Una Brevísima Introducción a la Ingeniería Inversa (Parte I)

Joxean Koret

Una Brevísima Introducción

- Introducción
- Tareas profesionales
- Recursos para empezar

Introducción

- ¿Qué es la ingeniería inversa? Wikipedia:
  - *”El objetivo de la ingeniería inversa es obtener información a partir de un producto accesible al público(*), con el fin de determinar de qué está hecho, qué lo hace funcionar y cómo fue fabricado.”*
  - *”La ingeniería inversa es un método de resolución. Aplicar ingeniería inversa a algo supone profundizar en el estudio de su funcionamiento, hasta el punto de que podamos llegar a entender, modificar y mejorar dicho modo de funcionamiento.”*
  - *”En el caso concreto del software, se conoce por ingeniería inversa a la actividad que se ocupa de descubrir cómo funciona un programa, función o característica de cuyo código fuente no se dispone, hasta el punto de poder modificar ese código o generar código propio que cumpla las mismas funciones.”*

** Sería más correcto decir ”accesible” sin especificar a quién...*

Pequeña Aclaración

- Existen diferentes tipos de ingeniería inversa.
- En el caso de esta charla, me ceñiré a la Ingeniería Inversa de Software, generalmente llamada SRE, siglas en Inglés de ”Software Reverse Engineering”.

Símiles

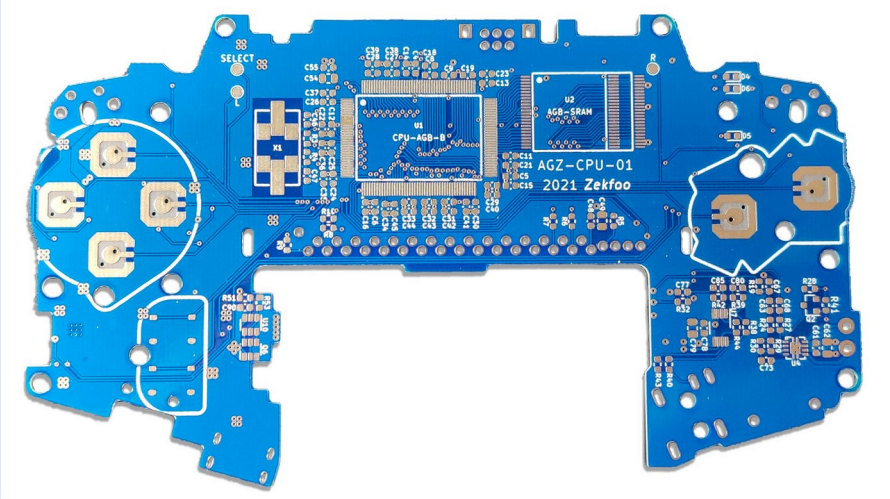
- 2 ejemplos de andar por casa para entender un poco qué es y para qué vale la ingeniería inversa:
  - Imaginemos que vamos a un restaurante y comemos un menú. Podríamos hacer ingeniería inversa de los platos para sacar los ingredientes **y** cómo se han cocinado los mismos.
  - De este modo podríamos crearnos nuestra receta para poder hacerla a nuestro gusto, con los mismos ingredientes y/o procesos. O con otros diferentes.
- Naturalmente, esto es ingeniería inversa. No es reversing de software o hardware, pero reversing es.

Símiles

- Otro ejemplo:
  - Escuchamos una canción de una de nuestras bandas favoritas y queremos hacer nuestra propia versión.
  - Analizamos el tempo, el ritmo, las progresiones de acordes utilizadas, la tonalidad, la conducción de voces utilizada, los instrumentos utilizados, etc…
  - Cuando tenemos toda la información que necesitamos, hacemos una versión, un arreglo, de dicha canción.
- Esto es, efectivamente, ingeniería inversa también.

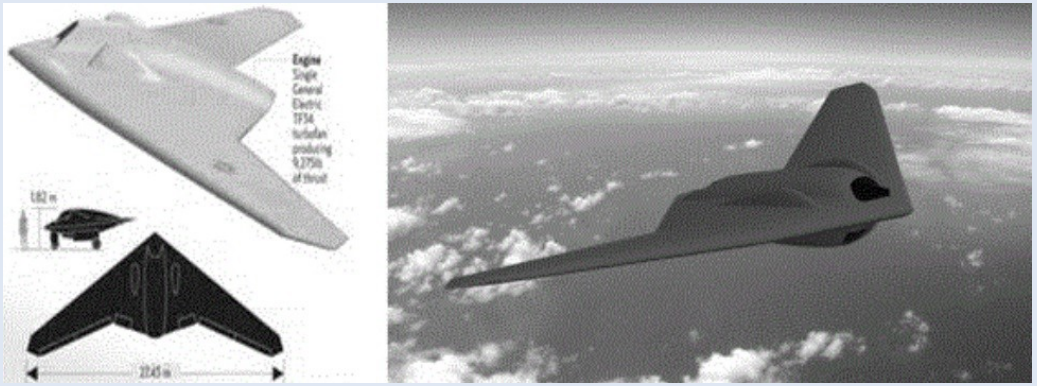
Legalidad

- ¿Es legal la ingeniería inversa?
  - Varía según el país.
  - Generalmente, si se dispone del producto legalmente, la respuesta es sí. De todos modos, repito, depende del país y es muy relativo.
  - http://en.wikipedia.org/wiki/Reverse_engineering#Legality



Legalidad

- Depende largamente del propósito pero, en ocasiones, hablar de legalidad no tiene sentido alguno.
- Un ejemplo: ¿Es legal que Irán haga ingeniería inversa al drone Estadounidense que ”se cayó” en Irán en el año 2011?
- http://en.wikipedia.org/wiki/Iran%E2%80%93U.S._RQ-170_incident
- ¿De acuerdo a qué legislación?

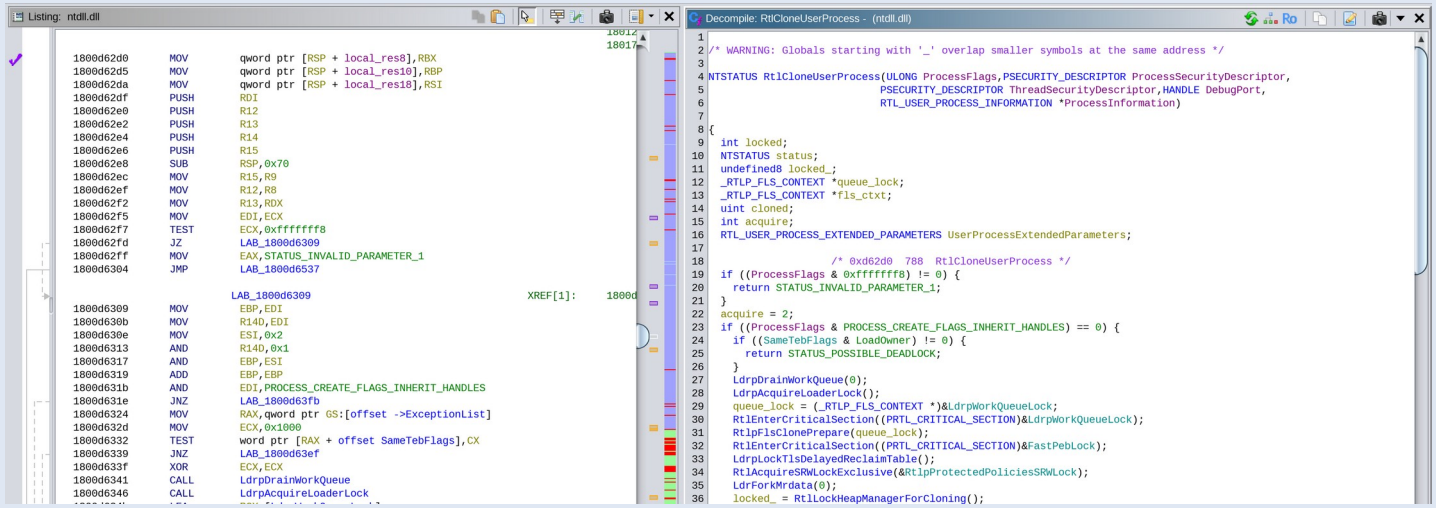


Métodos

- La Ingeniería Inversa se aplica de 2 (o 3) modos:
  - Usando Análisis Estático.
  - Usando Análisis Dinámico.
  - Usando una mezcla de ambas.

Análisis Estático

- El análisis estático es aquel que se hace leyendo ”el código” de dicho software, donde ”el código” puede ser código fuente propiamente, desensamblado o decompilado.



Análisis Dinámico

- A diferencia del análisis estático donde el software no se llega a ejecutar, el análisis dinámico se basa en ejecutar el software en un entorno más o menos controlado para analizar cómo se comporta.
- El software puede ser ejecutado en hardware real, en una máquina virtual o emulador (como Vmware, VirtualBox, Bochs o Qemu), o con emuladores embebibles (como Unicorn Engine).

¿Para qué se utiliza la ingeniería inversa?

- Profesionalmente:
  - Investigación de malware.
  - Desarrollo de vulnerabilidades y exploits.
  - Detección de plagio.
  - Compatibilidad con software y/o hardware propietario.
  - Modificación de software antiguo.
  - Análisis de productos de competidores.
  - Recuperación de código fuente.
  - Desarrollo de parches binarios.
  - Análisis de hardware y software foráneo.
  - Análisis forense.
- Hablemos de algunas de dichas tareas. Os daré mi opinión profesional de la mayoría de ellas.

Investigación de malware

Investigación de Malware

- Se conoce como ”malware” a cualquier software malicioso.
- Para poder determinar qué hace un software malicioso (por ejemplo, ”roba datos bancarios”) es necesario analizar dicho software, es decir, hacer ingeniería inversa del malware.
- El análisis de malware suele hacerse tanto de modo estático (leyendo el código fuente, desensamblado o decompilado sin llegar a ejecutar dicho software) como de modo dinámico (ejecutando dicho software para analizar su comportamiento).

Investigación de Malware

- Estos son algunos de los principales lugares donde es habitual hacer malware y análisis de malware, ambos, profesionalmente:
  - Empresas antivirus indiferentemente de cómo se autoproclamen en función del año que sea (”endpoint”, ”next gen”, etc).
  - SOC: Security Operations Centers. Básicamente, seguridad interna en grandes empresas de ámbitos tan diversos como, por ejemplo, transportes, bancos, cadenas de suministros varias, etc.
  - CERT: Computer Emergency Response Centers.
- Nota: CSIRT, CIRT, CERT y SOC suelen usarse para referirse a lo mismo y mucha gente los considera, básicamente, sinónimos con pequeños matices.

Investigación de Malware II

- Una variación del análisis de malware:
  - Análisis de cheats y videojuegos.
  - Más común de lo que os podéis pensar.
- Supongamos un ejemplo con un videojuego on-line multijugador inexistente como, Call of Nite.
  - Dicho juego se convierte en muy popular y empieza a haber gente que desea hacer trampas por ego o por dinero (torneos).
  - Uno o varios reversers analizan el videojuego para escribir cheats y que otros hagan trampas en el juego.
  - Uno o varios reversers analizan el cheat para ver cómo funciona, detectarlo, evitarlo, mitigarlo, etc...

Malware

- Veamos, brevemente, algún que otro caso curioso.
  - Stuxnet, 2005-2010.
  - Shamoon, 2012-2015.
  - WannaCry, 2017.

Stuxnet

- Malware gubernamental desarrollado conjuntamente por Estados Unidos e Israel.
  - Operation Olympic Games.
- Su finalidad era sabotear el programa nuclear Iraní.
  - Específicamente, la central nuclear de Natanz.
- Utilizó 4 exploits 0-days (cuatro) para propagarse.
- Un malware de altísima calidad y precio.

Shamoon

- En el año 2012 un ingeniero de IT de Saudi Aramco (una de las empresas petrolíferas más grande del mundo) recibió un mail con el que fue engañado (phishing) y un atacante consiguió acceso a la red interna.
- No mucho más tarde, unos 35.000 ordenadores fueron saboteados con un wiper (”borra cosas”). Dicha empresa petrolífera estuvo meses parada.
- El malware simplemente se copiaba por recursos de red una vez que una máquina estaba infectada, recopilaba una lista de ficheros, se comunicaba con un C&C, y una bomba lógica activaba el wiper, que borraba ”cosas” y el MBR.
- Las sospechas y rumores apuntan a Irán. Pero a saber.

WannaCry

- Simplificándolo mucho: un gusano con un ransomware.
- El gusano utilizaba una vulnerabilidad conocida (N-days) que era originalmente de la NSA, Eternal Blue. El componente ransomware no merece la pena ni mención.
- Inicialmente, muchos creían (yo incluído) que sería alguna mafia a la que se les fue de las manos durante el desarrollo (los gusanos son difícilmente controlables) y acabó infectando más de 300K ordenadores.
- Las sospechas y rumores, difícilmente verificables, sin embargo, apuntaron al muy popular grupo hackeril de Korea del Norte, The Lazarus Group.

Investigación de Malware

- Ventajas:
  - Relativamente sencillo de empezar.
  - Bien pagado, en general. Ambos casos. Los 2 lados.
  - El mundo del malware/cheats es cambiante, evoluciona, forzando a tener que aprender cosas nuevas.
- Desventajas:
  - Repetitivo.
  - Repetitivo.
  - La gran mayoría del malware es basura, siendo muy poco el malware que es gratificante analizar.
  - Los cheats también suelen ser repetitivos.
  - Programar cheats no es recomendable para tu salud.
  - Programar malware ”malicioso” te llevará a la cárcel.

Desarrollo de Vulnerabilidades

Desarrollo de Vulnerabilidades

- Uno de los campos donde se aplica la ingeniería inversa más comunes y también uno de los campos más…
  - Estresantes.
  - Gratificantes.
  - Estresantes.
  - Bien remunerados.
  - Estresantes.
  - Y causante de conflictos éticos y morales.

Vuln-dev

- ¿Qué es el desarrollo de vulnerabilidades?
  - Llamado en inglés habitualmente vuln-dev (VD) o vuln-research (VR).
- Resumiendo muy-mucho:
  - Buscar vulnerabilidades (0days o Ndays).
  - Determinar su explotabilidad.
  - Escribir exploits (común).
  - Escribir mitigaciones (opcional).
  - Escribir detecciones (opcional).
  - Escribir parches (incluso más opcional).

Vuln-dev: Ejemplo

- Pongamos un ejemplo:
  - Una empresa contrata a un reverser para buscar vulnerabilidades en un software, como Apple Safari.
  - El reverser utiliza el código fuente de WebKit y los binarios de Safari en Mac para buscar vulnerabilidades que puedan ser usadas para explotarlas remotamente simplemente visitando un enlace.
  - Tras encontrar una o varias vulnerabilidades (asumiremos vulnerabilidades de corrupción de memoria), el reverser pasará a escribir un exploit, lo cuál es una tarea que en la práctica totalidad de las ocasiones hoy en día significa ”trabajar en ensamblador haciendo sudokus con el heap”.
  - ...

Vuln-dev: Ejemplo

- Continuando con el ejemplo de un exploit para Safari:
  - Dicho exploit, probablemente, tenga que ser escrito 2 o hasta 3 veces.
  - Una vez para escritorio x86_64 (Intel).
  - Una vez para escritorio AArch64 (ARM).
  - Una vez más para iOS AArch64 (ARM).
  - Puede darse perfectamente el caso de que en una o varias de las plataformas dicha vulnerabilidad no sea explotable.
- Puede que, parcialmente, el exploit para la versión de escritorio y la versión de iPad/iPhone del exploit funcion igual, pero lo más probable es que no sean compatibles y necesiten muchos cambios.
- Ni que decir tiene cuánto va a cambiar un exploit para x86 en comparación a un exploit para ARM.

Vuln-dev: Las más jugosas

- En el ejemplo mencionado hemos asumido vulnerabilidades de corrupción de memoria:
  - Básicamente, algún tipo de problema como heap overflows que sobreescriben memoria que no debería ser escrita y que se puede utilizar para hacer ”cosas”.
- Sin embargo, existe un tipo de vulnerabilidad mucho mejor: las vulnerabilidades lógicas.
- Este tipo de vulnerabilidades suele ser más difícil de encontrar pero, cuando se encuentran, su explotabilidad suele ser del 100%, e incluso 100% multiplataforma.
  - Naturalmente, también son los exploits mejor pagados.

Vuln-dev

- Ventajas:
  - Gratificante. Tanto encontrar vulnerabilidades como explotarlas.
  - Muy-muy bien pagado.
- Desventajas:
  - Estresante. Es una montaña rusa.
  - Problemas éticos y morales: quienes más pagan son quienes quieren exploits para usarlos. Gobiernos.
    - Hay quienes corrigen las vulnerabilidades (ZDI), pero pagan cacahuetes en comparación.
  - ¿Contra quién o qué se usará dicho exploit? Nadie te lo va a decir. Igual mañana te lo encuentras en alguna campaña ”in-the-wild”.
  - Hay mucha demanda sobretodo para países OTAN, aliados, adversarios, países *altamente democráticos* de Oriente Medio, etc.

Plagio

Detección de Plagio

- Analizar si un software A es un plagio (una ”copia ilegítima”) de otro software B.
- Naturalmente, sin tener código fuente, solo se puede averiguar si hay plagio en estos casos:
  - Analizando dinámicamente el software y viendo si se comportan del mismo modo.
  - Analizando estáticamente el software y viendo si comparten ”mucho” código.
    - Por ejemplo, mensajes de error o depuración con errores sintaćticos/gramaticales compartidos en ambos softwares suelen ser buenos indicadores de plagio.

Detección de Plagio

- Ventajas:
  - Tarea relativamente sencilla *técnicamente*. Especialmente si el plagio es burdo.
- Desventajas:
  - Poco común.
  - Aburrido.
  - El trabajo técnico es relativamente fácil, demostrar que algo es un plagio no es trivial.
  - Es probable que sea necesario ir a declarar como perito a un juicio. No solo es trabajo técnico.

Compatibilidad con Software

”Compatibilidad” con software

- Esta es otra de las tareas más comunes: compatibilidad entre software.
- ¿Qué significa esto? Pues, por ejemplo, que LibreOffice pueda utilizar formatos de ficheros propietarios de Microsof Office.
- O que SAMBA, una implementación Open Source del protocolo SMB/CIFS, pueda comunicarse con servidores de red propietarios Windows.
- O para hacer add-ons comerciales para PowerPoint.
- O para hacer emuladores de videojuegos o computadoras.
- O para hacer nuevos motores de videojuegos utilizando los archivos de datos originales.

Compatibilidad con Software

- Ventajas:
  - Bastante divertido, a corto y medio plazo.
  - A largo plazo, se consigue tener un conocimiento superior incluso al de la gente que hace o hacía dicho software originalmente.
  - Relativamente bien pagado. Depende mucho del sitio.
- Desventajas:
  - Muchas ofertas ”oscuras”.
  - A largo plazo aburrido y monótono. Siempre es el mismo proyecto.
  - Debido a temas legales, puede ser obligatorio cambiarse de país o tener incluso clearance...

Modificación de Software Antiguo

Modificación de Software Antiguo

- Un caso hipotético:
  - Una empresa XX contrata a una empresa de software YY para hacerles un software. La empresa solo vende los binarios y se guardan el código fuente.
  - Al de unos años la empresa YY cierra.
  - La empresa XX quiere migrar de, digamos, Windows 2000 a Windows 11.
  - El software propietario que compraron no funciona hoy: no reconoce el hardware necesario. Por ejemplo.
  - ¿Qué pueden hacer? Contratar servicios de reversing. O comprar otro software.

Modificación de Software Antiguo

- Siguiendo el supuesto totalmente hipotético anterior:
  - Si el software que le vendieron estaba protegido (DRM, anti-copy, etc…) el o los reverers primero tendrán que crackear dicho software (eliminar las protecciones).
  - Después, deberán analizar qué APIs o métodos viejos utiliza dicho software para hacerle compatible con sistemas operativos más recientes.
  - Partes de dicho proceso serán escritas en puro ensamblador, y algunas partes en lenguajes de alto nivel.
  - …

Modificación de Software Antiguo

- Continuando con el ejemplo:
  - En función de la complejidad y de la calidad del trabajo realizado, el software anterior será más o menos modificado (cuanto menos mejor), y las nuevas partes (la nueva lógica) será escrita en lenguajes de alto nivel (como C++ o Rust).
  - Los reversers pueden decidir *inyectar* una DLL o Shared Object (.so) donde poner la nueva lógica, desde donde instalar ”hooks”, desde donde monitorizar eventos, etc...

Modificación de Software Antiguo

- Ventajas:
  - La primera vez es divertido.
  - Obliga a ser imaginativo.
- Desventajas:
  - Tedioso.
  - En ocasiones necesitas mirar documentación que ya no existe en ningún sitio, o necesitas compiladores de hace +20 años, etc…
  - A veces da la sensación de ser trabajo para nada.

REconstrucción de Código Fuente

REconstrucción de código fuente

- Otro caso hipotético que jamás ha pasado. En ningún sitio. Nunca.
  - Una empresa de software XXX produce el software SuperSoft.
  - Debido a un desastre, la empresa pierde todas las copias del código fuente SuperSoft.
  - O simplemente ya no saben dónde lo tienen.
- En este supuesto, la empresa tiene 2 opciones:
  - Reescribir el software desde cero.
  - Contratar servicios de reversing para reconstruir el proyecto.

REconstrucción de código fuente

- En este tipo de proyectos:
  - Se trabaja, naturalmente, exclusivamente con binarios.
  - Si hay muchísima suerte, el software se hizo en alguna tecnología que es más o menos decompilable correctamente, como Java o .NET.
  - Si no hay suerte a tal respecto pero aún se tiene algo de suerte, los binarios tienen símbolos con incluso variables locales que pueden ayudar mucho.
  - Si hay poca suerte solo hay binarios sin símbolos.
  - Si la suerte es nula, solo hay binarios sin símbolos y además están protegidos con algún software.
    - En un caso desastroso, el software está virtualizado.

REconstrucción de código fuente

- Ventajas:
  - Bien pagado.
- Desventajas:
  - Aburrido. Monótono. Tedioso.
  - Poco habitual.
  - Supuestamente, son proyectos de corta duración.
  - No es posible la recuperación de código fuente realmente, por lo que se fuerza a hacer una reconstrucción ciñéndose a una especificación ya existente (los binarios) sin mucha posibilidad a corregir fallos de diseño.

Recursos para empezar

Recursos para empezar

- ”Me interesa este tema. ¿Y cómo empiezo?”.
- Bien, pues no es muy sencillo, pero existen algunos recursos on-line, todos ellos en Inglés, que puedo recomendaros:
  - https://begin.re
  - https://crackmes.one
  - https://malwareunicorn.org/#/workshops
  - https://0xinfection.github.io/reversing/reversing-for-everyone.pdf

Recursos para empezar II

- Además de los mencionados recursos, tenéis la parte 2 de esta charla.
- No vamos a dar esa parte de la charla ahora mismo por problemas de tiempo, pero tenéis disponibles las slides con el contenido y explicaciones varias.
- Además, me podéis preguntar sin problema alguno:
  - @joxean@mastodon.social
  - joxean.piti@gmail.com

Fin de la parte I
¿Preguntas?

Eskerrik asko!

Una Brevísima Introducción a la Ingeniería Inversa (Parte II)

Joxean Koret

Herramientas más comunes

Herramientas más comunes

- General
  - Compiladores, intérpretes, monitores de procesos, máquinas virtuales, emuladores, etc.
  - Dependerán de nuestro ”target”: x86, arm, ppc, etc.
- Análisis dinámico
  - IDA Pro, GNU Debugger, x64dbg y Ghidra.
- Análisis estático
  - IDA Pro y Ghidra.

Fundamentos de Ingeniería Inversa

Puesta en marcha de un laboratorio de ingeniería inversa

¿Qué necesitaremos?

- Compiladores e intérpretes
  - Python (versión 3.X) y C/C++ (Clang/Gcc/Msvc).
- Debuggers
  - GDB, IDA Pro, Ghidra y x64dbg.
- Desensamblador y decompilador
  - IDA Pro y Ghidra.
- Máquinas Virtuales
  - VirtualBox, VMWare, QEmu.

Fundamentos de Ingeniería Inversa

Una brevísima introducción a la teoría de grafos
Análisis Estático

Teoría de grafos

- Brevísima iniciación

Teoría de grafos

- ¿Qué es un grafo? Wikipedia:
  - *”Un grafo es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto”.*
- Tipos de grafos
  - Dirigidos y no dirigidos.
  - En nuestro caso, nos interesan los dirigidos.

Teoría de grafos

- ¿Qué es un grafo dirigido? Un grafo en el que se conoce la dirección de relación entre nodos
  - Definición de andar por casa.
- Grafo dirigido:

- Grafo no dirigido:

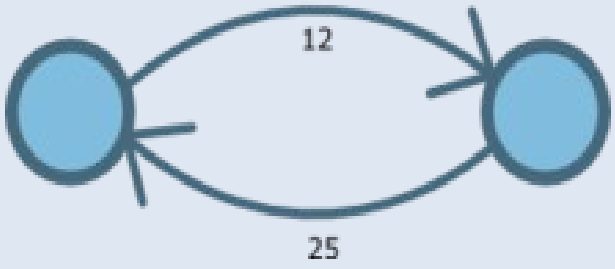
Teoría de grafos

- ¿Qué es un nodo (vértice)?
  - La unidad fundamental de la que están formados los grafos.
- ¿Qué es una arista?
  - Corresponde a la relación entre vértices.

Dos vértices y una arista que los une

Arista de un grafo dirigido

Aristas con pesos



Teoría de grafos

- ¿Qué tiene que ver la teoría de grafos con la ingeniería inversa de software?
  - Mucho, como vamos a ver ahora mismo...

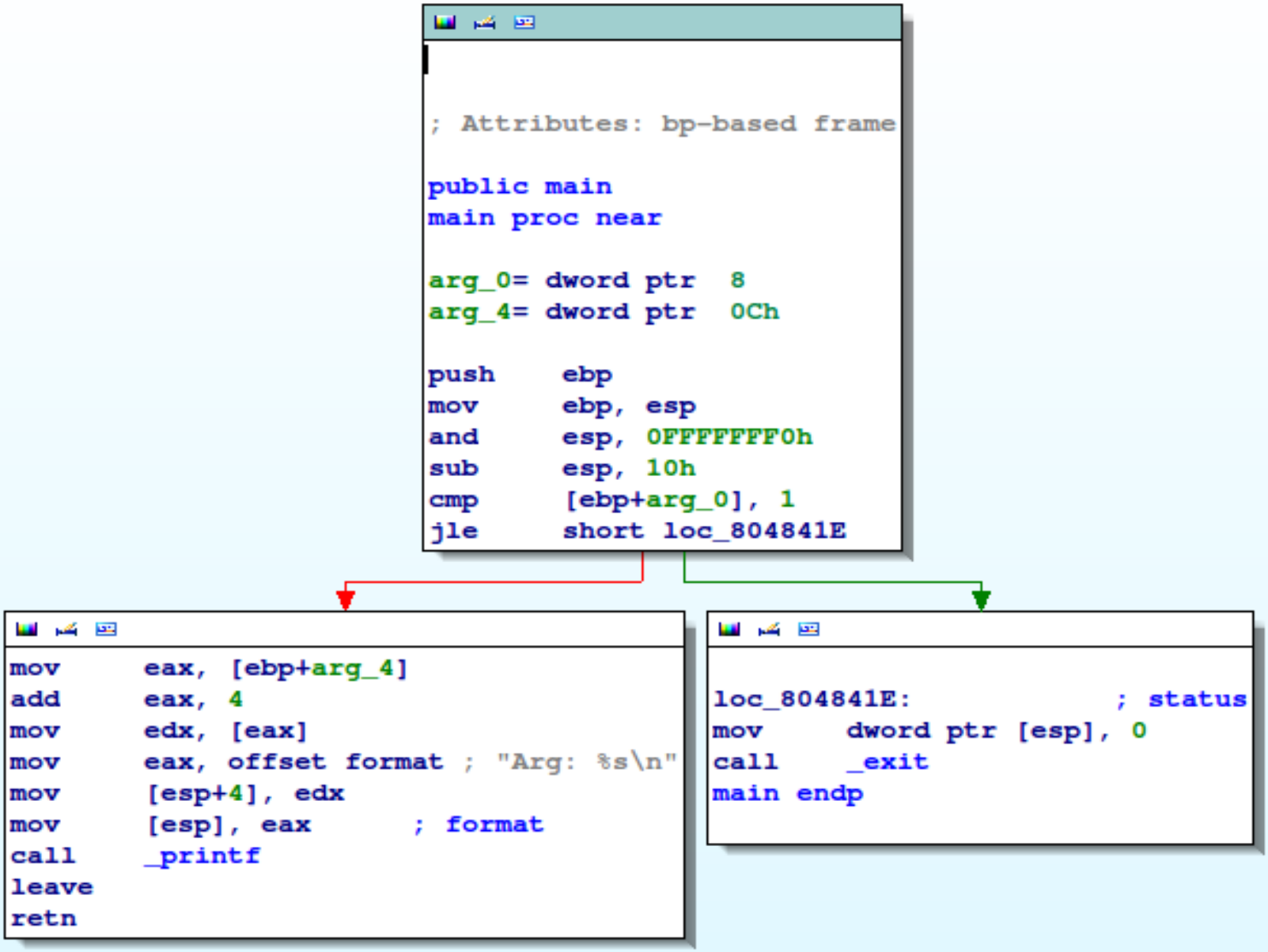
Teoría de grafos

- Una función o conjunto de funciones se puede representar como un grafo dirigido.
  - Call graph (en inglés), o grafo de llamadas:
    - Cada nodo es una función de un programa y cada arista una unión que determina que una función llama a otra.
  - Flow graph (en inglés), o grafo de flujo:
    - Cada nodo es un bloque básico de una función y cada arista determina el flujo de ejecución basado en tomas de decisiones.

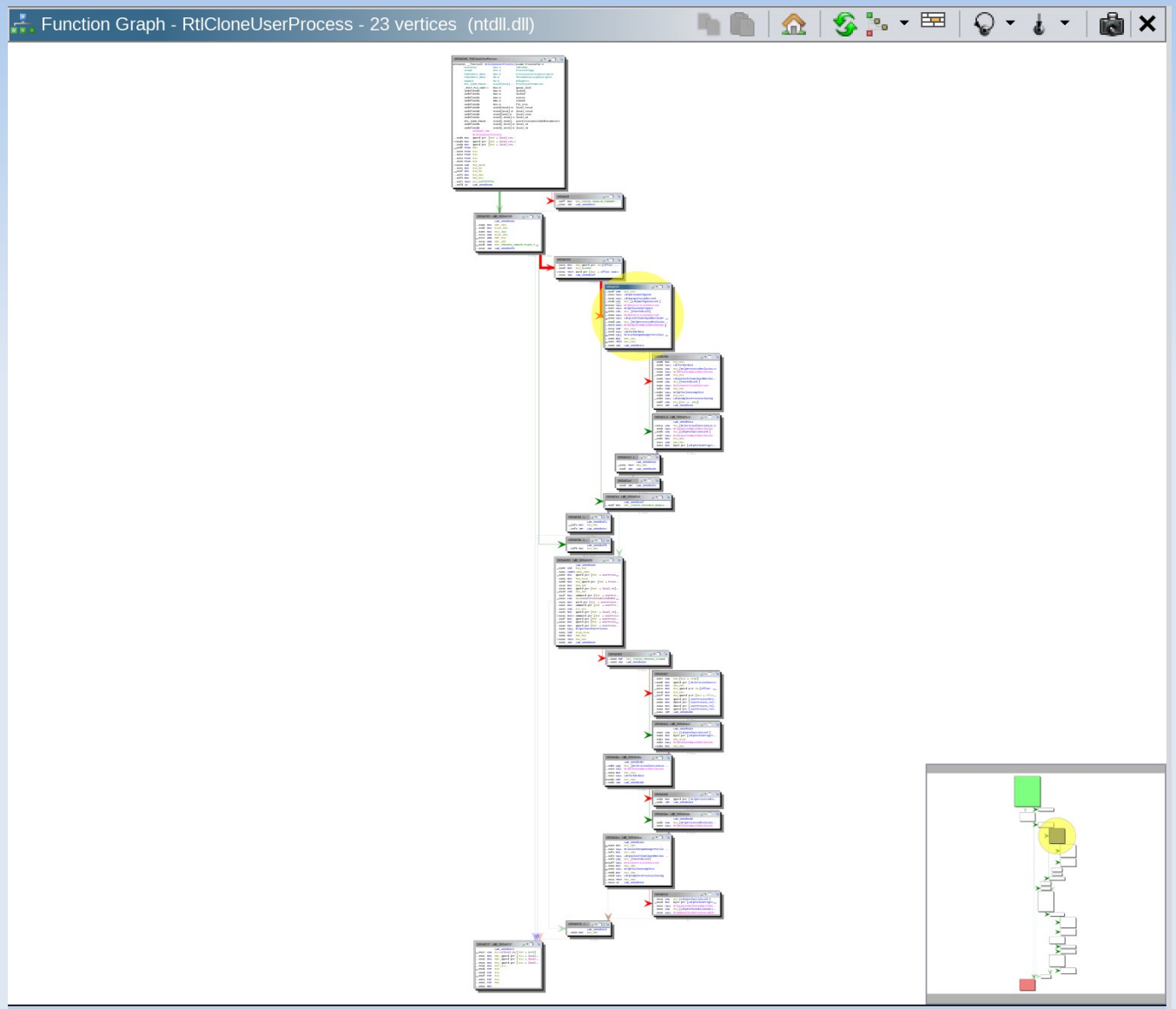
Teoría de grafos: Flowgraph

- Cada nodo es un bloque básico.
- ¿Qué es un bloque básico?
  - El conjunto de instrucciones contiguas hasta una toma de decisión.
- ¿Qué es una arista?
  - Determinan las relaciones entre bloques básicos, es decir, el flujo que tomará la ejecución del programa tras una toma de decisión.

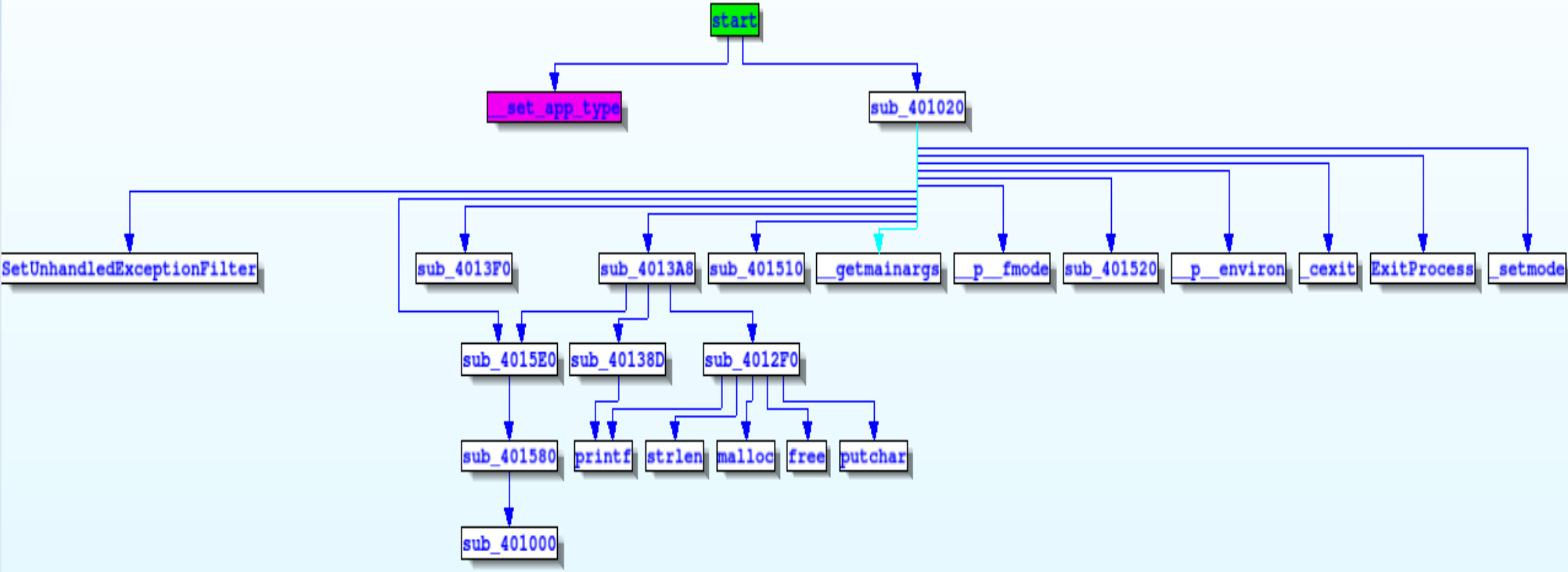
Teoría de grafos: Flow graph



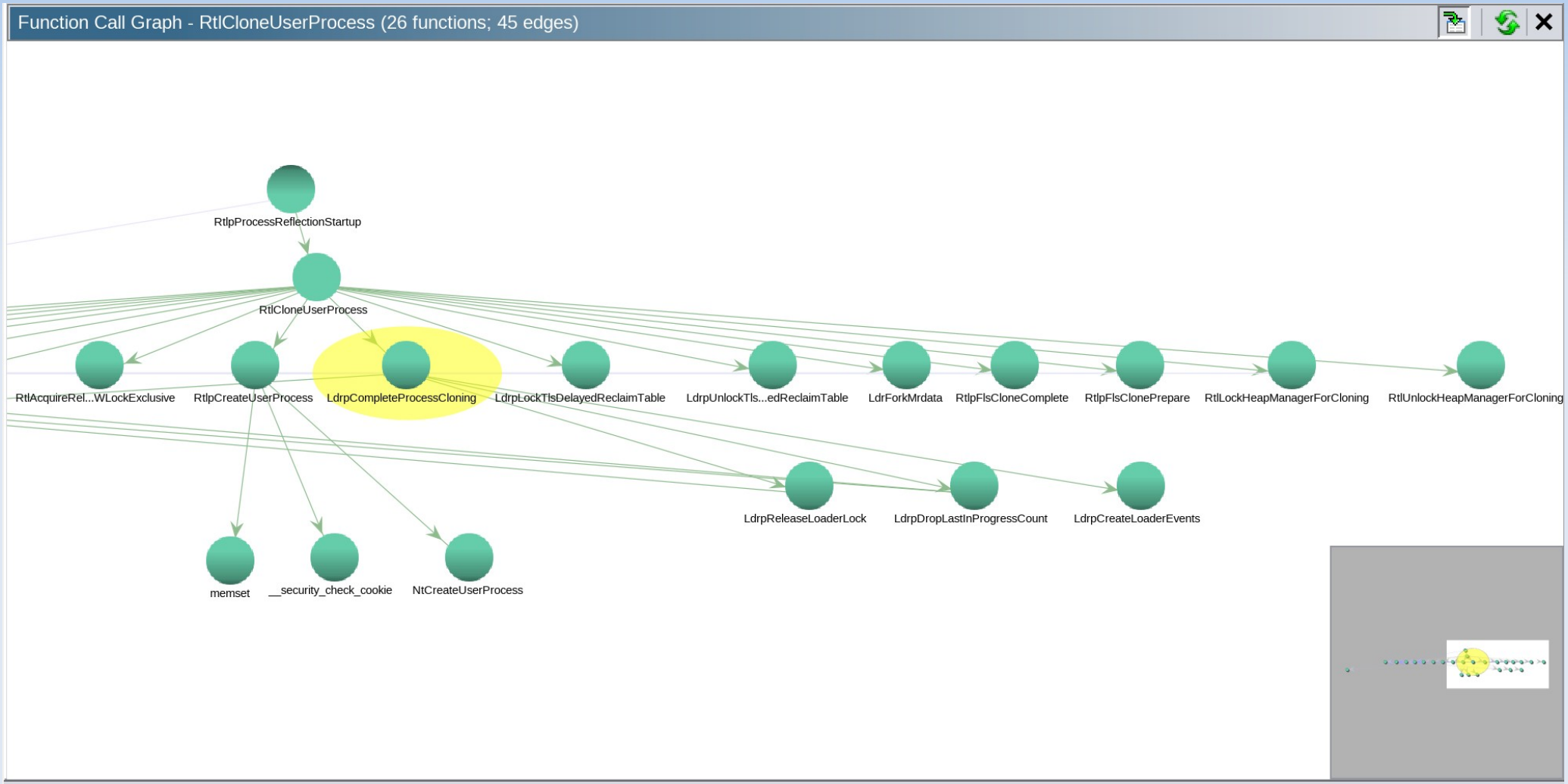
Teoría de grafos: Flow graph



Teoría de grafos: Call graph



Teoría de grafos: Call graph



Teoría de grafos

- Fin de la breve introducción
  - Preguntas? Dudas?
- Suficiente teoría?
  - Empecemos con la práctica ;)

Herramienta: IDA Pro

- ¿Qué versión de IDA puedo usar? ¿Tengo que pagar por una licencia? ¿Usar una versión pirata (NO)?
- IDA Pro es la que ha sido, de facto, siempre, la herramienta comercial para hacer RE de modo profesional. Su precio es altísimo.
- Pero, existe una versión freeware:
  - https://www.hex-rays.com/products/ida/support/download_freeware.shtml
  - La versión freeware actual trae debuggers y decompilers, y soporta Linux, Windows and Macos.

Herramienta: Ghidra

- Hoy en día la herramienta Open Source de facto para hacer RE, tanto de modo profesional como por deporte, es Ghidra:
  - https://ghidra-sre.org/
- Debería de funcionar en cualquier cosa donde Java 11 funcione.
- Podéis utilizar OpenJDK, ya que no solo está totalmente soportado sino que que los desarrolladores de Ghidra utilizan OpenJDK para trabajar.
- La versión actual en el momento que escribo estas líneas es la 11.0.2.

Y ahora sí, empecemos con la práctica...

Análisis Estático

- ¿Qué es? Extraído de wikipedia:
  - *”Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software”*
  - Es decir, el medio de averiguar como funciona un software sin llegar a ejecutar el mismo, solamente leyendo el código del que dispongamos.
    - Ensamblador, generalmente.
    - Empecemos ya a ver un poco de ensamblador ;)

Análisis estático: Ejemplo 1

- ¿Cómo quedaría en ensamblador x86 el siguiente código C?

```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("Hello world!\n");
}
```

Ejemplo 1

```
.text:08000000                public main
.text:08000000 main           proc near
.text:08000000                push    ebp
.text:08000001                mov     ebp, esp
.text:08000003                and     esp, 0FFFFFFF0h
.text:08000006                sub     esp, 10h          ; Prólogo de la función
.text:08000009                mov     dword ptr [esp], offset s ; Argumento 1 -> "Hello World!"
.text:08000010                call    puts              ; Llamada a función 'puts'
.text:08000015                leave                     ; Epílogo de la función
.text:08000016                retn
.text:08000016 main           endp
.text:08000016
.text:08000016 _text          ends
```

Ejemplo 1: Análisis

- Partes del código
  - Prólogo de función: Indica la convención de llamadas y el espacio reservado para variables locales.
    - NOTA: También alinea a 16 el stack (aunque no es importante ahora).
  - Cuerpo: Establece una constante como argumento 1 ([esp]) y después llama a la función 'puts'.
  - Epílogo: Restaura la pila y retorna a la función llamada.

Ejemplo 1: 64bits

- Ya no es común trabajar, en exclusiva, con procesadores de 32 bits, siendo practicamente todas los procesadores de 64bits.
- ¿Cómo se vería el ejemplo anterior en x86_64?

Ejemplo 1

```
063A main    proc near                          ; DATA XREF: _start+1D↑o
063A
063A var_10  = qword ptr -10h
063A var_4   = dword ptr -4
063A
063A ; __unwind {
063A         push    rbp
063B         mov     rbp, rsp
063E         sub     rsp, 10h
0642         mov     [rbp+var_4], edi
0645         mov     [rbp+var_10], rsi
0649         lea     rdi, s                     ; "Hello world!"
0650         call    _puts
0655         mov     eax, 0
065A         leave
065B         retn
065B ; } // starts at 63A
065B main    endp
```

Ejemplo 1: Análisis 64bits

- Hasta la instrucción ”sub”, esta incluída, tenemos el prólogo de la función.
- Desde la instrucción ”leave”, esta incluída, tenemos el epílogo de la función.
- Las siguientes instrucciones ”mov” las podemos ignorar.
- La instrucción ”lea rdi, s”, carga la dirección de la cadena ”s” en el registro RDI.
- Y, finalmente, llama a la función ”puts”, que tomará su argumento en RDI (segundo argumento).

Análisis estático: Ejemplo 2

- ¿Cómo quedaría en ensamblador x86 el siguiente código C?

```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("Hello %s!\n", argv[1]);
}
```

Ejemplo 2

```
.text:080483C4                 public main
.text:080483C4 main            proc near               ; DATA XREF: _start+17↑o
.text:080483C4
.text:080483C4 arg_4           = dword ptr  0Ch
.text:080483C4
.text:080483C4                 push    ebp
.text:080483C5                 mov     ebp, esp
.text:080483C7                 and     esp, 0FFFFFFF0h
.text:080483CA                 sub     esp, 10h
.text:080483CD                 mov     eax, [ebp+arg_4]
.text:080483D0                 add     eax, 4
.text:080483D3                 mov     edx, [eax]
.text:080483D5                 mov     eax, offset format ; "Hello %s!\n"
.text:080483DA                 mov     [esp+4], edx
.text:080483DE                 mov     [esp], eax      ; format
.text:080483E1                 call    _printf
.text:080483E6                 leave
.text:080483E7                 retn
.text:080483E7 main            endp
```

Ejemplo 2: Análisis

- Prólogo y epílogo: Igual que antes.
- Cuerpo:
  - Mueve la dirección en **ebp+arg4** (argv) al registro **EAX** y le incrementa 4 (argv[1]). El contenido de **EAX** se mueve a **EDX**.
  - Mueve a **EAX** la dirección de una constante ('Hello World!\n').
  - Establece como argumento 2 **EDX** y como argumento 1 **EAX** (es decir, argv[1] y ”Hello World!”).
  - Llama a la función **_printf**.

Ejemplo 2: Notas

- Se utiliza la convención de llamadas ”cdecl”.
- Los argumentos a funciones se pasan en la pila de derecha a izquierda. Se utiliza ESP para ”envíar” argumentos y EBP para ”recoger” argumentos. Ejemplos:
  - Para poner argumentos
    - mov [esp+4], eax → Argumento 2
    - mov [esp], eax → Argumento 1
  - Para leer argumentos
    - mov eax, [ebp+4] → Argumento 2
    - mov eax, [ebp] → Argumento 1

Ejemplo 2: x86_64

```
0064A ; Attributes: bp-based frame
0064A
0064A ; int __cdecl main(int argc, const char **argv, const char **envp)
0064A         public main
0064A main    proc near                       ; DATA XREF: _start+1D↑o
0064A
0064A var_10  = qword ptr -10h
0064A var_4   = dword ptr -4
0064A
0064A ; __unwind {
0064A         push    rbp
0064B         mov     rbp, rsp
0064E         sub     rsp, 10h
00652         mov     [rbp+var_4], edi
00655         mov     [rbp+var_10], rsi
00659         mov     rax, [rbp+var_10]
0065D         add     rax, 8
00661         mov     rax, [rax]
00664         mov     rsi, rax
00667         lea     rdi, format             ; "Hello %s!\n"
0066E         mov     eax, 0
00673         call    _printf
00678         mov     eax, 0
0067D         leave
0067E         retn
0067E ; } // starts at 64A
0067E main    endp
```

Ejemplo 2: Análisis 64bits

- Como podemos ver, realmente, la única diferencia es cómo se pasan los argumentos a las llamadas a función:
  - 32bits: Se pasan usando el stack.
  - 64bits: Se pasan usando RDI, RSI y, luego, el stack.
- Un modo fácil para recordar por qué R**D**I (**D**estination) y R**S**I (Source) se usan para envíar argumentos y en dicho orden, es el siguiente: strcpy(**d**estination, **s**ource).
  - Es decir, en C/C++ si se copia algo a algún sitio, el orden siempre es destino, origen.

Análisis estático: Ejemplo 3

- ¿Cómo quedaría en ensamblador x86 el siguiente código C?

```
#include <stdio.h>
int main(int argc, char **argv)
{
  if (argc > 1)
    printf("Hello %s!\n", argv[1]);
}
```

Ejemplo 3

```
.text:080483C4                 public main
.text:080483C4 main            proc near               ; DATA XREF: _start+17↑o
.text:080483C4
.text:080483C4 arg_0           = dword ptr  8
.text:080483C4 arg_4           = dword ptr  0Ch
.text:080483C4
.text:080483C4                 push    ebp
.text:080483C5                 mov     ebp, esp
.text:080483C7                 and     esp, 0FFFFFFF0h
.text:080483CA                 sub     esp, 10h
.text:080483CD                 cmp     [ebp+arg_0], 1
.text:080483D1                 jle     short locret_80483EC
.text:080483D3                 mov     eax, [ebp+arg_4]
.text:080483D6                 add     eax, 4
.text:080483D9                 mov     edx, [eax]
.text:080483DB                 mov     eax, offset format ; "Hello %s!\n"
.text:080483E0                 mov     [esp+4], edx
.text:080483E4                 mov     [esp], eax      ; format
.text:080483E7                 call    _printf
.text:080483EC
.text:080483EC locret_80483EC:                         ; CODE XREF: main+D↑j
.text:080483EC                 leave
.text:080483ED                 retn
.text:080483ED main            endp
```

Ejemplo 3: Análisis

- Se compara si **[ebp+arg_0]** es **menor o igual** que **1** y de ser así se salta a una etiqueta (**locret_XXX**) donde simplemente se restaura la pila y se retorna.
- De lo contrario, hace lo mismo que en el ejemplo 2.
- Al final, llega a la misma zona de código anterior, la marcada por la etiqueta locret_XXX.

Ejemplo 3: x86_64

- ¿Cómo se vería el mismo ejemplo en ensamblador AMD64 (x86_64)?

Ejemplo 3 x86_64

```
.text:00000000004004F4                  public main
.text:00000000004004F4 main             proc near
.text:00000000004004F4
.text:00000000004004F4 var_10           = qword ptr -10h
.text:00000000004004F4 var_4            = dword ptr -4
.text:00000000004004F4
.text:00000000004004F4                  push    rbp
.text:00000000004004F5                  mov     rbp, rsp
.text:00000000004004F8                  sub     rsp, 10h
.text:00000000004004FC                  mov     [rbp+var_4], edi
.text:00000000004004FF                  mov     [rbp+var_10], rsi
.text:0000000000400503                  cmp     [rbp+var_4], 1
.text:0000000000400507                  jle     short locret_400529
.text:0000000000400509                  mov     rax, [rbp+var_10]
.text:000000000040050D                  add     rax, 8
.text:0000000000400511                  mov     rdx, [rax]
.text:0000000000400514                  mov     eax, offset format ; "Hello %s!\n"
.text:0000000000400519                  mov     rsi, rdx
.text:000000000040051C                  mov     rdi, rax        ; format
.text:000000000040051F                  mov     eax, 0
.text:0000000000400524                  call    _printf
.text:0000000000400529
.text:0000000000400529 locret_400529:                           ; CODE XREF: main+13↑j
.text:0000000000400529                  leave
.text:000000000040052A                  retn
.text:000000000040052A main             endp
```

Análisis estático: Ejemplo 4

- ¿Cómo quedaría en ensamblador x86 el siguiente código C?

```
#include <stdio.h>
void foo(char *arg)
{
char buf[24];
  strcpy(buf, arg);
  printf("Hello %s!\n", buf);
}
```

Ejemplo 4

```
.text:08000000 ; int __cdecl foo(char *src)
.text:08000000                 public foo
.text:08000000 foo             proc near
.text:08000000
.text:08000000 dest            = byte ptr -20h
.text:08000000 src             = dword ptr  8
.text:08000000
.text:08000000                 push    ebp
.text:08000001                 mov     ebp, esp
.text:08000003                 sub     esp, 38h
.text:08000006                 mov     eax, [ebp+src]
.text:08000009                 mov     [esp+4], eax    ; src
.text:0800000D                 lea     eax, [ebp+dest]
.text:08000010                 mov     [esp], eax      ; dest
.text:08000013                 call    strcpy
.text:08000018                 mov     eax, offset format ; "Hello %s!\n"
.text:0800001D                 lea     edx, [ebp+dest]
.text:08000020                 mov     [esp+4], edx
.text:08000024                 mov     [esp], eax      ; format
.text:08000027                 call    printf
.text:0800002C                 leave
.text:0800002D                 retn
.text:0800002D foo             endp
.text:0800002D
.text:0800002D _text           ends
```

Ejemplo 4: Análisis (Parte 1)

- Mueve el argumento **[ebp+src]** a **EAX** y el registro se pone como argumento 2 a función (**[esp+4]**).
- Carga la dirección de la variable local **[ebp+dest]** y la pone como argumento 1 a función (**[esp]**).
- Llama a **strcpy**.
- ...

Ejemplo 4: Análisis (Parte 2)

- Mueve a **EAX** la constante (**”Hello World!\n”**).
- Carga la dirección de la variable local **[ebp+dest]** en **EDX**.
- Establece **EDX** como argumento 2 (**[esp+4]**) y **EAX** como argumento 1 (**[esp]**). Es decir:
  - Argumento 1 → Constante **”Hello World!\n”**
  - Argumento 2 → Variable local **[ebp+dest]**
- Llama a la función **printf**.

Análisis estático: Ejemplo 5

```
#include <stdio.h>
int foo(char *arg)
{
char buf[24];
  if (strlen(arg) > 23)
    return 0;
  strcpy(buf, arg);
  printf("Hello %s!\n", buf);
  return 1;
}
```

```
.text:08000000 ; int __cdecl foo(char *src)
.text:08000000                 public foo
.text:08000000 foo             proc near
.text:08000000
.text:08000000 dest            = byte ptr -20h
.text:08000000 src             = dword ptr  8
.text:08000000
.text:08000000                 push    ebp
.text:08000001                 mov     ebp, esp
.text:08000003                 sub     esp, 38h
.text:08000006                 mov     eax, [ebp+src]
.text:08000009                 mov     [esp], eax      ; s
.text:0800000C                 call    strlen
.text:08000011                 cmp     eax, 17h
.text:08000014                 jbe     short loc_800001D
.text:08000016                 mov     eax, 0
.text:0800001B                 jmp     short locret_8000048
.text:0800001D ; ---------------------------------------------------------------
.text:0800001D
.text:0800001D loc_800001D:                            ; CODE XREF: foo+14↑j
.text:0800001D                 mov     eax, [ebp+src]
.text:08000020                 mov     [esp+4], eax    ; src
.text:08000024                 lea     eax, [ebp+dest]
.text:08000027                 mov     [esp], eax      ; dest
.text:0800002A                 call    strcpy
.text:0800002F                 mov     eax, offset format ; "Hello %s!\n"
.text:08000034                 lea     edx, [ebp+dest]
.text:08000037                 mov     [esp+4], edx
.text:0800003B                 mov     [esp], eax      ; format
.text:0800003E                 call    printf
.text:08000043                 mov     eax, 1
.text:08000048
.text:08000048 locret_8000048:                         ; CODE XREF: foo+1B↑j
.text:08000048                 leave
.text:08000049                 retn
.text:08000049 foo             endp
.text:08000049
.text:08000049 _text           ends
```

Ejemplo 5: Análisis (Parte 1)

- Calcula el tamaño (strlen) del argumento src (ebp+src). El valor de retorno está guardado en EAX.
- Si EAX es **menor o igual** (Bellow or equal) salta a loc_XX1D. Sino, establece EAX a 0 y salta incondicionalmente (JMP) a loc_XX48.
- En loc_XX48 simplemente restaura la pila y retorna.
- ...

Ejemplo 5: Anaĺisis (Parte 2)

- En loc_XX1D copia el argumento src (ebp+src) a la variable local dest (ebp+dest).
- Después, llama a la función printf pasando como argumentos la variable local dest y la cadena de formato ”Hello %s!\n”.
- Tras hacer esto, establece EAX a 1 y continúa en loc_XX84.
- EAX contiene el valor de retorno de la función foo: 0 si el argumento es muy grande o 1 si es correcto.

Ejercicio 1

- Utilizaremos IDA, no Ghidra.
  - Quiero que aprendáis a analizar ensamblador, no leer pseudo-código.

- ¿Qué hace el programa?
- ¿Cuántos argumentos recibe?
- ¿Cuáles son los tipos de datos de los argumentos esperados?

Ejercicio 1: Solución

- ¿Qué hace el programa?
  - **Imprimir el nombre y la edad dados y retornar el número de años.**
- ¿Cuántos argumentos recibe?
  - **2 argumentos: El nombre y la edad.**
- ¿Cuáles son los tipos de datos de los argumentos esperados?
  - **char* para el nombre e int para la edad.**
-

Análisis estático

Preguntas, dudas?

Fundamentos de Ingeniería Inversa

REconstrucción de código

Guión

- Introducción
- Fundamentos
- Construcciones típicas
  - Variables
  - Llamadas a funciones
  - Sentencias condicionales
  - Bucles
- Ejemplos
  - Durante la charla se irá reconstruyendo un sencillo programa

Introducción

- ¿Para qué hace falta?
  - Para recuperar código fuente perdido
  - Para duplicar (y extender) algoritmos
  - Para entender al completo un algoritmo
- Decompilación ¿Hasta qué punto es posible?
  - Pérdida de información
  - Optimizaciones del compilador

Fundamentos

- Compiladores y dialectos de ensamblador
  - Cada compilador genera un código diferente
  - Cada compilador utiliza sus propias convenciones de llamadas
  - Cada compilador se basa en «estándares» diferentes
- Sin embargo, todos llevan al final la misma lógica
  - Una acción se puede hacer de mil modos pero sigue siendo la misma acción

Convenciones de llamadas

- Llamadas a función, las típicas son cdecl, stdcall, fastcall y thiscall.
- Cada compilador para cada CPU utiliza una convención por defecto diferente :
  - Típicamente, en C, se utiliza cdecl (funciones normales) y stdcall (funciones exportadas)

Convenciones de llamadas

- **Todas**: Los argumentos se pasan de derecha a izquierda, siempre.
- Cdecl: Se arregla la pila dentro de la función llamada.
- Stdcall: Exactamente igual pero la pila es arreglada por la función que llama, no por la función invocada.
- Fastcall(32): Los argumentos se pasan en ECX, EDX y la pila.
- Thiscall : Los argumentos se pasan como en cdecl pero el puntero a this se pasa en ECX/RCX.
- X86_64 calling conventions:
  - Microsoft: Argumentos en RCX, RDX, R8, R9 y pila.
  - System V: Argumentos en RDI, RSI, RDX, RCX, R8, R9 y pila.
- ARM32 calling conventions: Argumentos pasados en R0-R3 y pila.
- AArch64 (ARM 64) calling convention: Argumentos pasados en X0-X7 y pila.

Otras convenciones de llamadas

- En esta charla excluiremos ejemplos en C++, GO, Visual Basic, Pascal, etc...
- Solo hablaremos de C.

Construcciones típicas

- Variables
  - Stack, heap y globales
- Llamadas a funciones
  - Con los diferentes tipos de convenciones de llamadas
- Sentencias condicionales
  - If, else, switch, etc...
- Bucles
  - For y while
- Estructuras
  - ¿Cómo es una estructura en ensamblador?

Variables y llamadas a función
Stack y cdecl

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main           proc near               ; CODE XREF: _mainCRTStartup+AF↓p

argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                cmp     [ebp+argc], 1
                jnz     short loc_40106C
                mov     eax, [ebp+argv]
                mov     ecx, [eax]
                push    ecx
                call    _usage
                add     esp, 4
                push    1               ; Code
                call    _exit
; ---------------------------------------------------------------------------

loc_40106C:                             ; CODE XREF: _main+7↑j
                mov     edx, [ebp+argv]
                mov     eax, [edx+4]
                push    eax             ; Source
                call    _foo
                add     esp, 4
                pop     ebp
                retn
_main           endp
```

Variables y llamadas a función
Stack y cdecl

```
; int __cdecl foo(char *Source)
_foo            proc near               ; CODE XREF: _main+25↓p

Text            = byte ptr -100h
Source          = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 100h
                mov     eax, [ebp+Source]
                push    eax             ; Source
                lea     ecx, [ebp+Text]
                push    ecx             ; Dest
                call    _strcpy
                add     esp, 8
                push    0               ; uType
                lea     edx, [ebp+Text]
                push    edx             ; lpCaption
                lea     eax, [ebp+Text]
                push    eax             ; lpText
                push    0               ; hWnd
                call    ds:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
                mov     esp, ebp
                pop     ebp
                retn
_foo            endp
```

Variables y llamadas a función
Stack y cdecl

- En este ejemplo se utilizan:
  - Variables de stack
    - SUB ESP, 0x100 ; 255 bytes de memoria en ESP
    - LEA ECX, [EBP+TEXT] ; Se carga la dirección
  - Llamada a una función cdecl desde ”foo”
    - Call _strcpy ; Se llama a la función
    - Add esp, 8; Y luego se arregla la pila
    - Los argumentos se recogen de la pila

Reconstrucción de código

- ¿Qué hemos aprendido?
  - Sabemos cuanto espacio se ha reservado para variables locales
  - Sabemos el tamaño de las variables estáticas
  - Sabemos la convención de llamadas
    - Por lo cual, sabemos que argumentos se le han pasado a la función
  - Ya sabemos reconstruir simples llamadas a función
    - Veamos el ejemplo

Este código entonces ...

```
; int __cdecl foo(char *Source)
_foo            proc near               ; CODE XREF: _main+25↓p

Text            = byte ptr -100h
Source          = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 100h
                mov     eax, [ebp+Source]
                push    eax             ; Source
                lea     ecx, [ebp+Text]
                push    ecx             ; Dest
                call    _strcpy
                add     esp, 8
                push    0               ; uType
                lea     edx, [ebp+Text]
                push    edx             ; lpCaption
                lea     eax, [ebp+Text]
                push    eax             ; lpText
                push    0               ; hWnd
                call    ds:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
                mov     esp, ebp
                pop     ebp
                retn
_foo            endp
```

...quedaría entonces así

```
void __cdecl foo(int arg)
{
    BYTE buf[0x100];
    strcpy(buf, arg);

    MessageBoxA(0, buf, buf, 0);

}
```

```
void __cdecl foo(char *arg)
{
    char buf[0x100];
    strcpy(buf, arg);

    MessageBoxA(0, buf, buf, 0);

}
```

Más con variables

- ¿Y si la variable fuera local pero *dinámica* y no estática?

```
; int __cdecl foo(char *Source)
_foo proc near

lpText= dword ptr -4
Source= dword ptr  8

push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+Source]
push    eax                     ; Source
mov     ecx, [ebp+lpText]
push    ecx                     ; Dest
call    _strcpy
add     esp, 8
push    0                       ; uType
mov     edx, [ebp+lpText]
push    edx                     ; lpCaption
mov     eax, [ebp+lpText]
push    eax                     ; lpText
push    0                       ; hWnd
call    ds:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
mov     esp, ebp
pop     ebp
retn
```

Código

```
void __cdecl foo(int arg /* ptr */)
{
    int buf; // ptr
    strcpy(buf, arg);

    MessageBoxA(0, buf, buf, 0);

}
```

```
void __cdecl foo(char *arg)
{
    char *buf;
    strcpy(buf, arg);

    MessageBoxA(0, buf, buf, 0);

}
```

Más con variables

- ¿Y si la variable fuera global?
  - Veríamos un «offset d/qword_**address**» en IDA. En Ghidra… un texto con color *rosita*.
  - O con símbolos un offset a la variable:

```
; int __cdecl foo(char *Source)
_foo proc near

Source= dword ptr  8

push    ebp
mov     ebp, esp
mov     eax, [ebp+Source]
push    eax             ; Source
push    offset _buf     ; Dest
call    _strcpy
add     esp, 8
push    0               ; uType
push    offset _buf     ; lpCaption
push    offset _buf     ; lpText
push    0               ; hWnd
call    ds:__imp__MessageBoxA@16 ; MessageBoxA(x,x,x,x)
pop     ebp
retn
```

Código

```
BYTE buf[255];

void __cdecl foo(int arg /* ptr */)
{
    strcpy(buf, arg);
    MessageBoxA(0, buf, buf, 0);
}
```

```
char buf[255];

void __cdecl foo(char *arg)
{
    strcpy(buf, arg);
    MessageBoxA(0, buf, buf, 0);
}
```

Sentencias condicionales

- NOTA : Utilizaremos IDA, de momento.
  - Quiero que aprendáis a hacer esta tarea manualmente, no a leer pseudo-código.
- ¿Cómo se representa una construcción IF/ELSE o SWITCH en ensamblador?
  - Cada compilador hace lo que le da la gana
  - Pero en general, todos siguen la misma forma
  - A la hora de reconstruir el código tomaremos la siguiente norma:
    - Utilizar GOTOs inicialmente
    - Y, lo más importante, invertir la operación
    - Quedará más claro en el siguiente ejemplo….

Sentencias condicionales

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main           proc near               ; CODE XREF: _mainCRTStartup+AF↓p

argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                cmp     [ebp+argc], 1
                jnz     short loc_40106C
                mov     eax, [ebp+argv]
                mov     ecx, [eax]
                push    ecx
                call    _usage
                add     esp, 4
                push    1               ; Code
                call    _exit
; ---------------------------------------------------------------------------

loc_40106C:                             ; CODE XREF: _main+7↑j
                mov     edx, [ebp+argv]
                mov     eax, [edx+4]
                push    eax             ; Source
                call    _foo
                add     esp, 4
                pop     ebp
                retn
_main           endp
```

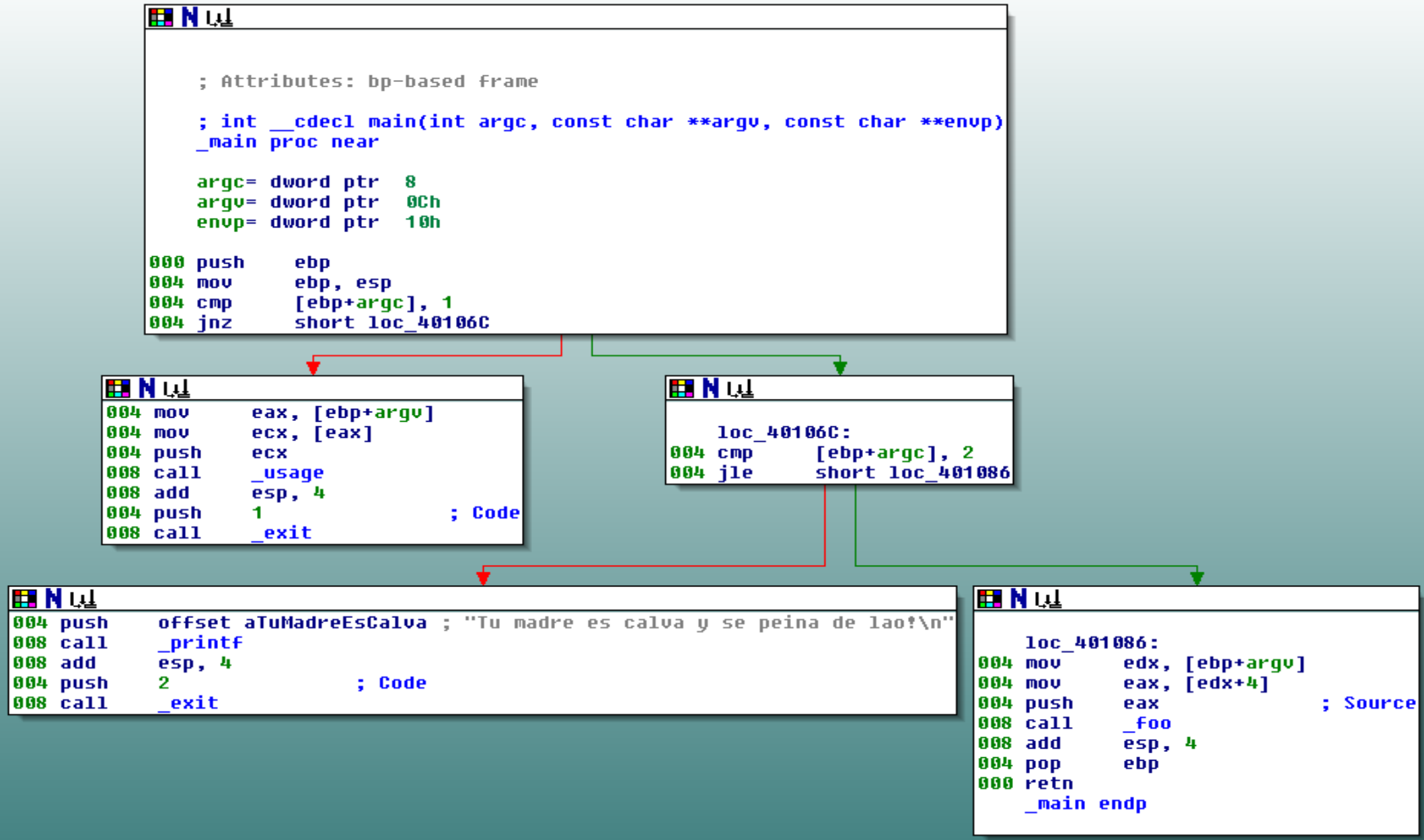
Código

```
if (argc == 1)
{
_usage(argv[0]);
_exit(1);
}
else
{
goto loc_40106C;
}

loc_40106C:
_foo(argv[1]);
```

```
if (argc == 1)
{
  usage(argv[0]);
  exit(1);
}
else
{
  foo(argv[1]);
}
```

Más sentencias condicionales



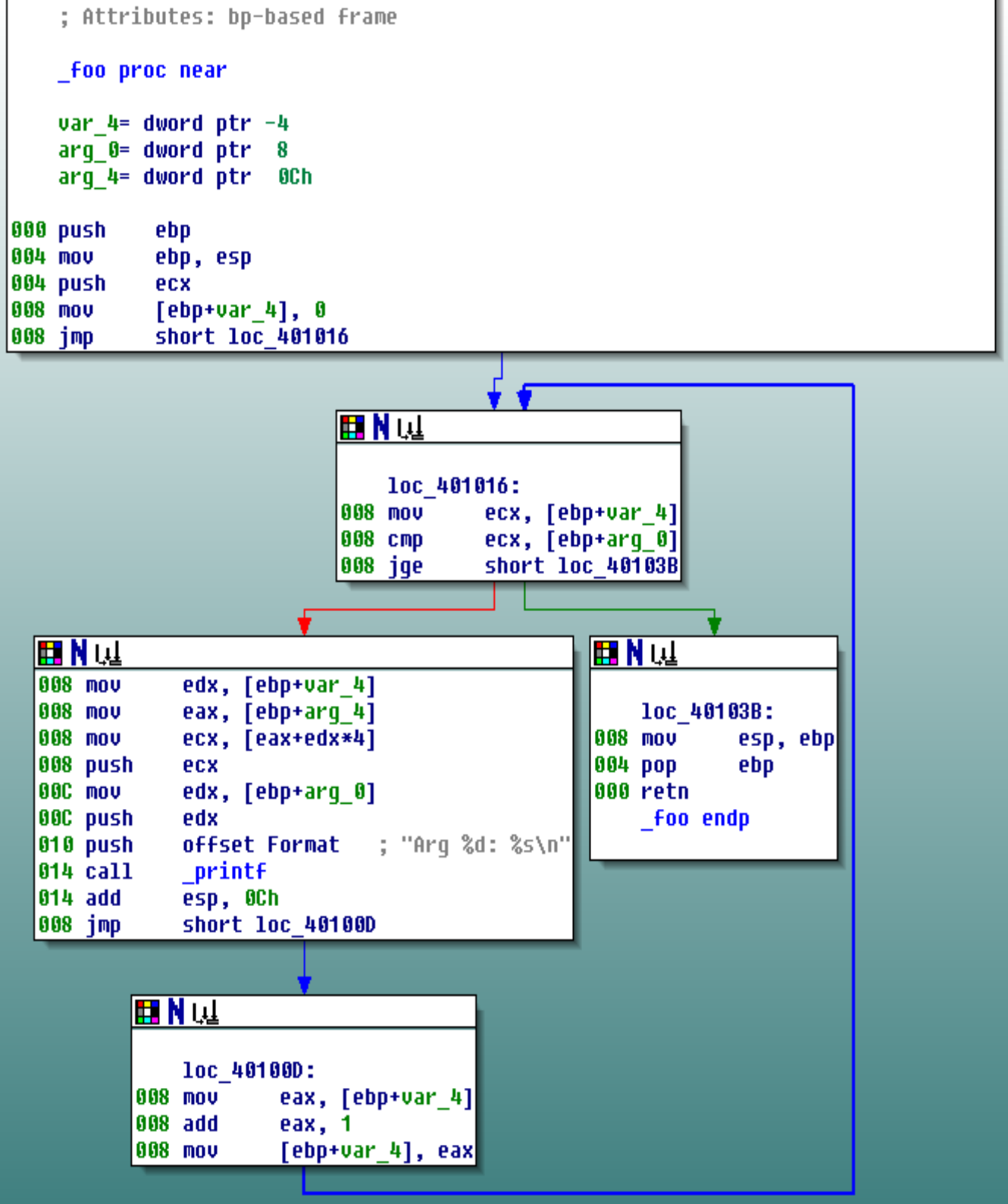
Código

```
if (argc == 1) {
_usage(argv[0]);
_exit(1);
}
If (argc > 2)
  goto loc_40106C;
_printf("Tu madre...");
_exit(2);
loc_40106C:
_foo(argv[1]);
```

```
if (argc == 1)
{
  usage(argv[0]);
  exit(1);
} else if (argc > 2) {
  foo(argv[1]);
} else {
  printf("Tu madre...");
  exit(2);
}
```

Bucles

- ¿Cómo se crean los bucles (for, while) en ensamblador?
  - Pocas veces se utilizan las construcciones que un “humano” usaría:
    - LOOP, LOOPE, etc...
  - Normalmente se utilizan saltos al igual que con las sentencias condicionales
    - LOC_XXX:
    - <<STUFF>>
    - CMP %REG, 1
    - JNZ LOC_XXX
  - Veamos un ejemplo...


; Attributes: bp-based frame
_foo proc near
var_4= dword ptr -4
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
000 push ebp
004 mov ebp, esp
004 push ecx
008 mov [ebp+var_4], 0
008 jmp short loc_401016
loc_401016:
008 mov ecx, [ebp+var_4]
008 cmp ecx, [ebp+arg_0]
008 jge short loc_40103B
008 mov edx, [ebp+var_4]
008 mov eax, [ebp+arg_4]
008 mov ecx, [eax+edx*4]
008 push ecx
00C mov edx, [ebp+arg_0]
00C push edx
010 push offset Format ; "Arg %d: %s\n"
014 call _printf
014 add esp, 0Ch
008 jmp short loc_40100D
loc_40103B:
008 mov esp, ebp
004 pop ebp
000 retn
_foo endp
loc_40100D:
008 mov eax, [ebp+var_4]
008 add eax, 1
008 mov [ebp+var_4], eax

```
_foo            proc near               ; CODE XREF: _main+B↓p

var_4           = dword ptr -4
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                push    ecx
                mov     [ebp+var_4], 0
                jmp     short loc_401016
; ---------------------------------------------------------------

loc_40100D:                             ; CODE XREF: _foo+39↓j
                mov     eax, [ebp+var_4]
                add     eax, 1
                mov     [ebp+var_4], eax

loc_401016:                             ; CODE XREF: _foo+B↑j
                mov     ecx, [ebp+var_4]
                cmp     ecx, [ebp+arg_0]
                jge     short loc_40103B
                mov     edx, [ebp+var_4]
                mov     eax, [ebp+arg_4]
                mov     ecx, [eax+edx*4]
                push    ecx
                mov     edx, [ebp+arg_0]
                push    edx
                push    offset Format   ; "Arg %d: %s\n"
                call    _printf
                add     esp, 0Ch
                jmp     short loc_40100D
; ---------------------------------------------------------------

loc_40103B:                             ; CODE XREF: _foo+1C↑j
                mov     esp, ebp
                pop     ebp
                retn
_foo            endp
```

```
void  _foo(int arg_0, int arg_1)
{
int var_4;

    var_4 = 0;
    goto loc_401016

loc_40100D:
    var_4++;

loc_401016:
    if (arg_0 < var_4)
    {
        _printf("Arg %d: %s\n", arg_0, arg_4);
    }
    goto loc_40100D
}
```

Código: Normalizando

```
void  _foo(int arg_0, int arg_4) {
int var_4;
var_4 = 0;
goto loc_401016
loc_40100D:
var_4++;
loc_401016:
if (arg_0 < var_4) {
_printf("Arg %d: %s\n", arg_0,
          arg_4[var_4]);
}
goto loc_40100D;
}
```

```
void  _foo(int arg_0, char **arg_4)
{
int var_4;

   for (var_4 = 0;
         var_4 < arg_0;
         var_4++)
    {
    _printf("Arg %d: %s\n",
                arg_0, arg_4[var_4]);
    }
}
```

Ejercicio

- Reconstruir el código C del programa 'reco01'
  - Se puede elegir el ELF x86 o el PE x86
- Punto adicional:
  - ¿Qué es lo que hace?

Soluciones

Demo con IDA y luego con Ghidra
(Son iguales)

Reconstrucción de código

Preguntas, dudas?

Fundamentos de Ingeniería Inversa

Búsqueda de vulnerabilidades en binarios

Búsqueda de vulnerabilidades

- ¿Qué es una vulnerabilidad? Wikipedia:
  - Es una debilidad que permite a un atacante reducir la seguridad de la información de un sistema.
- Tipos de vulnerabilidades
  - Memory safety violations
  - Input validation errors
  - Race conditions
  - Privilege escalation
  - ...

Búsqueda de vulnerabilidades

- Vulnerabilidades a identificar:
  - Desbordamientos de buffer (stack y heap)
    - Escritura más allá de los límites de ”variables”.
  - Cadenas de formato
    - Abuso de modificadores de cadenas de formato.
  - Desbordamiento de enteros
    - Cambio de tamaño de ”variables”.
  - Escaladas de privilegios locales en Win32 y Linux
    - Privilegios en subprocesos y librerías.
  - Fallas lógicas
    - ¡Hay tantos tipos!

Buffer overflows (stack)

- El siguiente es un ejemplo de buffer overflow de stack:

```
void foo(char *data)
{
  char buf[8];
  strcpy(buf, data);
}
```

Ejemplo 1

- En este ejemplo, la función ”foo” recibe un argumento ”data” de tipo char *.
- Este argumento se copia en una variable local (guardada en el stack) con tamaño 8.
- Si el tamaño de ”data” es mayor que 8 se sobreescribirán otros datos que estén en la pila
  - Como la dirección de retorno de la función, por ejemplo.

Ejemplo 1

- Ver una vulnerabilidad de este tipo en código fuente suele ser simple
- ¿Cómo se puede identificar una falla de este tipo en ensamblador?
- Veamos el ejemplo con un programa simple en C compilado para x86.

Ejemplo 1: Código C

```
void doprintf(char *data)
{
char buf[255];

  strcpy(buf, data);
  printf(data);
}
```

Ejemplo 1: Código ASM x86

```
; int __cdecl doprintf(char *format)
                public doprintf
doprintf        proc near               ; CODE XREF: main+1A↓p

dest            = byte ptr -107h
format          = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 118h
                mov     eax, [ebp+format]
                mov     [esp+4], eax    ; src
                lea     eax, [ebp+dest]
                mov     [esp], eax      ; dest
                call    _strcpy
                mov     eax, [ebp+format]
                mov     [esp], eax      ; format
                call    _printf
                leave
                retn
doprintf        endp
```

Ejemplo 1: Código ASM x86

```
; int __cdecl doprintf(char *format)
                public doprintf
doprintf        proc near               ; CODE XREF: main+1A↓p

dest            = byte ptr -107h
format          = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 118h
                mov     eax, [ebp+format]
                mov     [esp+4], eax    ; src
                lea     eax, [ebp+dest]
                mov     [esp], eax      ; dest
                call    _strcpy
                mov     eax, [ebp+format]
                mov     [esp], eax      ; format
                call    _printf
                leave
                retn
doprintf        endp
```

Ejemplo 1: Código ASM x86

- La variable local ”dest” tiene tamaño 0x107 según el desensamblado.
- Se substraen 0x118 bytes de la pila, para uso local.
- Se carga la dirección de la variable local ”dest” en el registro de uso general EAX.
- Esta variable se pasa como argumento 1 a la función strcpy.

Ejemplo 1

- ¿Cuántos bytes tenemos que sobreescribir para desbordar la variable ”dest”?
  - Mínimo, 0x107 bytes.
- ¿Cuántos bytes tendríamos que sobreescribir para cambiar todo el espacio reservado por la función?
  - Mínimo, 0x118 bytes.

Ejemplo 1

$ ./test1-x86 `perl -e 'print "a"x **0x106**;'`

aaa...

$ ./test1-x86 `perl -e 'print "a"x **0x107**;'`

Segmentation fault

$ gdb ./test1-x86

(gdb) r `perl -e 'print "a"x 0x107;' `

Program received signal SIGSEGV, Segmentation fault.

0x61616161 in ?? ()

Buffer overflows (heap)

```
void foo(char *arg)
{
char *buf;
  buf = malloc(255);
  strcpy(buf, arg);
  free(buf);
}
```

Ejemplo 2

- La vulnerabilidad está muy clara en el código fuente. ¿Será tan clara en ensamblador x86?

```
; int __cdecl foo(char *src)
                public foo
foo             proc near               ; CODE XREF: main+1A↓p

ptr             = dword ptr -0Ch
src             = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 28h
                mov     dword ptr [esp], 0FFh ; size
                call    _malloc
                mov     [ebp+ptr], eax
                mov     eax, [ebp+src]
                mov     [esp+4], eax    ; src
                mov     eax, [ebp+ptr]
                mov     [esp], eax      ; dest
                call    _strcpy
                mov     eax, [ebp+ptr]
                mov     [esp], eax      ; ptr
                call    _free
                leave
                retn
foo             endp
```

Ejemplo 2

- La vulnerabilidad está muy clara en el código fuente. ¿Será tan clara en ensamblador x86?

```
; int __cdecl foo(char *src)
                public foo
foo             proc near               ; CODE XREF: main+1A↓p

ptr             = dword ptr -0Ch
src             = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 28h
                mov     dword ptr [esp], 0FFh ; size
                call    _malloc
                mov     [ebp+ptr], eax
                mov     eax, [ebp+src]
                mov     [esp+4], eax    ; src
                mov     eax, [ebp+ptr]
                mov     [esp], eax      ; dest
                call    _strcpy
                mov     eax, [ebp+ptr]
                mov     [esp], eax      ; ptr
                call    _free
                leave
                retn
foo             endp
```

Ejemplo 2

- En este caso, el tamaño de la variable queda más claro: El argumento a malloc.
- Vemos que trabaja con punteros, moviendo el resultado de malloc a ”ptr”.
- ¿Cuántos bytes necesitaremos escribir para desbordar la variable ptr?
  - Más de 255 byte(s).

Cadenas de formato

- Ejemplo de función vulnerable a cadenas de formato:

```
void dolog(char *msg)
{
  printf(msg);
}
```

- Bastante obvio, verdad? Y en ensamblador?

Ejemplo 3

```
; int __cdecl dolog(char *format)
                public dolog
dolog           proc near               ; CODE XREF: main+1A↓p

format          = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                mov     eax, [ebp+format]
                mov     [esp], eax      ; format
                call    _printf
                leave
                retn
dolog           endp
```

Ejemplo 3

- ¿Qué necesitamos para identificar una vulnerabilidad de cadena de formato?
  - Identificar llamadas a funciones **printf** (sprintf, vsprintf, …).
  - Identificar el primer argumento: Si la cadena de formato no es una constante, podríamos estar ante una de estas fallas *ancestrales.*

Desbordamiento de enteros

```
void foo(char *argv, int arg_size)
{
char *data;
size_t size;

   size = arg_size;
   if (strlen(argv) > arg_size)
   {
      printf("Too big!\n");
   }
   else
   {
      data = malloc(arg_size);
      strcpy(data, argv);
   }
}

int main(int argc, char **argv)
{
   if (argc == 3)
      foo(argv[1], atoi(argv[2]));
}
```

Desbordamiento de enteros

- ¿Dónde está la vulnerabilidad? ¿Nadie la ha visto? ;)
- Veamos, ¿Qué es lo que hace?
  - Recibe una cadena y un tamaño.
  - Si el tamaño de la cadena es mayor que el tamaño especificado, muestra un error.
    - Aquí está el error.
  - De lo contrario, reserva el tamaño especificado y copia la cadena recibida en una variable local.
    - Sin verificar si ha reservado memoria correctamente...

Desbordamiento de enteros

- Hagamos una prueba:

  $ ./test5 1234 4

  (OK)

  $ ./test5 1234 0

  Too big!

  $ ./test5 1234 -1

  Segmentation fault

- ¿Qué ha pasado? Promoción de enteros.

Promoción de enteros

- Cuando se comparan 2 tipos de diferente tamaño (signed y unsigned) se convierten ambos elementos al de mayor tamaño (unsigned).
- El argumento arg_size tiene signo (int es signed int) mientras que el tamaño (size_t) es unsigned.
- La conversión provoca que el valor de arg_size se cambie de ”-1” a su representación como unsigned int de 32 bits: 0xFFFFFFFF.
- ¿Cuál sería el tamaño del malloc? 4GB.

Desbordamiento de enteros

- Una vez que se conoce este tipo de vulnerabilidad se hace relativamente fácil identificarla teniendo el código.
- ¿Cómo se puede identificar esta vulnerabilidad en código ASM x86?

Ejemplo 4 (Parte I)

- La función ”atoi” devuelve un ”signed integer”

```
main            proc near               ; DATA XREF: _start+17↑o

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                cmp     [ebp+arg_0], 3
                jnz     short locret_8048507
                mov     eax, [ebp+arg_4]
                add     eax, 8
                mov     eax, [eax]
                mov     [esp], eax      ; nptr
                call    atoi
                mov     edx, [ebp+arg_4]
                add     edx, 4
                mov     edx, [edx]
                mov     [esp+4], eax    ; size
                mov     [esp], edx      ; src
                call    foo
```

Ejemplo 4 (Parte II)

- La función **strlen** devuelve un **unsigned**. La instrucción **jbe** hace una comparación a **unsigned** mezclando **signed** e **unsigned**:

```
foo             proc near               ; CODE XREF: main+2E↓p

dest            = dword ptr -10h
var_C           = dword ptr -0Ch
src             = dword ptr  8
size            = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 28h
                mov     eax, [ebp+size]
                mov     [ebp+var_C], eax
                mov     eax, [ebp+src]
                mov     [esp], eax      ; s
                call    _strlen
                mov     edx, eax
                mov     eax, [ebp+size]
                cmp     edx, eax
                jbe     short loc_80484B2
                mov     dword ptr [esp], offset s ; "Too big!"
                call    _puts
```

Ejemplo 4

- Esta vulnerabilidad no es tan sencilla de identificar la primera vez, verdad ;)
- Sin embargo, una vez conocida, se hace más simple.
- Existen otros tipos de promoción de enteros:
  - Signed y unsigned char, por ejemplo.
- La regla principal a recordar: Siempre se convierte al tipo más grande.

Búsqueda de vulnerabilidades

Preguntas, dudas?

Fundamentos de Ingeniería Inversa

FIN
Muchas gracias a tod@s!