, "apicall: putchar()");
    else
        r_strbuf_setf (&salida, "apicall: %02x",*p);
    break;
case 1:
    vIP+= 1;
    r_strbuf_setf (&salida, "ret");
    break;
case 0:
    vIP+= 1;
    r_strbuf_setf (&salida, "nop");
    break;
```

```

        case 29:
            vIP+= 1;
            r_strbuf_setf (&salida, "end virtual");
            break;

        default:
            vIP+= 1;
            r_strbuf_setf (&salida, "nop");
            break;
    }
    printf("%08x: %s    (size = %i)\n",tmp,salida,vIP-tmp);
    getchar();
};

return 0;
}

```

. código plug-in para radare2.

```

/* radare - LGPL - Copyright 2009-2014 - pancake, nibble */
/* baleful plugin by SkUaTeR */

#include <stdio.h>
#include <string.h>
#include <r_types.h>
#include <r_lib.h>
#include <r_asm.h>
int asm_baleful_getregs(const ut8 *buf,ut8 * b,char * oper,int type) {
    const ut8 * c;
    const ut8 *r0;
    const ut8 *r1;
    const ut8 *r2;
    const ut8 *r3;
    const ut32 *imm;
    const ut32 *imm1;

    int size=0;
    c = buf+1;
    switch(type) {
    case 0: // 8 8 11 5
        r0 = buf + 2;
        switch(*c) {
        case 1:
            r1 = buf + 3;
            imm = buf + 4;
            snprintf(b, 64, "r_%02x = r_%02x %s 0x%04x",*r0,*r1,oper,*imm);
            //snprintf(b, 64, "%s",oper);
            size=8;
            break;

        case 2:
            imm = buf + 3;
            r1 = buf + 4;
            snprintf(b, 64, "r_%02x = 0x%04x %s r_%02x",*r0,*imm,oper,*r1);

            //snprintf(b, 64, "%s",oper);
            size=8;
            break;

        case 4:
            imm = buf + 3;
            imm1 = buf + 7;
            snprintf(b, 64, "r_%02x = 0x%04x %s 0x%04x",*r0,*imm,oper,*imm1);
            //snprintf(b, 64, "%s",oper);
            size=11;
            break;

        case 0:

```

```
        r1 = buf + 3;
        r2 = buf + 4;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=5;
        break;
default:
        r1 = buf + 3;
        r2 = buf + 4;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);

        //snprintf(b, 64, "%s",oper);
        size=5;
        break;
    }
    break;
case 1: // 9 9 12 6
    r0 = buf + 2;
    r3 = buf + 3; // guarda aki el resto
    switch(*c) {
    case 1:
        r1 = buf + 4;
        imm = buf + 5;
        snprintf(b, 64, "r_%02x = r_%02x %s 0x%04x",*r0,*r1,oper,*imm);
        //snprintf(b, 64, "%s",oper);
        size=9;
        break;
    case 2:
        r1 = buf + 5;
        snprintf(b, 64, "r_%02x = 0x%04x %s r_%02x",*r0,*imm,oper,*r1);

        //snprintf(b, 64, "%s",oper);
        size=9;
        break;
    case 4:
        imm = buf + 4;
        imm1 = buf + 8;
        snprintf(b, 64, "r_%02x = 0x%04x %s 0x%04x",*r0,*imm,oper,*imm1);
        //snprintf(b, 64, "%s",oper);
        size=12;
        break;
    case 0:
        r1 = buf + 4;
        r2 = buf + 5;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=6;
        break;
    default:
        r1 = buf + 4;
        r2 = buf + 5;
        snprintf(b, 64, "r_%02x = r_%02x %s r_%02x",*r0,*r1,oper,*r2);

        //snprintf(b, 64, "%s",oper);
        size=6;
        break;
    }
    break;
case 2: // 7 7 10 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        snprintf(b, 64, "r_%02x %s 0x%04x",*r1,oper,*imm);
        //snprintf(b, 64, "%s",oper);
        size=7;
        break;
    case 2:
        imm = buf + 2;
        r1 = buf + 6;
        snprintf(b, 64, "0x%04x %s r_%02x",*imm,oper,*r1);
        //snprintf(b, 64, "%s",oper);
```

```
        size=7;
        break;
    case 4:
        imm = buf + 2;
        imm1 = buf + 6;
        snprintf(b, 64, "0x%04x %s 0x%04x",*imm,oper,*imm1);
        //snprintf(b, 64, "%s",oper);
        size=10;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "r_%02x %s r_%02x",*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "r_%02x %s r_%02x",*r1,oper,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    }
    break;
case 3:// 7 4
    switch(*c) {
    case 1:
        r1 = buf + 2;
        imm = buf + 3;
        snprintf(b, 64, "%s r_%02x,0x%04x",oper,*r1,*imm);
        //snprintf(b, 64, "%s",oper);
        size=7;
        break;
    case 0:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "%s r_%02x,r_%02x",oper,*r1,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    default:
        r1 = buf + 2;
        r2 = buf + 3;
        snprintf(b, 64, "%s r_%02x,r_%02x",oper,*r1,*r2);
        //snprintf(b, 64, "%s",oper);
        size=4;
        break;
    }
    break;
case 4: // 6 3
    switch(*c) {
    case 1:
        imm = buf + 2;
        snprintf(b, 64, "%s 0x%04x",oper,*imm);

        //snprintf(b, 64, "%s",oper);
        size=6;
        break;
    case 0:
        r0 = buf + 2;
        snprintf(b, 64, "%s r_%02x",oper,*r0);

        //snprintf(b, 64, "%s",oper);
        size=3;
        break;
    default:
        r0 = buf + 2;
        snprintf(b, 64, "%s r_%02x",oper,*r0);

        //snprintf(b, 64, "%s",oper);
        size=3;
```

```
        break;
    }
    break;
case 5: //5
    imm = buf + 1;
    snprintf(b, 64, "%s 0x%04x", oper, *imm);

    //snprintf(b, 64, "%s", oper);
    size=5;
    break;
case 6://2
    r0 = buf + 1;
    snprintf(b, 64, "%s r_%02x", oper, *r0);

    //snprintf(b, 64, "%s", oper);
    size=2;
    break;
break;
}
return size;
}

static int disassemble(RAsm *a, RAsmOp *op, const ut8 *buf, int len) {
    const ut8 *p;
    const ut8 *r;
    const ut8 *r1;
    const ut32 *imm;
    const ut32 *imm1;

    switch (*buf) {
        case 2://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "+", 0);
            break;
        case 3://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "-", 0);
            break;
        case 4://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "*", 0);
            break;
        case 6://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "^", 0);
            break;
        case 9://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "&", 0);
            break;
        case 10://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "|", 0);
            break;
        case 12://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "<<", 0);
            break;
        case 13://8 8 11 5
            op->size = asm_baleful_getregs(buf, op->buf_asm, ">>", 0);
            break;
        case 5: // //9 9 12 6
            op->size = asm_baleful_getregs(buf, op->buf_asm, "/", 1);
            break;
        case 22: // 7 7 10 4
            op->size = asm_baleful_getregs(buf, op->buf_asm, "and", 2);
            break;
        case 23: // 7 7 10 4
            op->size = asm_baleful_getregs(buf, op->buf_asm, "cmp", 2);
            break;
        case 24: //7 4
            op->size = asm_baleful_getregs(buf, op->buf_asm, "mov", 3);
            break;
        case 30: // 6 3
            op->size = asm_baleful_getregs(buf, op->buf_asm, "push", 4);
            break;
        case 15: //5
            op->size = asm_baleful_getregs(buf, op->buf_asm, "call", 5);
            break;
    }
```

```
case 14: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jmp",5);
    break;
case 16: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jz",5);
    break;
case 17: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"js",5);
    break;
case 18: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jbe",5);
    break;
case 19: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jg",5);
    break;
case 20: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jns",5);
    break;
case 21: //5
    op->size = asm_baleful_getregs(buf,op->buf_asm,"jnz",5);
    break;
case 27: //3
    op->size = 3;
    r = buf + 1;
    r1 = buf + 2;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "mov r_%02x,[r_%02x]",*r,*r1);
    break;
case 28: //3
    r = buf + 1;
    r1 = buf + 2;
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "mov [r_%02x],r_%02x",*r,*r1);
    break;
case 11: //3
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "regX= regY==0");
    break;
case 7: //3
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "regX= NEG regY");
    break;
case 8: //3
    op->size = 3;
    snprintf (op->buf_asm, R_ASM_BUFSIZE, "regX= NOT regY");
    break;
case 25: //2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"++",6);
    break;
case 26: //2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"--",6);
    break;
case 31: //2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"pop",6);
    break;
case 32: // 2
    op->size = asm_baleful_getregs(buf,op->buf_asm,"apicall",6);
    break;
case 1:
    op->size = 1;
    strcpy (op->buf_asm, "ret");

    break;
case 0:
    op->size = 1;
    strcpy (op->buf_asm, "nop");

    break;
case 29:
    op->size = 1;
    strcpy (op->buf_asm, "end virtual");
    break;
default:
```



```
        op->size = 1;
        strcpy (op->buf_asm, "nop");
        break;
    }
    return op->size;
}

RAsmPlugin r_asm_plugin_baleful = {
    .name = "baleful",
    .arch = "baleful",
    .license = "LGPL3",
    .bits = 32,
    .desc = "Baleful",
    .init = NULL,
    .fini = NULL,
    .disassemble = &disassemble,
    //.assemble = null// &assemble
};

#ifdef CORELIB
struct r_lib_struct_t radare_plugin = {
    .type = R_LIB_TYPE_ASM,
    .data = &r_asm_plugin_baleful
};
#endif
```

. Código Virtual originalmente usado .

```
1000:  mov r_00,0x103a
1007:  mov r_01,0x1edb
100e:  mov r_03,r_00
1012:  mov r_05,0x174cf42
1019:  mov r_04,[r_03]
101c:  r_04=r_04 ^ r_05
1024:  mov [r_03],r_04
1024:  r_03=r_03 +0004
102c:  cmp r_01,r_03
1030:  jns 0x1019
1035:  jmp 0x103a
103a:  jmp 0x1bc0
103f:  ncall 0x00
1041:  ret
1042:  ncall 0x01
1044:  ret
1045:  ncall 0x02
1047:  ret
1048:  ncall 0x03
104a:  ret
104b:  ncall 0x04
104d:  ret
104e:  ncall 0x05
1050:  ret
1051:  ncall 0x06
1053:  ret
1054:  ncall 0x07
1056:  ret
1057:  ncall 0x08
1059:  ret
105a:  ncall 0x09
105c:  ret
105d:  ncall 0x0a
105f:  ret
1060:  ncall 0x0b
1062:  ret
1063:  ncall 0x0c
```

```
1065: ret
1066: ncall 0x0d
1068: ret
1069: ncall 0x0e
106b: ret
106c: ncall 0x0f
106e: ret
106f: ncall 0x10
1071: ret
1072: cmp r_00,r_00
1076: jz 0x107c
107b: ret
107d: ncall 0x11
107f: ret
1080: push r_1e
1083: push r_1d
1086: mov r_1e,r_01
108a: r_00 = r_1e * r_00
108f: r_00=r_00 +0008
1097: mov r_1d,r_00
109b: ncall 0x11
10a1: jz 0x10c2
10a6: mov r_04,r_1d
10aa: r_04 = r_04 >> 0003
10b5: mov [r_00],r_04
10b5: r_00=r_00 +0008
10bd: pop r_1d
10bf: pop r_1e
10c1: ret
10c2: endvm
10c3: mov r_00,0x0043
10ca: call 0x103f
10cf: mov r_00,0x006f
10d6: call 0x103f
10db: mov r_00,0x006e
10e2: call 0x103f
10e7: mov r_00,0x0067
10ee: call 0x103f
10f3: mov r_00,0x0072
10fa: call 0x103f
10ff: mov r_00,0x0061
1106: call 0x103f
110b: mov r_00,0x0074
1112: call 0x103f
1117: mov r_00,0x0075
111e: call 0x103f
1123: mov r_00,0x006c
112a: call 0x103f
112f: mov r_00,0x0061
1136: call 0x103f
113b: mov r_00,0x0074
1142: call 0x103f
1147: mov r_00,0x0069
114e: call 0x103f
1153: mov r_00,0x006f
115a: call 0x103f
115f: mov r_00,0x006e
1166: call 0x103f
116b: mov r_00,0x0073
1172: call 0x103f
1177: mov r_00,0x0021
117e: call 0x103f
1183: mov r_00,0x000a
118a: call 0x103f
118f: ret
1192: mov r_00,0x0053
1199: call 0x103f
119e: mov r_00,0x006f
11a5: call 0x103f
11aa: mov r_00,0x0072
11b1: call 0x103f
11b6: mov r_00,0x0072
```

```
11bd: call 0x103f
11c2: mov r_00,0x0079
11c9: call 0x103f
11ce: mov r_00,0x002c
11d5: call 0x103f
11da: mov r_00,0x0020
11e1: call 0x103f
11e6: mov r_00,0x0077
11ed: call 0x103f
11f2: mov r_00,0x0072
11f9: call 0x103f
11fe: mov r_00,0x006f
1205: call 0x103f
120a: mov r_00,0x006e
1211: call 0x103f
1216: mov r_00,0x0067
121d: call 0x103f
1222: mov r_00,0x0020
1229: call 0x103f
122e: mov r_00,0x0070
1235: call 0x103f
123a: mov r_00,0x0061
1241: call 0x103f
1246: mov r_00,0x0073
124d: call 0x103f
1252: mov r_00,0x0073
1259: call 0x103f
125e: mov r_00,0x0077
1265: call 0x103f
126a: mov r_00,0x006f
1271: call 0x103f
1276: mov r_00,0x0072
127d: call 0x103f
1282: mov r_00,0x0064
1289: call 0x103f
128e: mov r_00,0x0021
1295: call 0x103f
129a: mov r_00,0x000a
12a1: call 0x103f
12a6: ret
12a9: push r_09
12ac: push r_0a
12af: mov r_0a,r_00
12b3: mov r_01,0x001e
12ba: mov r_00,0x0004
12c1: call 0x1080
12c6: mov r_09,r_00
12ca: mov r_01,0x0004
12d1: mov r_00,0x0004
12d8: call 0x1080
12dd: mov r_05,r_00
12e1: mov r_1e,0x00fd
12eb: mov [r_05],r_1e
12eb: mov r_1e,0x0001
12f2: r_1e = r_1e * 0004
12fa: mov r_00,r_05
12fe: r_1e=r_1e + r_00
1303: mov r_01,r_1e
1307: mov r_1e,0x000e
1311: mov [r_01],r_1e
1311: mov r_1e,0x0001
1318: r_1e = r_1e * 0008
1320: mov r_00,r_05
1324: r_1e=r_1e + r_00
1329: mov r_01,r_1e
132d: mov r_1e,0x0063
1337: mov [r_01],r_1e
1337: mov r_1e,0x0001
133e: r_1e = r_1e * 000c
1346: mov r_00,r_05
134a: r_1e=r_1e + r_00
134f: mov r_01,r_1e
```

```
1353: mov r_1e,0x004f
135d: mov [r_01],r_1e
135d: mov r_1e,0x008d
1367: mov [r_09],r_1e
1367: mov r_1e,0x0001
136e: r_1e = r_1e * 0004
1376: mov r_00,r_09
137a: r_1e=r_1e + r_00
137f: mov r_01,r_1e
1383: mov r_1e,0x006f
138d: mov [r_01],r_1e
138d: mov r_1e,0x0001
1394: r_1e = r_1e * 0008
139c: mov r_00,r_09
13a0: r_1e=r_1e + r_00
13a5: mov r_01,r_1e
13a9: mov r_1e,0x0000
13b3: mov [r_01],r_1e
13b3: mov r_1e,0x0001
13ba: r_1e = r_1e * 000c
13c2: mov r_00,r_09
13c6: r_1e=r_1e + r_00
13cb: mov r_01,r_1e
13cf: mov r_1e,0x0024
13d9: mov [r_01],r_1e
13d9: mov r_1e,0x0001
13e0: r_1e = r_1e * 0010
13e8: mov r_00,r_09
13ec: r_1e=r_1e + r_00
13f1: mov r_01,r_1e
13f5: mov r_1e,0x0098
13ff: mov [r_01],r_1e
13ff: mov r_1e,0x0001
1406: r_1e = r_1e * 0014
140e: mov r_00,r_09
1412: r_1e=r_1e + r_00
1417: mov r_01,r_1e
141b: mov r_1e,0x007c
1425: mov [r_01],r_1e
1425: mov r_1e,0x0001
142c: r_1e = r_1e * 0018
1434: mov r_00,r_09
1438: r_1e=r_1e + r_00
143d: mov r_01,r_1e
1441: mov r_1e,0x0010
144b: mov [r_01],r_1e
144b: mov r_1e,0x0001
1452: r_1e = r_1e * 001c
145a: mov r_00,r_09
145e: r_1e=r_1e + r_00
1463: mov r_01,r_1e
1467: mov r_1e,0x0010
1471: mov [r_01],r_1e
1471: mov r_1e,0x0001
1478: r_1e = r_1e * 0020
1480: mov r_00,r_09
1484: r_1e=r_1e + r_00
1489: mov r_01,r_1e
148d: mov r_1e,0x009c
1497: mov [r_01],r_1e
1497: mov r_1e,0x0001
149e: r_1e = r_1e * 0024
14a6: mov r_00,r_09
14aa: r_1e=r_1e + r_00
14af: mov r_01,r_1e
14b3: mov r_1e,0x0060
14bd: mov [r_01],r_1e
14bd: mov r_1e,0x0001
14c4: r_1e = r_1e * 0028
14cc: mov r_00,r_09
14d0: r_1e=r_1e + r_00
14d5: mov r_01,r_1e
```

```
14d9: mov r_1e,0x0007
14e3: mov [r_01],r_1e
14e3: mov r_1e,0x0001
14ea: r_1e = r_1e * 002c
14f2: mov r_00,r_09
14f6: r_1e=r_1e + r_00
14fb: mov r_01,r_1e
14ff: mov r_1e,0x0010
1509: mov [r_01],r_1e
1509: mov r_1e,0x0001
1510: r_1e = r_1e * 0030
1518: mov r_00,r_09
151c: r_1e=r_1e + r_00
1521: mov r_01,r_1e
1525: mov r_1e,0x008b
152f: mov [r_01],r_1e
152f: mov r_1e,0x0001
1536: r_1e = r_1e * 0034
153e: mov r_00,r_09
1542: r_1e=r_1e + r_00
1547: mov r_01,r_1e
154b: mov r_1e,0x0063
1555: mov [r_01],r_1e
1555: mov r_1e,0x0001
155c: r_1e = r_1e * 0038
1564: mov r_00,r_09
1568: r_1e=r_1e + r_00
156d: mov r_01,r_1e
1571: mov r_1e,0x0010
157b: mov [r_01],r_1e
157b: mov r_1e,0x0001
1582: r_1e = r_1e * 003c
158a: mov r_00,r_09
158e: r_1e=r_1e + r_00
1593: mov r_01,r_1e
1597: mov r_1e,0x0010
15a1: mov [r_01],r_1e
15a1: mov r_1e,0x0001
15a8: r_1e = r_1e * 0040
15b0: mov r_00,r_09
15b4: r_1e=r_1e + r_00
15b9: mov r_01,r_1e
15bd: mov r_1e,0x009c
15c7: mov [r_01],r_1e
15c7: mov r_1e,0x0001
15ce: r_1e = r_1e * 0044
15d6: mov r_00,r_09
15da: r_1e=r_1e + r_00
15df: mov r_01,r_1e
15e3: mov r_1e,0x0060
15ed: mov [r_01],r_1e
15ed: mov r_1e,0x0001
15f4: r_1e = r_1e * 0048
15fc: mov r_00,r_09
1600: r_1e=r_1e + r_00
1605: mov r_01,r_1e
1609: mov r_1e,0x0007
1613: mov [r_01],r_1e
1613: mov r_1e,0x0001
161a: r_1e = r_1e * 004c
1622: mov r_00,r_09
1626: r_1e=r_1e + r_00
162b: mov r_01,r_1e
162f: mov r_1e,0x0010
1639: mov [r_01],r_1e
1639: mov r_1e,0x0001
1640: r_1e = r_1e * 0050
1648: mov r_00,r_09
164c: r_1e=r_1e + r_00
1651: mov r_01,r_1e
1655: mov r_1e,0x0085
165f: mov [r_01],r_1e
```

```
165f: mov r_1e,0x0001
1666: r_1e = r_1e * 0054
166e: mov r_00,r_09
1672: r_1e=r_1e + r_00
1677: mov r_01,r_1e
167b: mov r_1e,0x0061
1685: mov [r_01],r_1e
1685: mov r_1e,0x0001
168c: r_1e = r_1e * 0058
1694: mov r_00,r_09
1698: r_1e=r_1e + r_00
169d: mov r_01,r_1e
16a1: mov r_1e,0x0011
16ab: mov [r_01],r_1e
16ab: mov r_1e,0x0001
16b2: r_1e = r_1e * 005c
16ba: mov r_00,r_09
16be: r_1e=r_1e + r_00
16c3: mov r_01,r_1e
16c7: mov r_1e,0x003c
16d1: mov [r_01],r_1e
16d1: mov r_1e,0x0001
16d8: r_1e = r_1e * 0060
16e0: mov r_00,r_09
16e4: r_1e=r_1e + r_00
16e9: mov r_01,r_1e
16ed: mov r_1e,0x00a2
16f7: mov [r_01],r_1e
16f7: mov r_1e,0x0001
16fe: r_1e = r_1e * 0064
1706: mov r_00,r_09
170a: r_1e=r_1e + r_00
170f: mov r_01,r_1e
1713: mov r_1e,0x0061
171d: mov [r_01],r_1e
171d: mov r_1e,0x0001
1724: r_1e = r_1e * 0068
172c: mov r_00,r_09
1730: r_1e=r_1e + r_00
1735: mov r_01,r_1e
1739: mov r_1e,0x000b
1743: mov [r_01],r_1e
1743: mov r_1e,0x0001
174a: r_1e = r_1e * 006c
1752: mov r_00,r_09
1756: r_1e=r_1e + r_00
175b: mov r_01,r_1e
175f: mov r_1e,0x0010
1769: mov [r_01],r_1e
1769: mov r_1e,0x0001
1770: r_1e = r_1e * 0070
1778: mov r_00,r_09
177c: r_1e=r_1e + r_00
1781: mov r_01,r_1e
1785: mov r_1e,0x0090
178f: mov [r_01],r_1e
178f: mov r_1e,0x0001
1796: r_1e = r_1e * 0074
179e: mov r_00,r_09
17a2: r_1e=r_1e + r_00
17a7: mov r_01,r_1e
17ab: mov r_1e,0x0077
17b5: mov [r_01],r_1e
17b5: mov r_03,0x0000
17bc: jmp 0x17c6
17c1: jmp 0x1878
17c6: mov r_01,0x0000
17cd: mov r_02,0x0000
17d4: jmp 0x1860
17d9: mov r_1e,r_02
17dd: r_1e = r_1e * 0004
17e5: mov r_00,r_0a
```

```
17e9: r_1e=r_1e + r_00
17ee: mov r_03,r_1e
17f2: mov r_04,[r_03]
17f5: r_00 = r_02 / 0004
17f5: r_03 = r_02 mod 0004
17fe: mov r_1e,r_03
1802: r_1e = r_1e * 0004
180a: mov r_00,r_05
180e: r_1e=r_1e + r_00
1813: mov r_03,r_1e
1817: mov r_03,[r_03]
181a: r_04=r_04 ^ r_03
181f: mov r_1e,r_02
1823: r_1e = r_1e * 0004
182b: mov r_00,r_09
182f: r_1e=r_1e + r_00
1834: mov r_03,r_1e
1838: mov r_03,[r_03]
183b: mov r_1e,r_04
183f: mov r_00,r_03
1843: cmp r_1e,r_00
1847: jnz 0x1851
184c: jmp 0x1858
1851: mov r_01,0x0001
1858: r_02=r_02 +0001
1860: mov r_03,r_01
1864: mov r_1e,r_02
1868: mov r_00,0x001e
186f: cmp r_1e,r_00
1873: js 0x17d9
1078: r_03 and r_03
187c: jnz 0x1952
1881: mov r_00,0x0043
1888: call 0x103f
188d: mov r_00,0x006f
1894: call 0x103f
1899: mov r_00,0x006e
18a0: call 0x103f
18a5: mov r_00,0x0067
18ac: call 0x103f
18b1: mov r_00,0x0072
18b8: call 0x103f
18bd: mov r_00,0x0061
18c4: call 0x103f
18c9: mov r_00,0x0074
18d0: call 0x103f
18d5: mov r_00,0x0075
18dc: call 0x103f
18e1: mov r_00,0x006c
18e8: call 0x103f
18ed: mov r_00,0x0061
18f4: call 0x103f
18f9: mov r_00,0x0074
1900: call 0x103f
1905: mov r_00,0x0069
190c: call 0x103f
1911: mov r_00,0x006f
1918: call 0x103f
191d: mov r_00,0x006e
1924: call 0x103f
1929: mov r_00,0x0073
1930: call 0x103f
1935: mov r_00,0x0021
193c: call 0x103f
1941: mov r_00,0x000a
1948: call 0x103f
194d: jmp 0x1a66
1952: mov r_00,0x0053
1959: call 0x103f
195e: mov r_00,0x006f
1965: call 0x103f
196a: mov r_00,0x0072
```

```
1971: call 0x103f
1976: mov r_00,0x0072
197d: call 0x103f
1982: mov r_00,0x0079
1989: call 0x103f
198e: mov r_00,0x002c
1995: call 0x103f
199a: mov r_00,0x0020
19a1: call 0x103f
19a6: mov r_00,0x0077
19ad: call 0x103f
19b2: mov r_00,0x0072
19b9: call 0x103f
19be: mov r_00,0x006f
19c5: call 0x103f
19ca: mov r_00,0x006e
19d1: call 0x103f
19d6: mov r_00,0x0067
19dd: call 0x103f
19e2: mov r_00,0x0020
19e9: call 0x103f
19ee: mov r_00,0x0070
19f5: call 0x103f
19fa: mov r_00,0x0061
1a01: call 0x103f
1a06: mov r_00,0x0073
1a0d: call 0x103f
1a12: mov r_00,0x0073
1a19: call 0x103f
1a1e: mov r_00,0x0077
1a25: call 0x103f
1a2a: mov r_00,0x006f
1a31: call 0x103f
1a36: mov r_00,0x0072
1a3d: call 0x103f
1a42: mov r_00,0x0064
1a49: call 0x103f
1a4e: mov r_00,0x0021
1a55: call 0x103f
1a5a: mov r_00,0x000a
1a61: call 0x103f
1a66: pop r_0a
1a68: pop r_09
1a6a: ret
1a6d: mov r_00,0x0050
1a74: call 0x103f
1a79: mov r_00,0x006c
1a80: call 0x103f
1a85: mov r_00,0x0065
1a8c: call 0x103f
1a91: mov r_00,0x0061
1a98: call 0x103f
1a9d: mov r_00,0x0073
1aa4: call 0x103f
1aa9: mov r_00,0x0065
1ab0: call 0x103f
1ab5: mov r_00,0x0020
1abc: call 0x103f
1ac1: mov r_00,0x0065
1ac8: call 0x103f
1acd: mov r_00,0x006e
1ad4: call 0x103f
1ad9: mov r_00,0x0074
1ae0: call 0x103f
1ae5: mov r_00,0x0065
1aec: call 0x103f
1af1: mov r_00,0x0072
1af8: call 0x103f
1afd: mov r_00,0x0020
1b04: call 0x103f
1b09: mov r_00,0x0079
1b10: call 0x103f
```



```
1b15: mov r_00,0x006f
1b1c: call 0x103f
1b21: mov r_00,0x0075
1b28: call 0x103f
1b2d: mov r_00,0x0072
1b34: call 0x103f
1b39: mov r_00,0x0020
1b40: call 0x103f
1b45: mov r_00,0x0070
1b4c: call 0x103f
1b51: mov r_00,0x0061
1b58: call 0x103f
1b5d: mov r_00,0x0073
1b64: call 0x103f
1b69: mov r_00,0x0073
1b70: call 0x103f
1b75: mov r_00,0x0077
1b7c: call 0x103f
1b81: mov r_00,0x006f
1b88: call 0x103f
1b8d: mov r_00,0x0072
1b94: call 0x103f
1b99: mov r_00,0x0064
1ba0: call 0x103f
1ba5: mov r_00,0x003a
1bac: call 0x103f
1bb1: mov r_00,0x0020
1bb8: call 0x103f
1bbd: ret
1bc0: push r_09
1bc3: push r_0a
1bc6: push r_0b
1bc9: call 0x1a6d
1bce: mov r_01,0x001e
1bd5: mov r_00,0x0004
1bdc: call 0x1080
1be1: mov r_0b,r_00
1be5: jmp 0x1bef
1bea: jmp 0x1d7a
1bef: mov r_09,0x0000
1bf6: jmp 0x1d66
1bfb: mov r_1e,r_09
1bff: r_1e = r_1e * 0004
1c07: mov r_00,r_0b
1c0b: r_1e=r_1e + r_00
1c10: mov r_0a,r_1e
1c14: call 0x104b
1c19: mov r_01,r_00
1c20: mov [r_0a],r_01
1c20: mov r_01,[r_0a]
1c23: mov r_1e,r_01
1c27: mov r_00,0x000a
1c2e: cmp r_1e,r_00
1c32: jz 0x1c3c
1c37: jmp 0x1d5e
1c3c: mov r_00,0x0053
1c43: call 0x103f
1c48: mov r_00,0x006f
1c4f: call 0x103f
1c54: mov r_00,0x0072
1c5b: call 0x103f
1c60: mov r_00,0x0072
1c67: call 0x103f
1c6c: mov r_00,0x0079
1c73: call 0x103f
1c78: mov r_00,0x002c
1c7f: call 0x103f
1c84: mov r_00,0x0020
1c8b: call 0x103f
1c90: mov r_00,0x0077
1c97: call 0x103f
1c9c: mov r_00,0x0072
```

```
1ca3: call 0x103f
1ca8: mov r_00,0x006f
1caf: call 0x103f
1cb4: mov r_00,0x006e
1cbb: call 0x103f
1cc0: mov r_00,0x0067
1cc7: call 0x103f
1ccc: mov r_00,0x0020
1cd3: call 0x103f
1cd8: mov r_00,0x0070
1cdf: call 0x103f
1ce4: mov r_00,0x0061
1ceb: call 0x103f
1cf0: mov r_00,0x0073
1cf7: call 0x103f
1cfc: mov r_00,0x0073
1d03: call 0x103f
1d08: mov r_00,0x0077
1d0f: call 0x103f
1d14: mov r_00,0x006f
1d1b: call 0x103f
1d20: mov r_00,0x0072
1d27: call 0x103f
1d2c: mov r_00,0x0064
1d33: call 0x103f
1d38: mov r_00,0x0021
1d3f: call 0x103f
1d44: mov r_00,0x000a
1d4b: call 0x103f
1d50: mov r_00,0x0000
1d57: pop r_0b
1d59: pop r_0a
1d5b: pop r_09
1d5d: ret
1d5e: r_09=r_09+0001
1d66: mov r_1e,r_09
1d6a: mov r_00,0x001e
1d71: cmp r_1e,r_00
1d75: js 0x1bfb
1d7a: call 0x104b
1d7f: mov r_01,r_00
1d83: mov r_1e,r_01
1d87: mov r_00,0x000a
1d8e: cmp r_1e,r_00
1d92: jnz 0x1d9c
1d97: jmp 0x1ebe
1d9c: mov r_00,0x0053
1da3: call 0x103f
1da8: mov r_00,0x006f
1daf: call 0x103f
1db4: mov r_00,0x0072
1dbb: call 0x103f
1dc0: mov r_00,0x0072
1dc7: call 0x103f
1dcc: mov r_00,0x0079
1dd3: call 0x103f
1dd8: mov r_00,0x002c
1ddf: call 0x103f
1de4: mov r_00,0x0020
1deb: call 0x103f
1df0: mov r_00,0x0077
1df7: call 0x103f
1dfc: mov r_00,0x0072
1e03: call 0x103f
1e08: mov r_00,0x006f
1e0f: call 0x103f
1e14: mov r_00,0x006e
1e1b: call 0x103f
1e20: mov r_00,0x0067
1e27: call 0x103f
1e2c: mov r_00,0x0020
1e33: call 0x103f
```

```
1e38: mov r_00,0x0070
1e3f: call 0x103f
1e44: mov r_00,0x0061
1e4b: call 0x103f
1e50: mov r_00,0x0073
1e57: call 0x103f
1e5c: mov r_00,0x0073
1e63: call 0x103f
1e68: mov r_00,0x0077
1e6f: call 0x103f
1e74: mov r_00,0x006f
1e7b: call 0x103f
1e80: mov r_00,0x0072
1e87: call 0x103f
1e8c: mov r_00,0x0064
1e93: call 0x103f
1e98: mov r_00,0x0021
1e9f: call 0x103f
1ea4: mov r_00,0x000a
1eab: call 0x103f
1eb0: mov r_00,0x0000
1eb7: pop r_0b
1eb9: pop r_0a
1ebb: pop r_09
1ebd: ret
1ebe: mov r_00,r_0b
1ec2: call 0x12a9
1ec7: mov r_00,0x0000
1ece: pop r_0b
1ed0: pop r_0a
1ed2: pop r_09
1ed4: ret
```