Introducción al reversing con r2

pancake <pancake@nopcode.org>

(1/6)

RootedLabs 2013

Bienvenidos al taller de ingeniería inversa con radare2.

Para seguir el taller es necesario tener configurada la red y instalar la máquina virtual y un putty.

Al entrar usaremos: root/radare y seguiremos los pasos descritos en el README

En el primer arranque deberán definir una contraseña para el usuario ’radare’

Para realizar el taller usaremos la terminal en todo momento, asi que el alumno debe saber manejarse por la shell.

Test de nivel

- Alguien sabe que es radare?

Test de nivel

- Alguien sabe que es radare?

- Y la ingenieria inversa?

Test de nivel

- Alguien sabe que es radare?
- Y la ingenieria inversa?
- Cómo os manejais en la shell de un sistema *nix?

Introducción

Radare es un proyecto con más de 7 años de desarrollo a sus espaldas, empezaremos viendo su historia y entenderemos las filosofias aplicadas en su desarrollo.

En este taller aprenderemos el uso de la herramienta a través de varios ejercicios prácticos que nos ayudarán a entender mejor el funcionamiento tanto del programa como del elemento que estemos analizando.

Procuraremos centrarnos en la ingeniería inversa, aunque mostraremos por encima las otras caras de radare2 cómo analisis forense, la depuración, fuzzing, exploiting, entre otras.

Qué es el reversing?

La ingeniería inversa es el proceso por el cual se pretende descubrir el funcionamiento técnico de algo a partir del análisis de sus resultados, estructuras de datos, funciones y operaciones.

Este curso se enfoca al reversing de binarios (programas ejecutables) con el objetivo de entender su funcionamiento y manipularlo para adaptarlo a nuestras "necesidades".

Radare es un framework de reversing que nos ofrece varias herramientras accesibles a traves de programas y librerias con bindings para multitud de lenguajes de scripting.

Historia

Radare nació en 2006 como una herramienta forense; un editor hexadecimal de 64 bits para hacer busquedas en discos duros.

Pronto el proyecto creció permitiendo desensamblar codigo maquina de varias arquitecturas, depurar en windows, linux y mac, scripting..

Despues de 4 años creciendo decidí reescribirlo desde cero con tal de solventar varias limitaciones implicitas en el diseño monolítico de la primera versión.

Así nació radare2, implementado sobre un conjunto de librerias, permitiendo un scripting completo a través de las APIs, mejor rendimiento y calidad de codigo.

Capacidades

radare2 es un framework que nos ofrece:

- ensamblador / desensamblador
- editor hexadecimal por bloques de 64bits
- calcular checksums por bloques
- maneja transparentemente procesos, discos, ficheros, ram, ..
- montar sistemas de archivos (fat, ntfs, ext2, ..)
- analizar binarios de windows, linux, mac, java, dalvik, ..
- depurador (w32, linux, mac, ios)
- busqueda de diferencias binarias
- herramientas para la creacion de shellcodes
- soporte para varios lenguajes de scripting (python, js, ..)

Filosofia

Desde sus inicios ha sido un proyecto personal, aunque ha recibido numerosas contribuciones y durante este tiempo la comunidad ha crecido, alimentando el proyecto con ideas, nuevas funcionalidades.

Lo distribuyo bajo licencia LGPL3, lo cual permite su uso para proyectos propietarios, aunque obliga a publicar cualquier modificación de las librerias.

Casi todas las funcionalidades se pueden extender con plugins o scripts, con lo que todos los usuarios se benefician de las mejoras que realice cualquier proyecto que use el framework.

Muchas ideas han sido basadas en conceptos de UNIX. Como la ortogonalidad, las pipes, filtros, todo el i/o de trata como un fichero, ...

Aprendizaje

La curva de aprendizaje de radare es parecida a la de otros programas de UNIX.

Al principio los comandos parecen complicados, pero conforme vayamos aprendiendo la logica detrás de cada caracter iremos relacionando conceptos más rapidamente.

La shell de radare tiene muchos parecidos con las shells posix de los sistemas *NIX, es por esto que conocerla nos facilitará el aprendizaje.

Ya que el desarrollo es muy activo es recomendable usar la versión de git y entrar en el irc para enterarse de novedades o reportar ideas/fallos/...

Desarrollo

R2 está escrito en C y compila en multitud de plataformas sin necesidad de ninguna dependencia más que make, una shell posix y un compilador.

El codigo se publica en un repositorio git (hay mirrors) y se intenta cortar una versión estable cada 6 meses. Aunque suele ser cada 300 commits.

Existe una test suite que pretende detectar errores de funcionamiento pero aún no hay suficientes tests cómo para verificar todas las opciones del software.

Para generar los bindings desarrollé un programa llamado ’valabind’ que permite generar bindings para python−ctypes, nodejs o swig (lua, perl, ...) a partir de las descripciones de las apis en formato vapi (del lenguaje Vala)

La mejor documentación siempre es el código fuente. La documentación de la api se genera a partir de las interfaces Vapi usando valadoc.

Instalando desde GIT

Los alumnos aprenderán a instalar radare2 desde su repositorio con tal de poder seguir el taller usando la ultima versión en desarrollo.

```
$ git clone git://github.com/radare/radare2
$ cd radare2
$ ./configure --prefix=/usr
$ make
$ sudo make symstall
$ r2 -v
radare2 0.9.3git @ darwin-little-x86-64 build 2013-02-01
commit: dc2690e5294a201aeef96a038ddf152fc9f69257
```

Aunque GNU/Linux y OSX sean las principales plataformas soportadas, tambien es posible usar radare2 desde cualquier sistema BSD o Windows.

Detalles

El proceso de instalación es el común de la mayoria de aplicaciones libres, aunque cabe destacar ciertos detalles:

− usa acr (en vez de autoconf)

− make symstall instala con symlinks (más rápido y util para desarrollar sin instalar cada vez)

− sys/install.sh instala con symlinks a /usr, es comodo usar este metodo si estas desarrollando (cada compilacion no requiere ser instalada)

− make uninstall/purge − para desinstalar la versión actual o todas las instalaciones previas.

− en el directorio sys/ existen varios scripts que facilitan las compilaciones nativas, de windows o android.

Ejercicios y dudas

En esta primera parte del curso lo importante es llegar a tener una instalación funcional de radare2. Punto indispensable para seguir el resto de apartados.

Instalación fuera de /usr/bin

```
$ ./configure --prefix=/opt/r2 --with-rpath
$ make
$ sudo make symstall
$ export PATH=/opt/r2/bin:$PATH
```

Limpiar sistema de instalaciones previas

```
$ sudo make purge
```

Crosscompilar para windows:

```
$ sys/mingw32.sh
```

Crosscompilar para android:

```
$ sys/android-arm.sh
```

Introducción al reversing con r2

pancake <pancake@nopcode.org>

(2/6)

Herramientas

En esta sección se hará un repaso a las varias herramientas incluidas en radare2. Realizaremos varios ejercicios con el objetivo de comprender su uso y combinarlas para componer utilidades más complejas.

Podemos observar que todos los comandos terminan con un ’2’. Esto permite la convivencia entre radare1 y radare2 en el sistema.

Cada herramienta implementa parte de las funcionalidades de las librerias, excepto ’r2’ que las agrupa todas.

Mediante la flag ’−h’ obtendremos un resumen de las opciones y ’−L’ nos listará los plugins soportados.

Los grafos se generan en formato dot, para visualiarlos usaremos:
’dot −Tpng −oa.png a.dot’

rax2

Podriamos decir que rax2 es una calculadora multibase; permite el uso de expresiones matemáticas, cambio de endian, de base y formato.

Las expresiones pueden ser pasadas como argumentos o por stdin.

```
$ rax2 3+0x80       $ rax2 0x80+3           $ echo 0x80+3 | rax2
0x83                131                     131

$ rax2 -s 4142      $ rax2 -S AB            $ rax2 -S < bin.foo
AB                  4142                    ...

$ rax2 -e 33        $ rax2 -e 0x21000000    $ rax2 -k 90203010
0x21000000          33                      +--[0x10302090]---+
```

Desde r2 podemos evaluar estas expresiones mediante el comando ’? ’

rahash2

Los dispositivos forenses de captura de discos suelen calcular el hash los datos con tal de asegurar que no han sido alterados durante su manipulación.

Esta herramienta nos permite calcular checksums de ficheros o strings.

```
$ rahash2 -L                    # listar todos los algoritmos
$ rahash2 -qa crc32 /bin/ls  # calculate crc32 of given file
$ rahash2 -a entropy -Bb 512 /bin/ls # calculate entropy of file every 512
bytes
$ rahash2 -ks "hello world"  # show randomkey art for given string
```

Podemos especificar el tamaño de bloque (-b) y usar un checksum incremental (-B), con tal de calcular el resultado total de golpe o por bloques. Esto puede sernos util para averiguar que partes de un firmware o disco han sido modificados de una version a otra.

rahash2 tambien permite realizar calculos de entropia, distancia de hamming, porcentaje de caracteres alfanumericos, ..

radiff2

Compara dos ficheros a nivel de byte con soporte de cambios con desplazamiento. Tambien es posible calcular un bindiffing a nivel del analysis de codigo de los dos programas '−C'.

```
$ radiff2 a b # repetir ejercicio anterior
```

Tambien nos permite calcular la distancia de texto '−s' entre dos documentos o cadenas de texto.

Mostrar las diferencias de codigo analizado:

```
$ radiff2 −C /bin/true /bin/false
```

Mostrar que bytes cambian de un fichero a otro de tamaños distintos:

```
$ radiff2 −pd /bin/true /bin/false
```

rasm2

Este programa permite ensamblar y desensamblar multitud de arquitecturas.

```
$ rasm2 -h      # mostrar ayuda
$ rasm2 -L      # listar plugins
```

Ensamblando

```
$ rasm2 -a x86 -b 32 'mov eax, 33'
b821000000
$ echo 'push eax;nop' | rasm2 -f -
509090
$ rasm2 -a x86 -b 32 -f file.asm
```

Desensamblando:

```
$ rasm2 -a x86 -b 32 -d '90'
nop
$ echo 77 | rasm2 -dBf -
aaa
aaa
```

rafind2

Este programa nos permite realizar busquedas de cadenas de texto o bytes, con posibilidad de aplicarles una máscara binaria.

Los resultados se muestran de forma cómoda para ser procesada por otras aplicaciones. Mediante las flags −Z y −X podemos visualizar las cadenas de texto y un hexdump de los resultados.

Por ejemplo:

```
$ rafind2 −Zs lib /bin/ls | grep dy
0x1649 lib/dyld
0x1745 lib/libncurses.5.4.dylib
0x177d lib/libSystem.B.dylib
...
```

ragg2

Permite compilar shellcodes, y aplicarle un encoder y paddings. Tambien permite generar binarios o ejecutarlas para comprobar que funcionan correctamente.

ragg2−cc nos permite compilar código C para generar shellcodes.

Crear una shellcode para linux−x86−32 que ejecute /bin/sh:

```
$ ragg2 −a x86 −b 32 −k linux −i exec
```

Crear una shellcode desde C:

```
$ cat a.c
int main() { write(1,"Hi\n", 3); exit(1); }
$ ragg2−cc −x a.c
e90000000083ec0ce800000000588d882a000000b804000000606a03516a0150cd80...
$ ragg2−cc a.c
$ ./a.c.bin
Hi
```

rabin2

Parsea binarios y extrae información de simbolos, imports, strings, ...

```
$ echo $(rabin2 -L|awk '{print $2}')
dex elf elf64 fs java mach0 mach064 mz p9 pe pe64 rar dyldcache fatmach0
```

En caso de ser un binario gordo (fat). Deberemos usar las flags -a y -b para especificar el sub-binario.

Mediante la flag '-c' podemos crear binarios 'tiny'.o

```
$ rabin2 -c elf:cc trap
$ ./trap
Trace/BPT trap
```

Extraer la información del binario en formato de comandos de radare con -r

```
$ rabin2 -risSeI /bin/ls > ls.txt
$ r2 -i ls.txt /bin/ls
```

rarun2

Rarun nos permite definir entornos de ejecución, donde podemos especificar un chroot, diferente UID, ficheros como stdin, argumentos, conexiones de red, timeouts...

Algunos casos de uso:

- depurar programas delicados
- test suite
- fuzzing
- ..

```
$ rarun2 timeout=3 program=/bin/cat          # matar el cat despues de 3
segundos
$ rarun2 listen=9999 program=/bin/sh         # bindear una shell por tcp
$ rarun2 stdin=input.txt program=./crackme   # especificar entrada desde un
fichero
$ rarun2 stdin=/etc/services program=/usr/bin/grep arg1=pop
```

r2

Finalmente nos encontramos con la herramienta principal de radare2. Construida sobre ’r_core’ constituye la pieza central del resto de librerias y permite la interacción mediante la linea de comandos, scripts o modo visual y web.

Cargar un fichero sin analizarlo:

```
$ r2 -n /bin/ls
```

Lanzando un script:

```
$ cat make-test.r
x;pi 4;s+33
$ r2 -qi make-test.r -w test
```

Recargando un proyecto

```
$ r2 -p blah
```

Abrir en modo escritura

```
$ r2 -w program
```

Jugando con tuberias

Los comandos podemos unirlos facilmente con tal de crear nuevas herramientas o automatizar ciertas tareas.

```
$ ( rafind2 -rslib ; echo 'w lob@@hit*' ) | r2 -qw ls
$ ragg2 -ri exec > shellcode
$ r2 -nqfcpDi shellcode | grep syscall
> pd > disasm.txt
> !grep mov disasm.txt | grep eax
> !!grep mov disasm.txt~eax
> pdf~mov| grep eax
```

Comandos de ayuda: !? ~??

Dudas?

Hasta aqui el resumen de todas las herramientas incluidas en radare2.

A partir de ahora nos centraremos en usar ’radare2’, y dejaremos el resto a parte. Si tenéis alguna duda.. este es vuestro momento.

Introducción al reversing con r2

pancake <pancake@nopcode.org>

(3/6)

Manejo por linea de comandos

Una de las características más notables y temidas de radare es la sintaxis de los comandos.

Durante éste capitulo entenderemos la logica que se esconde detrás de los comandos y veremos como manejarnos en la shell para realizar varias tareas sobre ficheros binarios.

La shell nos permite interactuar con todas las funcionalidades del programa. Nos será de gran ayuda, no tan solo para tener una interfaz en linea, sino para automatizar tareas sin usar la API.

Empecemos!

El prompt

Al iniciar radare se nos presenta un prompt indicando el offset donde estamos.

```
$ r2 -
[0x00000000]>
```

El argumento '−', es un alias para malloc://512. r2 −L lista los plugins de IO
A partir de ahora en adelante usaremos '$' como simbolo de la shell del sistema y '>' como simbolo de la shell de radare.

Si introducimos '?' veremos la lista de los comandos soportados. Para obtener la ayuda de cada comando, simplemente hay que escribir '?' al final de cada comando. Los comandos se separan por el carácter ';'.

```
> ?;p?;c?
```

Al igual que con readline (no es readline), podemos movernos por el historial de comandos con las flechas y autocompletar con el tabulador.

Evaluando expresiones

Para obtener la ayuda del comando ’?’ debemos teclear ’???’.

Existen varios subcomandos dentro de ’?’ que pueden ser utiles para calcular operaciones o interactuar con el usuario.

```
> ? 33+2
35 0x23 043 0000:0023 35 00100011 35.0 0.000000
```

Podemos listar las variables ’$’ con el comando ’?$?’. Éstas son manejadas internamente por radare y utilizables desde cualquier expresion evaluable.

```
> ?v ${asm.bits}
```

Entrecomillados

Las comillas evitan que se evaluen caracteres especiales como ’@’, ’|’, ’;’, ...

```
> ?e Hello @ World
> "?e Hello @ World"
```

Puede sernos util...

- buscar strings "/ hello@world"
- comentarios "CC hehe|hoho"
- escribir string "w ola@ke|ase;"
- echo ?e "?e Contactadme @trufae"

Mostrando datos

El comando ’p’ nos permite mostrar la información del bloque actual en el formato deseado.

```
> px          # hexdump (alias: ’x’)
> pd          # desensamblado
```

Si pulsamos ’<enter>’ con una linea vacia ejecutaremos el último comando de print pero desplazandonos al bloque siguiente.

Al ser un editor hexadecimal basado en bloques deberemos especificar el tamaño como argumento o definiendo el tamaño con ’b’ o ’@!’. Usaremos ’s’ para cambiar y cambiar nuestra posición con ’s’.

```
> b 64
> s 0x80480
```

Seeks

Para movernos dentro del espacio de memoria usaremos el comando ’s’. Éste evaluará el argumento para resolver la dirección nueva.

Un seek relativo puede realizarse asi:

```
> s+10                    > s++
> s $$+10                 > s--
```

Todos los seeks son registrados en un historial (s*) y podemos deshacer o rehacer los seeks con:

```
> s main # seek a la flag ’main’
> s-     # undo
> s+     # redo
```

Mediante el separador ’@’ realizaremos un desplazamiento temporal. El separador ’!’ nos permite realizar un cambio de tamaño de bloque temporal.

```
> x @ 0x80480
> x @ 0x80480!10
```

Escribiendo

Para editar un fichero debemos especificarlo con la flag ’−w’ que cargará el fichero en modo lectura y escritura.

Ensamblar un opcode

```
> wa mov eax, 33
```

Escribir una cadena de texto:

```
> w hello world\n
```

Escribir los contenidos de un fichero

```
> wf dump.bin
```

Escribir un numero en little endian

```
> wv 31337
> p8 4
```

Operaciones en el bloque (wo)

Usando ’wo’ podemos modificar el bloque actual aplicandole una operación a cada byte del conjunto.

```
> woa 1        # incrementa 1 a todos los bytes del bloque
> wox 0x89     # aplicar xor con 0x89 sobre el bloque
> wox 0x89     # aplicar xor con 0x89 sobre el bloque
```

Para generar secuencias de bytes repetidos por todo el bloque:

```
> wb 01020304 # llena el bloque con esta secuencia
```

Copy–Paste

El comando ’y’ nos permite hacer ’yank & paste’ o dicho de otra forma: Copiar y pegar los bytes deseados.

```
> w Hello World
> y 16
> yp
Hello World
> ps@16
Hello World
> yy@16
> ps@16
> x 32
```

Tuberias

Es posible usar pipes ’|’ para canalizar la salida del comando que ejecutemos a la entrada del programa de shell de la derecha.

```
> pd | grep push
```

En este caso podemos reemplazar el ’grep’ por el operador ’~’ que implementa un grep interno sin necesidad de ejecutar comandos de sistema (más portable).

```
> pd~push
```

El grep interno nos permite contar las lineas y realizar greps por columnas o filas.

```
> ~??          # muestra la ayuda del grep interno
> pd~?         # 64 <- el numero de lines mostradas
> pi~[0]       # muestra el mnemonico de cada opcode desensamblado
> pi~call[1] # muestra las direcciones de destino de los calls
> pi~call:1  # muestra el segundo call
```

Interpretando comandos

Muchos comandos aceptan el subcomando '*', que, al igual que los programas con la flag −r muestran los datos en formato de comandos entendibles por radare.

Podemos interpretar la salida de un comando usando el prefijo '.', por ejemplo:

```
> .!rabin2 −rs $FILE
> .is*
```

Tambien podemos usar este comando para ejecutar scripts externos:

```
> . script.rsc
```

Macros

Las macros son unidades de scripting construidas por una lista de comandos separados por comas, es posible pasarle argumentos y evaluar las variables $0, $1, ..

```
> (hola msg,?e hello $0,p8 16)
> .(hola world)
hello world
```

Con '@@' podemos ejecutar un comando para cada flag que encaje con un patrón.

```
> .(test) @@ hit*
```

Configuración

Las opciones son accesibles con el comando ’e’ que evalua una expresion para leer y definir el valor de una entrada de la configuración.

Se clasifican por dominios y pueden contener texto, numeros o booleanos.

```
e               # listar todas las variables
e asm           # listar las variables del dominio ’asm’
e??             # listar todas las variables y sus descripciones
e?asm.arch      # muestra la descripcion de la variable
e asm.arch      # muestra el valor de la variable
e asm.bytes=0   # tambien es válido usar ’e asm.bytes=false’
? ${asm.bits}   # mostrar el valor numerico de la variable asm.bits
```

Estos comandos pueden usarse desde el fichero ’~/.radare2rc’. Y así definir una configuración predefinida en el arranque. Otra forma común de usarlo es mediante el uso de proyectos o incluyendo scripts con ’r2 −i’.

Banderas

Las banderas (flags), son marcas que dejamos en el espacio de io. Estan compuestas por un nombre, una posición, una longitud y un comentario opcional.

Las flags se organizan dentro de flagspaces. El comando ’fs’ nos permite listar, crear y seleccionar el flagspace actual.

```
> f blah=0x80480400
> fd 0x80480404
blah+4
```

Las flags pueden ser accedidas por nombre en las expresiones:

```
> f loop_begin=0x8048449
> s loop_legin
```

Exportando y importando las flags

```
> f* > flags.txt
> . flags.txt
```

Modo visual

El modo visual nos ofrece una cómoda forma de interactuar con radare mediante keybindings.

```
> V
```

Para movernos utilizaremos las teclas ’hjkl’. Aunque las flechas tambien funcionan, es recomendable acostumbrarse a usarlas. En mayusculas nos moveremos más rapido (’HJKL’).

Al igual que en la shell. La tecla ’?’ nos ofrecerá la ayuda.

Teclas

Algunas de las acciones más comunes:

0–9	saltar al jmp/call número..
p P	cambiar entre modos de print (cmd.visual)
y Y	copy paste
:	introducir comando
t	movernos por las flags
e	cambiar las opciones
g G	ir al principio y final del fichero
c	activar/desactivar el modo cursor
i	pasaremos a modo inserción. (cursor+tab)
C	activar/desactivar los colores
B	activa/desactiva el auto blocksize
+ –	inc/dec el byte (cursor) o bloque
mm ’’	marcar posicion y saltar a ella

Escritura visual

El modo visual nos ofrece varias formas de escribir bytes, strings o codigo ensamblador.

- ’i’nsertar text o hexpairs
- usar + y − para modificar el valor de un byte
- usando ’y’ y ’Y’ para copiar y pegar bytes
- modo ’c’ursor para seleccionar un offset o rango

Mediante el comando VA podremos ensamblar codigo de forma dinamica. Tal como escribimos o borramos el codigo se ensamblará y se mostrará en el buffer. Esta funcionalidad puede sernos muy util cuando vayamos a parchear codigo ya existente.

La tecla ’A’ nos abre un menú especial para ensamblar de modo interactivo.

Ejercicios

Para asimilar con conceptos previamente expuestos realizaremos algunos ejercicios. No dudéis en preguntar cualquier duda que tengáis.

− solucionar crackme parcheando

− ensamblar en modo visual

Introducción al reversing con r2

pancake <pancake@nopcode.org>

(4/6)

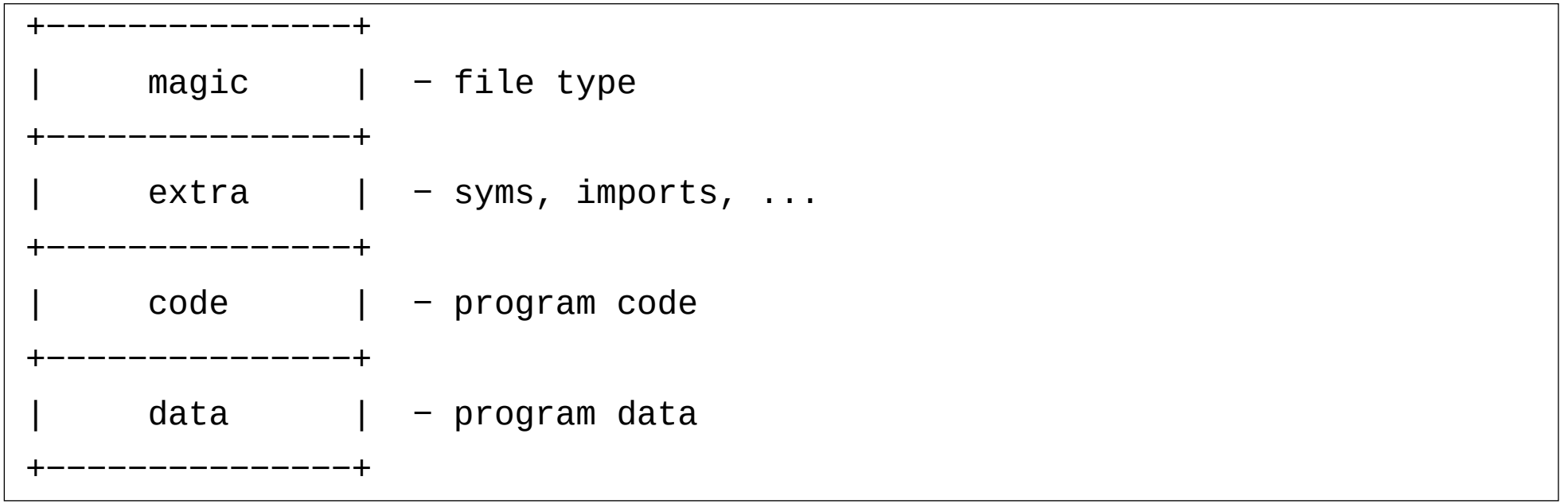
Analizando binarios

Una de los usos más interesantes de radare es para realizar analisis estatico sobre programas (independientemente de sistema operativo y arquitectura). Esto nos permitirá comprender mejor el funcionamiento del programa que queramos analizar.

El motor de análisis de radare permite generar grafos de bloques básicos, definir funciones, identificar saltos, referencias a variables, incremento de la pila, argumentos...

Analizando cabeceras

Los programas se estructuran de la siguiente forma:



rabin2 nos permite recoger toda esta información de forma generica, sin importar el formato del fichero (siempre que este soportado) Pero tambien podemos acceder a ella a través del comando ’i’ de la shell de r2.

Mostrando información del binario

Aqui algunos ejemplos de como recoger información del programa.

```
$ rabin2 -I foo
$ rabin2 -rI foo
$ rabin2 -s foo
```

Usando el comando ’i’ podemos importar la información del binario.

```
> .ia*
```

```
io : open file              ic : classes
ia : show all info          id : debug info
ie : entrypoints            ih : headers
ii : imports                iI : file info
is : symbols                iS : sections
iz : strings                iaj: json
```

Mostrando datos con formato

Mediante el comando ’pf’ definiremos una cadena de formato la cual queremos usar para visualizar los datos del bloque actual.

Nos permite especificar el endian, el tipo de variable, nombre de la variable, generar arrays de estructuras, punteros, ...

Usaremos el comando ’C’ para definir un formato que va a ser usado cuando se muestre el desensamblado.

```
pf?       # mostrar la ayuda
Cf 12 3xxi id
```

Añadir comentarios

En ocasiones, cuando estemos analizando un programa grande, nos será de gran ayuda poder dejar anotaciones al lado del desensamblado.

Los comentarios pueden estar asociados a un offset (CC) o a una flag (fC).

```
> CC good boy @ main+0x80
> fC bad boy @ sym.fail
```

Estas anotaciones pueden importarse y exportarse usando proyectos o pipes.

```
$ r2 -p test /bin/ls    # crear proyecto nuevo
$ r2 -p test            # cargar proyecto
> CC* > comments.txt    # exportar comentarios
> . comments.txt        # importar comentarios
```

Desensamblando

Existen varias formas de desensamblar (pd?) y el formato puede configurarse mediante ’e asm.’.

```
> pi 4          # desensamblar 4 instrucciones
> pd 4          # igual que el anterior, pero más completo
> pDi 4         #
> e asm.pseudo = true
> e asm.arch = arm
> e asm.bits = 64
* Cx
```

Ficheros mágicos

El comando ’file’ de *NIX nos permite identificar el tipo de archive comparando la cabecera contra una base de datos de definiciones (magic database).

r2 incorpora en r_magic la implementación de OpenBSD y podemos usara con el comando ’pm’ para identificar el tipo de dato en el cual nos encontramos.

```
> pm
> pm elf
> / ELF
```

* ejercicio: buscar un elfo en un pajar

Cálculos estadisticos

El analisis de datos pretende ofrecernos agregar la información cruda en resultados utiles para el reverser.

− ’pz’ zoom view, configurable mediante zoom.* , ’pz?’ para ayuda
− ’p=’ muestra graficas de barras de entropia y numero de caracteres alfanumericos
− rahash2 −a entropy −B −b 512 /bin/ls

Entropia:
- cero o muy bajo: (vacio)
- medio de entropia: texto plano, datos
- alto: codigo
- muy alto: comprimido o cifrado

Buscando cadenas

El comando ’/’ realiza busquedas de strings o cadenas binarias dentro de un rango de offsets

Comparando datos

- cargar 2 archivos a la vez (¡o maps!)
- comparando en memoria

Analizando opcodes

Podemos extraer información de los opcodes mediante el comando ’ao’ o ’a8’. Veamos

```
> pi 1
call dword 0x7fff5fc013b3
> ao 1
addr: 0x7fff5fc0104d
size: 5
type: 8 (call)
eob: 0
jump: 0x7fff5fc013b3
fail: 0x7fff5fc01052
```

En caso de no querer analizar el opcode de la posición actual podemos especificar los bytes del opcode para que este sea analizado.

```
> a8 c3cc90 @ 0x80480240
```

Analizando codigo

El motor de análisis de radare permite buscar y definir funciones. Podemos renombrarlas y cambiar sus propiedades mediante los subcomandos de ’af’.

Estos son algunos de los comandos que usaremos:

```
> aa          # analizar desde entrypoint y simbolos
> af          # analizar desde el offset actual
> afr         # renombrar función
> af-$$       # borrar funcion en el offset actual
> pdf         # desensambla la función entera
> af        # analyzar función en el offset actual
> afi $$    # mostrar informacion de la funcion
> pdf       # desensamblar solo la función
```

Generando un grafo

A partir de la información analizada de las funciones podemos generar grafos en formato Dot que nos muestren los caminos de ejecución.

```
> af
> ag $$ > a.dot
> agv $$
> !dot −Tpng −o a.png a.dot
```

Tambien podemos generar un grafo de las llamadas entre funciones (callgraph)

```
> agc $$ > a.dot
```

* nota * con ’dtc/dtg’ podemos realizar runtime callgraphs.

Referencias

Buscar en que parte del programa se accede a cierta string, es un caso de uso bastante común en el reversing. Desensamblado con ’pd’ se nos mostrarán las referencias de codigo y datos:

Si no queremos depender del motor de code analysis, podemos realizar una busqueda que encaje con el opcode desensamblado y que referencie la dirección de esta.

```
> s main
> af
> ? str.HelloWorld
0x100000f2e
> pd~0x100000f2e
```

* ejercicio parchear crackme con xrefs *

Extra: Hints

El usuario puede modificar la información que devuelve el motor de análisis usando el comando ’ah’. De esta forma podemos corregir errores del desensamblador o redefinir los mnemonicos a mano.

```
> ahl 10             # definir la longitud del opcode
> aho push GOODBOY   # cambiar el opcode en el desensamblado
> aha arm 512        # especificar asm.arch en los siguientes 512 bytes
> ahb
```

Redefinir la string del opcode puede sernos util para mejorar la legibilidad.

* ejercicio * files/antidis

Y más...

Las próximas versiones de radare2 pretenden mejorar cuantitativamente el análisis de código estático. Con representaciones completas y multi−arquitectura de todos los opcodes, mejor rendimiento y menor consumo de memoria.

Aunque el soporte actual no nos permita emular código, es suficiente para muchas tareas.

Si teneis preguntas... Ahora es vuestro momento!

Introducción al reversing con r2

pancake <pancake@nopcode.org>

(5/6)

Depurando programas

Una vez asimilados los conceptos de como usar radare como editor hexadecimal y analizador de codigo, veremos que podremos aplicar los mismos conocimientos para depurar programas.

Durante este apartado usaremos los subcomandos de ’d’ que nos permitiran controlar el debugger.

Hay que tener en cuenta que la API de debug de radare es independiente de plataforma, al igual que el resto de apis. Esto nos permite automatizar la depuración (unpacking, debugging, fuzzing) con scripts o programas escritos en C/Vala..

Iniciando el proceso

Para empezar debemos lanzar un procesos o atacharnos a ellos. Cabe destacar que depurar en radare implica no solo usar el plugin de IO correspondiente para que pueda leer/escribir memoria, sino tambien configurar los handlers, definir pid, realizar el fork, etc..

Todo esto queda simplificado con una sola flag: ’−d’

```
$ r2 −d ls                  $ r2 −d ./ls
$ r2 −d 1924                $ r2 −d ‘pidof firefox‘
$ r2 ptrace://20138         $ r2 −d "ls /"
```

Dentro de la shell, podremos usar ’dp’ para seleccionar o listar el proceso que estamos depurando, sus hijos y sus hilos de ejecución.

Inicio avanzado

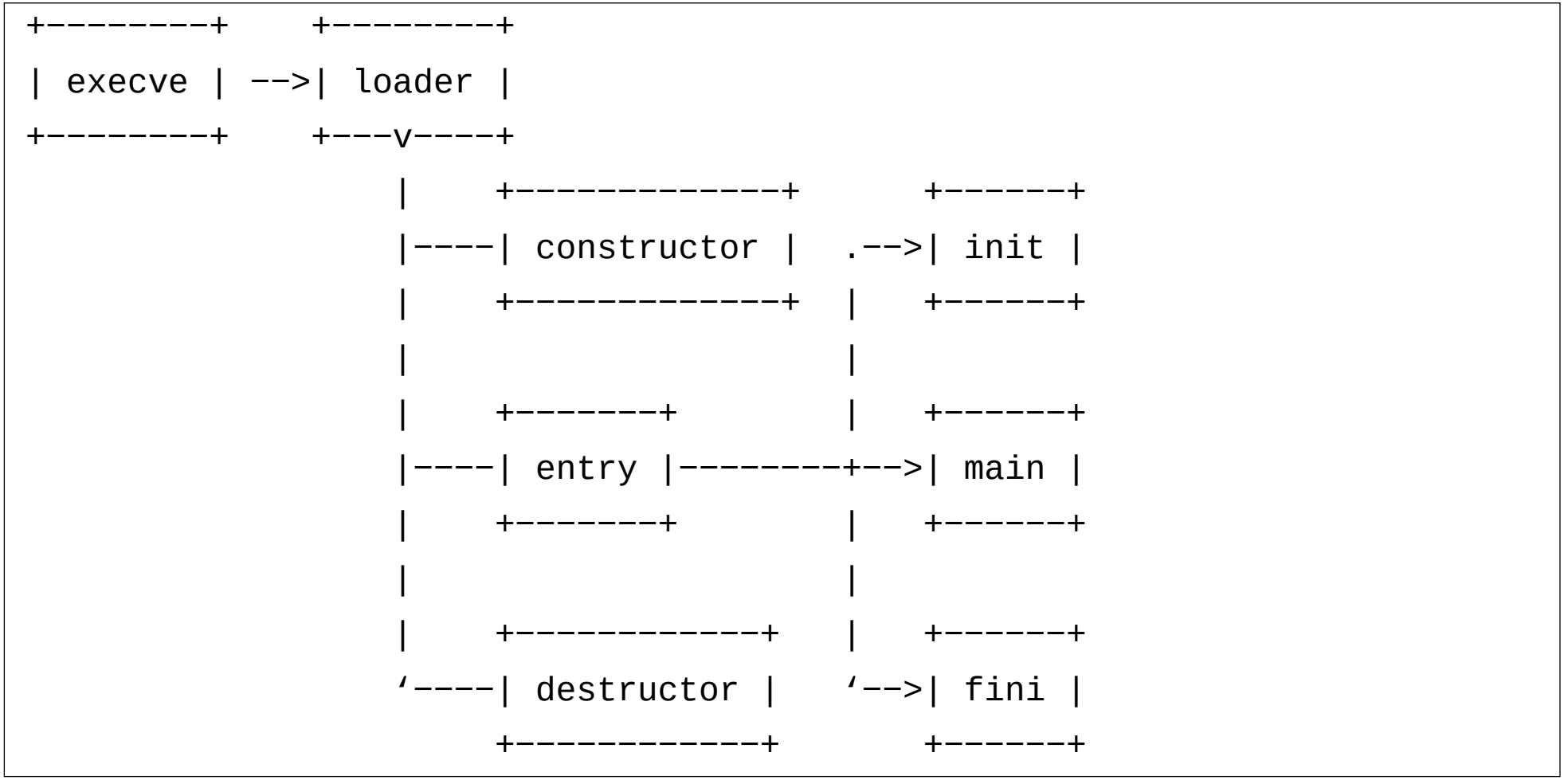
A veces nos interesará lanzar los programas más de una vez con parametros similares o con un entorno predefinido. Radare nos ofrece la herramient ’rarun2’ que nos permite definir varios parametros para lanzar el proceso:

- chdir
- chroot
- port binding
- diferente UID/GID
- precargar librerias
- variables de entorno
- stdin/stdout desde ficheros
- detener despues de N segundos

```
$ rarun2 listen=8080 program=/bin/sh
```

Bootstrap

Asi es como el sistema operativo carga los programas. Utilizando la variable dbg.bep podemos escoger en que punto detenernos como punto de partida.



Break entry..

Para lanzar el depurador deberemos usar la flag ’−d’.

```
$ r2 −d ./a.out   # depurando un programa en el directorio actual.
$ r2 −d ls        # iniciando la depuración de ’ls’ en $PATH
$ r2 −d ’ls /’    # pasando argumentos al programa
```

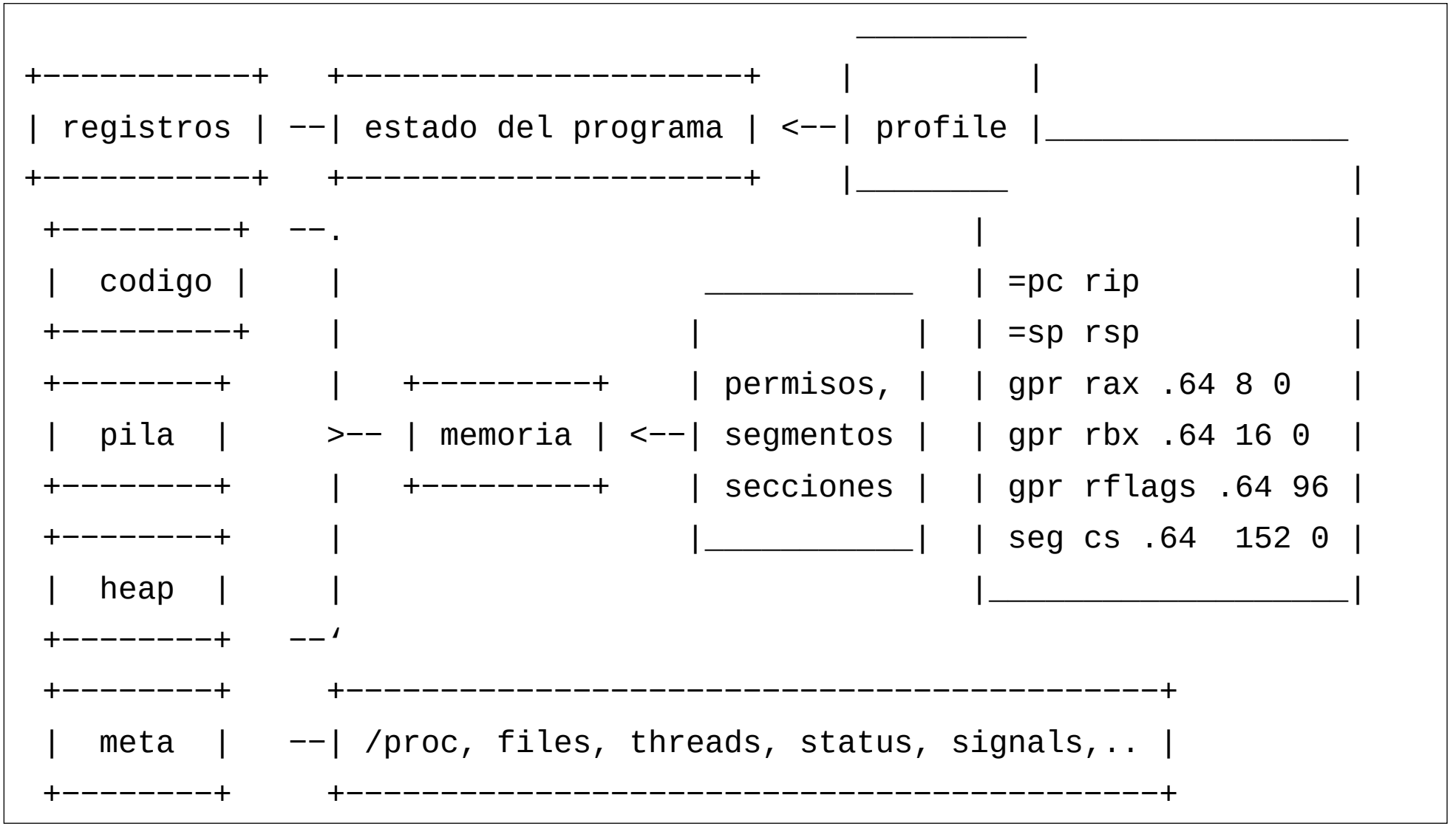
Escojamos donde queremos empezar a depurar con la variable dbg.bep:

```
$ r2 −e dbg.bep=loader −d ls
$ r2 −e dbg.bep=entry −d ls
$ r2 −e dbg.bep=main −d ls
$ r2 −e dbg.bep=0x8048620 −d ls
```

Seria un equivalente a:

```
$ r2 −c’dcu main’ −d ls
```

Elementos de un programa



Paso a paso

Si la plataforma no soporta stepping (mips) deberemos usar dbg.swstep

- instrucción a instrucción (ds)

```
0x8049520  eip:  mov eax, 33  ----.  single step
0x8049525        call 0x80497b8 <-‘
```

- saltando calls (dso)

```
0x804951d        push ecx  ----------.   step over
0x8049520        mov eax, 33  ---. <-‘
0x8049525  eip:  call 0x80497b8   )      step over
0x8049520        cmp eax, 0    <-‘
```

- ignorar N instrucciones (dss)

```
0x8049525  eip:  call 0x80497b8 -.
0x8049528        cmp eax, 0       |
0x8049528        jz 0x8049712     |  skip 5
0x8049525        push 0x804ab70   |
0x8049520        cmp eax, 0    <-‘
```

Mapas de memoria

Con ’dm’ podemos ver y manipular los mapas de memoria del proceso.

Esto puede sernos de gran utilidad para entender que librerias estan cargadas, en que punto de ejecución nos encontramos y usar la MMU para detener la ejecución cuando el programa quiera leer/escribir/ejecutar cierta página de memoria (4096)

En el entrypoint del programa veremos que las librerias de las que depende no están aún cargadas. Deberemos detenernos en main o en alguna syscall para tenerlas listas.

Con ’dmd’ y ’dml’ podemos volcar a disco y recargar el segmento de memoria actual.

* ejemplo *

Registros

El programa (thread) mantiene su estado en los registros. Estos pueden ser manipulados con el comando ’dr’.

```
> dr        # mostrar registros
> dr=       # ’’ en horizontal
> .dr*      # cargamos las flags
> dr eip=0x803840
> dr eax=0 # cambiamos el valor
> dr?rax    # leer el valor de rax
```

Los registros se mapean en memoria y se usa un profile para parsear sus valores.

```
> drp       # muestra el profile
```

Continuar la ejecución

Con el comando ’dc’ reanudaremos la ejecución del proceso.

- continuar hasta que termine o salte una excepcion (dc)
- ’’ hasta un punto (dcu)
- ’’ hasta una syscall (dcs)
- ’’ hasta que se haga un fork (dcf)
- ’’ hasta que se ejecute un call (dcc)
- ’’ hasta llegar al programa (dcp)

```
> dcu main
> dcs write
```

Breakpoints

Para definir puntos de parada usaremos el comando ’db’.

```
> db main       # añadir breakpoint en main
> dc            # continuar ejecución
> db-main       # eliminar breakpoint
```

Con el comando ’dsi’ podemos definir condiciones de parada más complejas.

```
> "dsi eax==3,ecx>0"
```

NOTA: Aún no es posible usar los breakpoints por hardware tal como ya hacía radare1. Esto estará implementado para antes de r2-1.0

Backtrace

Con el comando ’dbt’ analizaremos la pila en busca de direcciones de codigo siguiendo las estructuras de los ’stack frames’ con tal de saber desde donde venimos.

En casos de corrupcion de pila, puede que esta información no nos sea muy util, asi que podemos especificarle un numero puntero base como parámetro.

Asimismo, el comando ’ad @ ebp’ analizará la pila y nos mostrará qué tipo de datos hay y cuales son punteros a datos, strings, etc..

```
$ r2 -d ls
> ad@esp
> adk
```

Depurando en modo visual

El modo visual con el depurador nos mostrará el estado de los registros y la pila.

Tendremos a nuestra disposición nuevos keybindings:

```
.    seek program counter
s    ejecutar una instrucción
S    ejecutar una instrucion o parar después del call
b    definir/borrar breakpoint en el cursor
```

Podemos definir comandos a ejecutar con las teclas de función mediante las variables ’e key.f..’

Para acabar...

El debugger de r2 no nos ofrece un control tan completo como radare1.. de momento, pero espero poder dedicarle algo de tiempo y importar más código de radare1 para añadir soporte de inyección de código, breakpoints por hardware, manipulacion de filedescriptors, etc..

Al igual que el análisis; las funcionalidades actuales són más que suficientes para la mayoria de casos de uso que nos podamos encontrar.

En caso contrario siempre podemos usar radare1 :)

Ejercicios

Vamos a realizar algún ejercicio práctico usando el debugger para solucionar algunos problemas que he preparado para la ocasión...

* solucionar el crackme con el depurador

* saltar trucos anti−depuración

Introducción al reversing con r2

pancake <pancake@nopcode.org>

(6/6)

Interfaz gráfica y cloud

Despues de haber escrito unas 5 interfaces graficas distintas para radare y viendo cómo de dificil es conseguir desarrolladores que quieran implicarse en programar interficies gráficas decidí apostar por la web.

- HTML/CSS/JS son tecnologias muy comunes y estandarizadas
- adaptado a multiples tamaños de pantallas (responsive design)
- podemos lanzar un servidor web de radare en un embedido y depurar desde el desktop
- multiples clientes trabajando a la vez sobre la misma sesión

gradare, bokken, ragui, r2w, r2w2...

El servidor web

Radare viene con una implementación simplista de un client y un servidor http.

- HTTP 1.0 singlethread
- 600LOC
- get/post
- upload files
- ajax
- servir ficheros estaticos
- ejecutar comandos en /cmd/...

Y la forma de lanzarlo es utilizando '='. Este comando nos permite conectar radare con otros radares en local o remoto mediante los protocolos rap o http.

La interfaz es html estatico, y los contenidos se generan dinamicamente con peticiones ajax a /cmd/ para extraer información del core.

Iniciando el servidor web

Para arrancar el servidor web, podemos

```
$ r2 -qc=h /bin/ls          # carga el servidor
$ r2 -qc'=h 9999' /bin/ls   # escucha en el puerto 9999
$ r2 -qc=H /bin/ls          # carga el servidor y el navegador
```

Para conectarnos:

```
$ firefox http://localhost:9090
$ r2 -C localhost:9090
$ r2 -c=+http://localhost:9090/cmd/
$ echo x | r2 -C localhost:9090
> =
```

JSON

Recientemente se está añadiendo el subcomando ’j’ en varios comandos. Funciona de forma parecida a ’*’, sólo que en vez de mostrar los datos en forma de comandos de radare, se muestra en formato JSON.

De esta forma es muy fácil acceder a los datos de radare desde javascript.

```
> iIj
{"type":"mach0","class":"MACH064","endian":"little","machine":
"x86 64 all","arch":"x86","os":"darwin","lang":"c","pic":false,
"va":true,"bits":64,"stripped":false,"static":false,"linenums":
false,"syms":false,"relocs":false}
```

Scripting

Python y Javascript són los lenguajes de scripting oficiales de radare2, aunque existen bindings para muchos otros lenguajes.

La forma de interactuar desde javascript es haciendo peticiones AJAX a /cmd/<...> y el output que seria el output del comando, podemos parsearlo cómo texto o json.

* bindings *

Futuro

Es bastante probable que el desarrollo del GUI continue siendo HTML5, aunque no necesariamente usando el framework actual (enyo).

El cloud (cloud.rada.re) nos permitirá publicar sesiones de radare, para que otros usuarios puedan colaborar en tiempo real.

Actualmente sirve cómo sitio de demostración.

Demo y ejercicios

* go *

EOF