# Reverse Engineering



Email: hughpearse@gmail.com
Twitter: https://twitter.com/hughpearse
LinkedIn: http://ie.linkedin.com/in/hughpearse

# Table of Contents

**Our Goal: You should be able to complete the challenges on google docs after this presentation.**

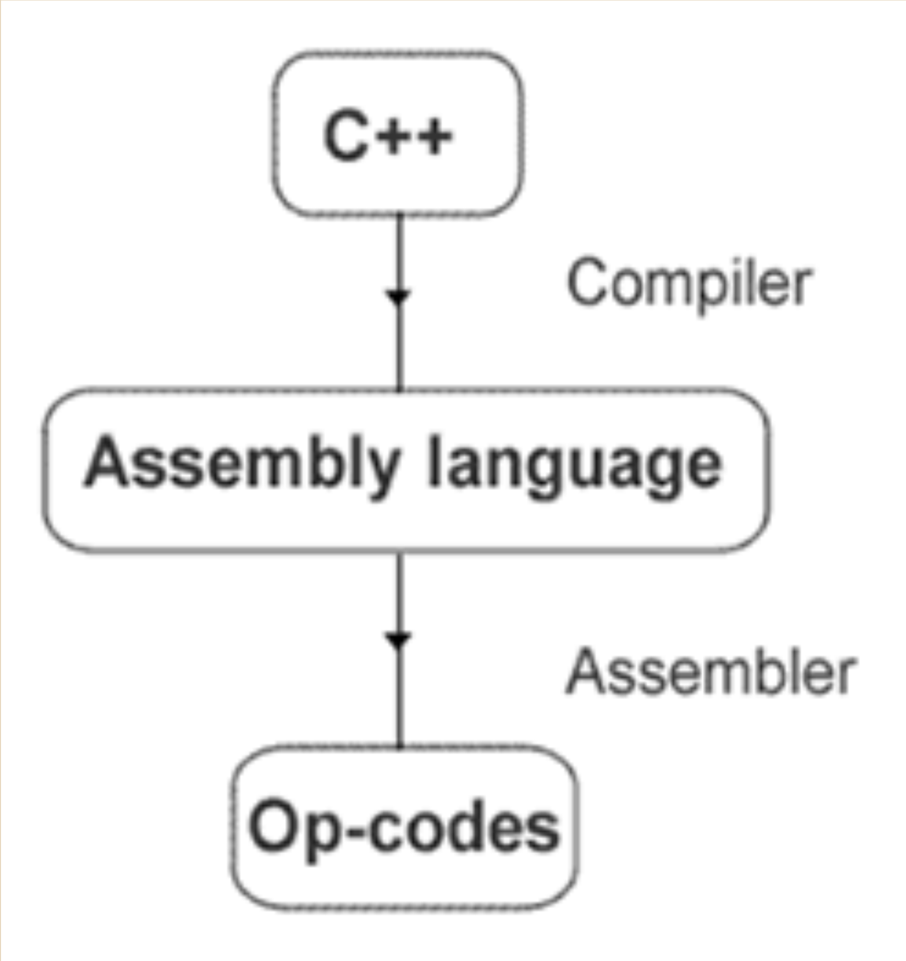
You may want to take some notes for questions at the end.

# Compiling Natively

Natively Compiled Code

```
//hello-world.c
main(){
    printf("hello, world");
}
```

# Compiling Natively



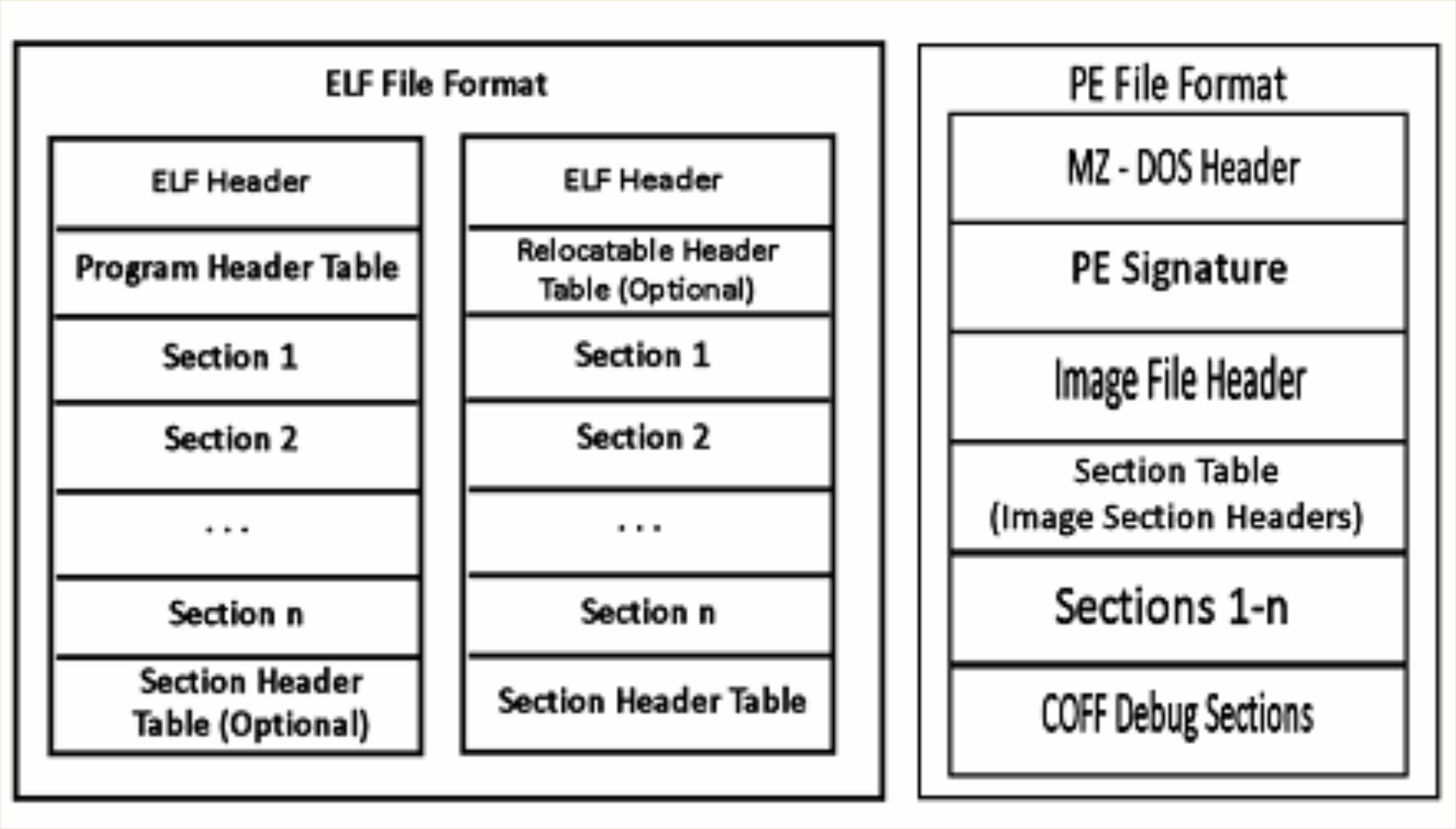
# Compiling Natively

## Applications on Disk - PE and ELF File Formats

There are predominantly two types of native program file formats, Executable and Linking Format (ELF) and Portable Executable (PE) format.

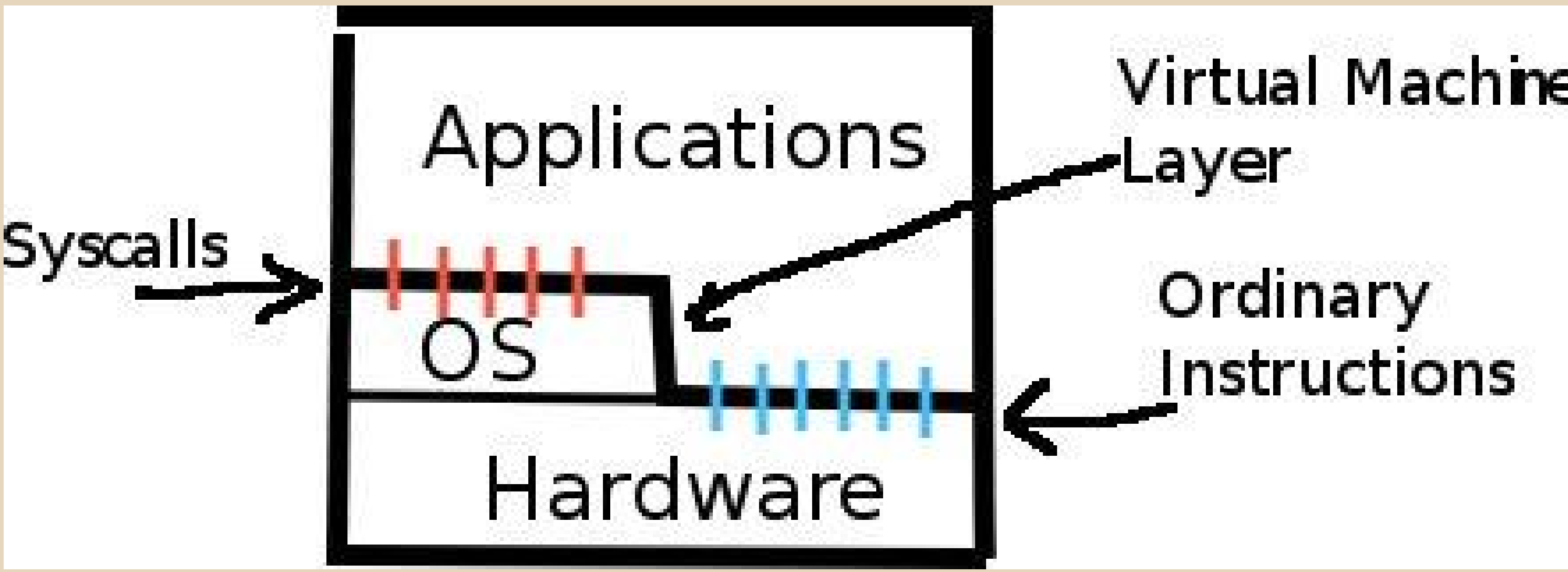
The ELF file typically runs on a Linux/Apple/UNIX while the PE file typically runs on Windows.

# Compiling Natively



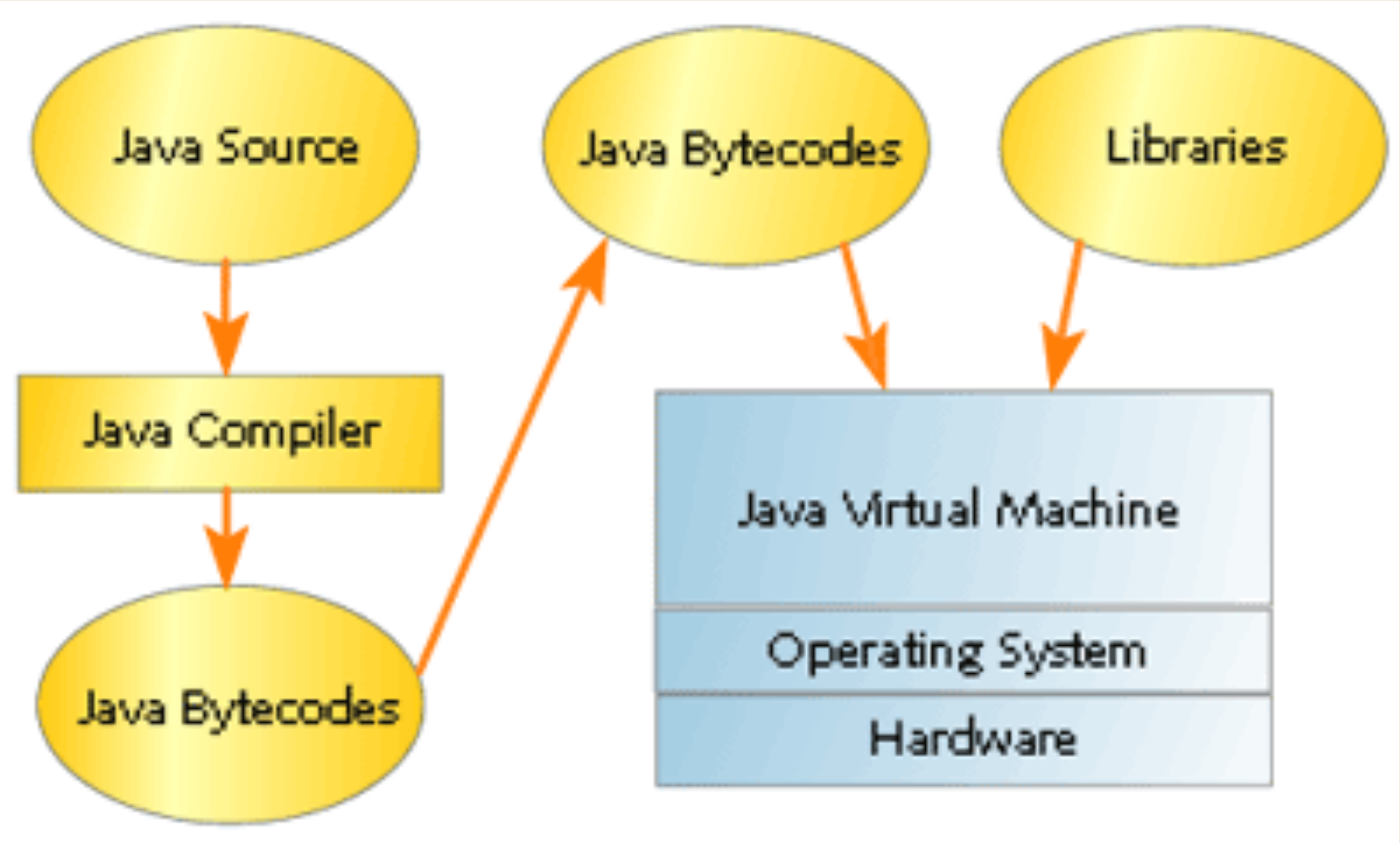
# Compiling Natively

## Native Software Stack



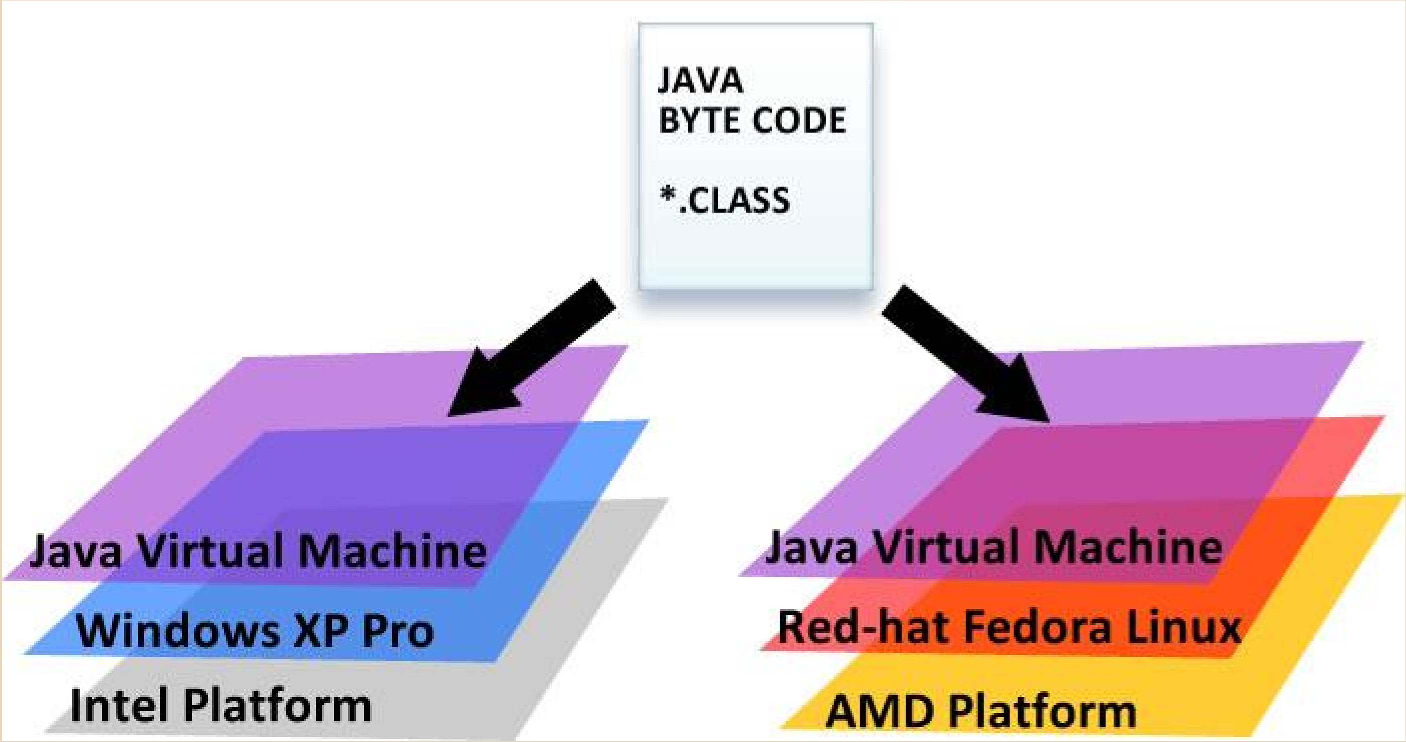
# Compiling Bytecode

Java bytecode - Not native instructions



# Compiling Bytecode

Java is a native application that runs virtualized applications

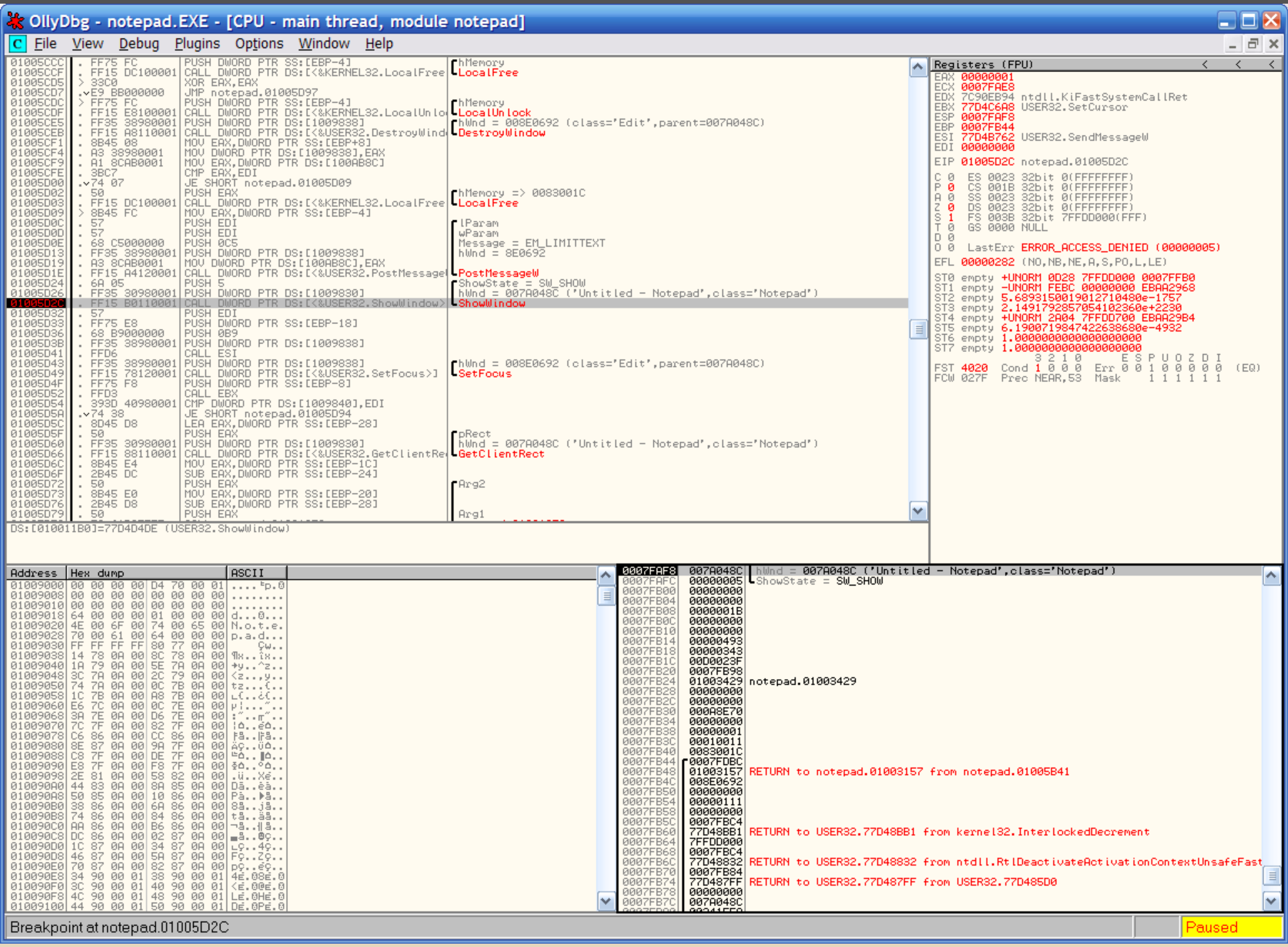


# Disassemblers And Debuggers

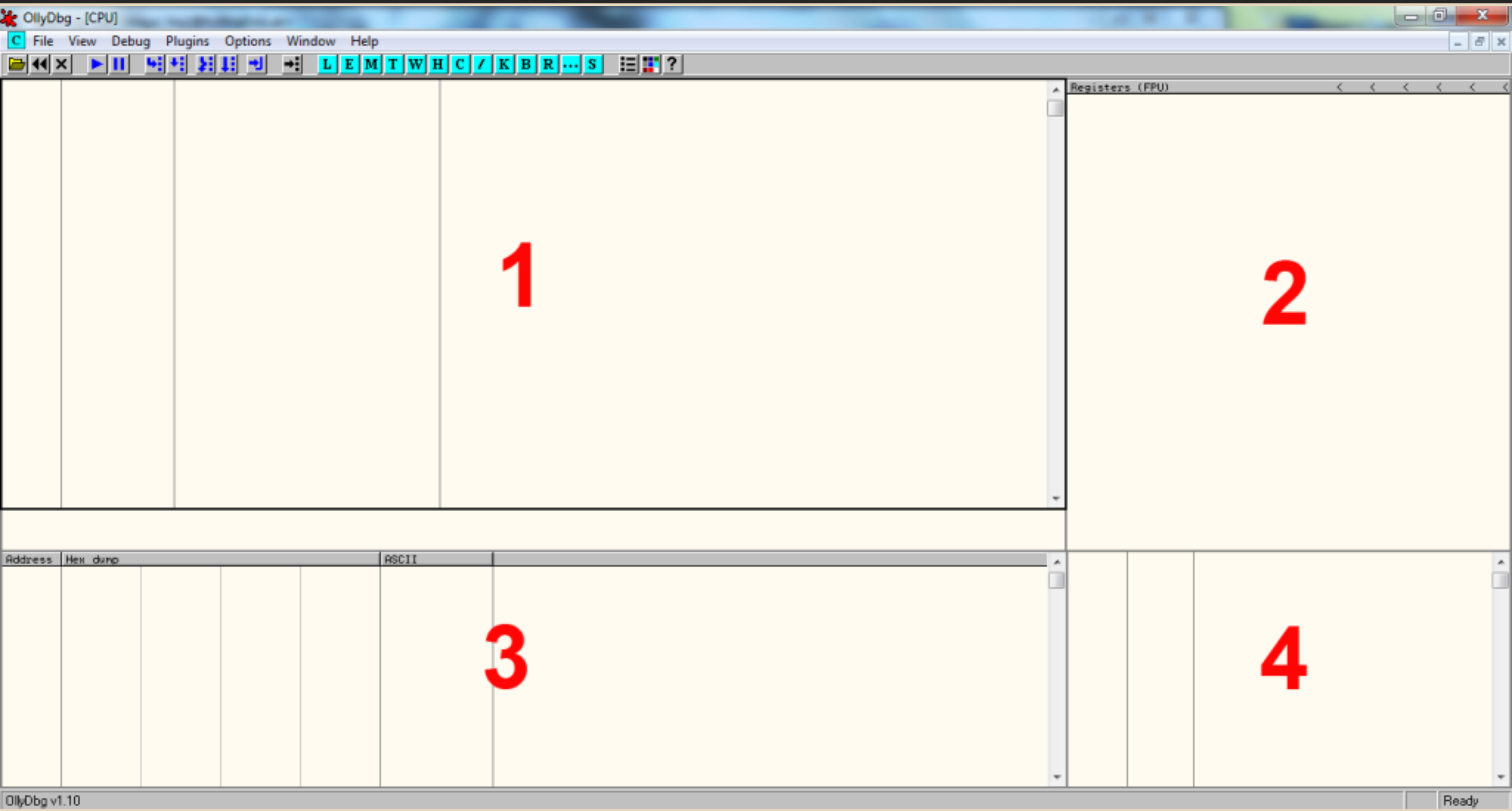
OllyDbg

GDB
The GNU Project Debugger

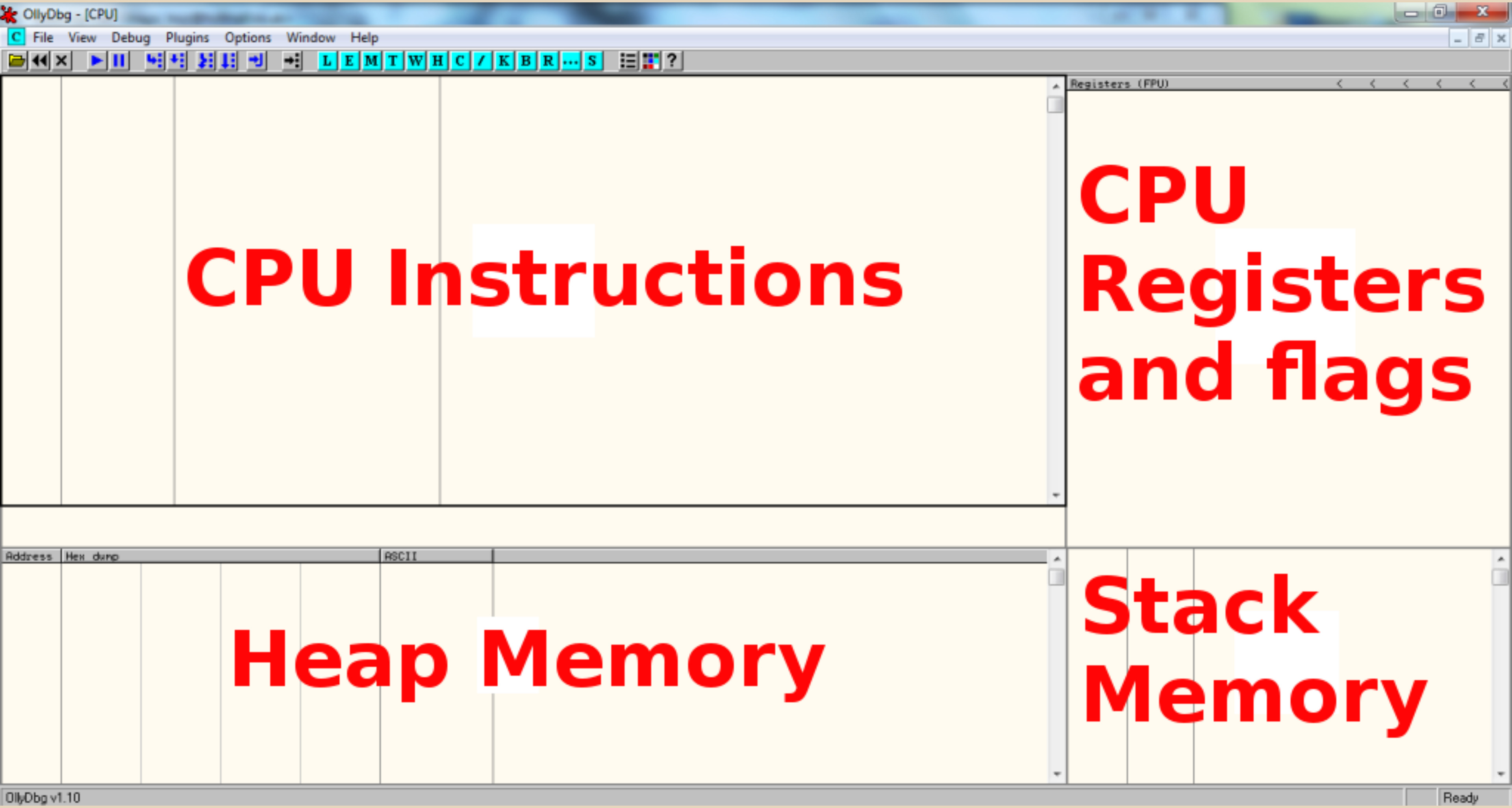
IDA Pro
by Ilfak Guilfanov


OllyDbg - notepad.EXE - [CPU - main thread, module notepad]
C File View Debug Plugins Options Window Help
01005CCC . FF75 FC PUSH DWORD PTR SS:[EBP-4] hMemory
01005CCF . FF15 DC100001 CALL DWORD PTR DS:[<&KERNEL32.LocalFree LocalFree
01005CD5 > 33C0 XOR EAX,EAX
01005CD7 .v E9 BB000000 JMP notepad.01005D97
01005CDC > FF75 FC PUSH DWORD PTR SS:[EBP-4] hMemory
01005CDF . FF15 E8100001 CALL DWORD PTR DS:[<&KERNEL32.LocalUnlo LocalUnlock
01005CE5 . FF35 38980001 PUSH DWORD PTR DS:[1009838] hWnd = 008E0692 (class='Edit',parent=007A048C)
01005CEB . FF15 A8110001 CALL DWORD PTR DS:[<&USER32.DestroyWind DestroyWindow
01005CF1 . 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
01005CF4 . A3 38980001 MOV DWORD PTR DS:[1009838],EAX
01005CF9 . A1 8CAB0001 MOV EAX,DWORD PTR DS:[100AB8C]
01005CFE . 3BC7 CMP EAX,EDI
01005D00 .v 74 07 JE SHORT notepad.01005D09
01005D02 . 50 PUSH EAX hMemory => 0083001C
01005D03 . FF15 DC100001 CALL DWORD PTR DS:[<&KERNEL32.LocalFree LocalFree
01005D09 > 8B45 FC MOV EAX,DWORD PTR SS:[EBP-4]
01005D0C . 57 PUSH EDI lParam
01005D0D . 57 PUSH EDI wParam
01005D0E . 68 C5000000 PUSH 0C5 Message = EM_LIMITTEXT
01005D13 . FF35 38980001 PUSH DWORD PTR DS:[1009838] hWnd = 8E0692
01005D19 . A3 8CAB0001 MOV DWORD PTR DS:[100AB8C],EAX
01005D1E . FF15 A4120001 CALL DWORD PTR DS:[<&USER32.PostMessage PostMessageW
01005D24 . 6A 05 PUSH 5 ShowState = SW_SHOW
01005D26 . FF35 30980001 PUSH DWORD PTR DS:[1009830] hWnd = 007A048C ('Untitled - Notepad',class='Notepad')
01005D2C . FF15 B0110001 CALL DWORD PTR DS:[<&USER32.ShowWindow> ShowWindow
01005D32 . 57 PUSH EDI
01005D33 . FF75 E8 PUSH DWORD PTR SS:[EBP-18]
01005D36 . 68 B9000000 PUSH 0B9
01005D3B . FF35 38980001 PUSH DWORD PTR DS:[1009838]
01005D41 . FFD6 CALL ESI
01005D43 . FF35 38980001 PUSH DWORD PTR DS:[1009838] hWnd = 008E0692 (class='Edit',parent=007A048C)
01005D49 . FF15 78120001 CALL DWORD PTR DS:[<&USER32.SetFocus>] SetFocus
01005D4F . FF75 F8 PUSH DWORD PTR SS:[EBP-8]
01005D52 . FFD3 CALL EBX
01005D54 . 393D 40980001 CMP DWORD PTR DS:[1009840],EDI
01005D5A .v 74 38 JE SHORT notepad.01005D94
01005D5C . 8D45 D8 LEA EAX,DWORD PTR SS:[EBP-28]
01005D5F . 50 PUSH EAX pRect
01005D60 . FF35 30980001 PUSH DWORD PTR DS:[1009830] hWnd = 007A048C ('Untitled - Notepad',class='Notepad')
01005D66 . FF15 88110001 CALL DWORD PTR DS:[<&USER32.GetClientRe GetClientRect
01005D6C . 8B45 E4 MOV EAX,DWORD PTR SS:[EBP-1C]
01005D6F . 2B45 DC SUB EAX,DWORD PTR SS:[EBP-24]
01005D72 . 50 PUSH EAX Arg2
01005D73 . 8B45 E0 MOV EAX,DWORD PTR SS:[EBP-20]
01005D76 . 2B45 D8 SUB EAX,DWORD PTR SS:[EBP-28]
01005D79 . 50 PUSH EAX Arg1
DS:[010011B0]=77D4D4DE (USER32.ShowWindow)
Registers (FPU)
EAX 00000001
ECX 0007FAE8
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 77D4C6A8 USER32.SetCursor
ESP 0007FAF8
EBP 0007FB44
ESI 77D4B762 USER32.SendMessageW
EDI 00000000
EIP 01005D2C notepad.01005D2C
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 1 FS 003B 32bit 7FFDD000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_ACCESS_DENIED (00000005)
EFL 00000282 (NO,NB,NE,A,S,PO,L,LE)
ST0 empty +UNORM 0D28 7FFDD000 0007FFB0
ST1 empty -UNORM FEBC 00000000 EBAA2968
ST2 empty 5.6893150019012710480e-1757
ST3 empty 2.1491792857054102360e+2230
ST4 empty +UNORM 2A04 7FFDD700 EBAA29B4
ST5 empty 6.1900719847422638680e-4932
ST6 empty 1.0000000000000000000
ST7 empty 1.0000000000000000000
3 2 1 0 E S P U O Z D I
FST 4020 Cond 1 0 0 0 Err 0 0 1 0 0 0 0 0 (EQ)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1
Address Hex dump ASCII
01009000 00 00 00 00 D4 70 00 01
01009008 00 00 00 00 00 00 00 00
01009010 00 00 00 00 00 00 00 00
01009018 64 00 00 00 01 00 00 00
01009020 4E 00 6F 00 74 00 65 00 N.o.t.e.
01009028 70 00 61 00 64 00 00 00 p.a.d...
01009030 FF FF FF FF 80 77 0A 00
01009038 14 78 0A 00 8C 78 0A 00
01009040 1A 79 0A 00 5E 7A 0A 00
01009048 3C 7A 0A 00 2C 79 0A 00
01009050 74 7A 0A 00 0C 7B 0A 00
01009058 1C 7B 0A 00 A8 7B 0A 00
01009060 E6 7C 0A 00 0C 7E 0A 00
01009068 3A 7E 0A 00 D6 7E 0A 00
01009070 7C 7F 0A 00 82 7F 0A 00
01009078 C6 86 0A 00 CC 86 0A 00
01009080 8E 87 0A 00 9A 7F 0A 00
01009088 C8 7F 0A 00 DE 7F 0A 00
01009090 E8 7F 0A 00 F8 7F 0A 00
01009098 2E 81 0A 00 58 82 0A 00
010090A0 44 83 0A 00 8A 85 0A 00
010090A8 50 85 0A 00 10 86 0A 00
010090B0 38 86 0A 00 6A 86 0A 00
010090B8 74 86 0A 00 84 86 0A 00
010090C0 AA 86 0A 00 B6 86 0A 00
010090C8 DC 86 0A 00 02 87 0A 00
010090D0 1C 87 0A 00 34 87 0A 00
010090D8 46 87 0A 00 5A 87 0A 00
010090E0 70 87 0A 00 82 87 0A 00
010090E8 34 90 00 01 38 90 00 01
010090F0 3C 90 00 01 40 90 00 01
010090F8 4C 90 00 01 48 90 00 01
01009100 44 90 00 01 50 90 00 01
0007FAF8 007A048C hWnd = 007A048C ('Untitled - Notepad',class='Notepad')
0007FAFC 00000005 ShowState = SW_SHOW
0007FB00 00000000
0007FB04 00000000
0007FB08 0000001B
0007FB0C 00000000
0007FB10 00000000
0007FB14 00000493
0007FB18 00000343
0007FB1C 00D0023F
0007FB20 0007FB98
0007FB24 01003429 notepad.01003429
0007FB28 00000000
0007FB2C 00000000
0007FB30 000A8E70
0007FB34 00000000
0007FB38 00000001
0007FB3C 00010011
0007FB40 0083001C
0007FB44 0007FDBC
0007FB48 01003157 RETURN to notepad.01003157 from notepad.01005B41
0007FB4C 008E0692
0007FB50 00000000
0007FB54 00000111
0007FB58 00000000
0007FB5C 0007FBC4
0007FB60 77D48BB1 RETURN to USER32.77D48BB1 from kernel32.InterlockedDecrement
0007FB64 7FFDD000
0007FB68 0007FBC4
0007FB6C 77D48832 RETURN to USER32.77D48832 from ntdll.RtlDeactivateActivationContextUnsafeFast
0007FB70 0007FB84
0007FB74 77D487FF RETURN to USER32.77D487FF from USER32.77D485D0
0007FB78 00000000
0007FB7C 007A048C
Breakpoint at notepad.01005D2C
Paused

# Disassemblers And Debuggers



# Disassemblers And Debuggers



# Disassemblers And Debuggers

## GDB Tips

| | |
|---|---|
| disas HandleTCPClient | Disassemble the function “HandleTCPClient” |
| disas vulnerable | Disassemble the function “vulnerable” |
| set args 8080 | Set the program arguments to “8080” |
| break *0x1234567 | Set a breakpoint to pause execution at memory address “1234567”. Hint: try setting this to the last instruction in the vulnerable function. |
| break main | Set a breakpoint at the main function |
| run | Execute the program until a breakpoint is reached |
| step | Execute the next instruction in the executable |
| info frame | Display the current stack frame information. Try doing this when you a the breakpoint. |
| x/128xb $rsp | Display 128 bytes of memory in hexadecimal ($rsp is the stack pointer, sometimes $esp). |
| print variable | Display value of variable |
| continue | Continue executing the program until the next breakpoint is reached. |
| kill | Terminate the application without exiting the debugger |
| quit | Exit the GDB application |

# Disassemblers And Debuggers

## GDB has some cool layouts.

(gdb) layout split

(gdb) focus src

(gdb) focus cmd

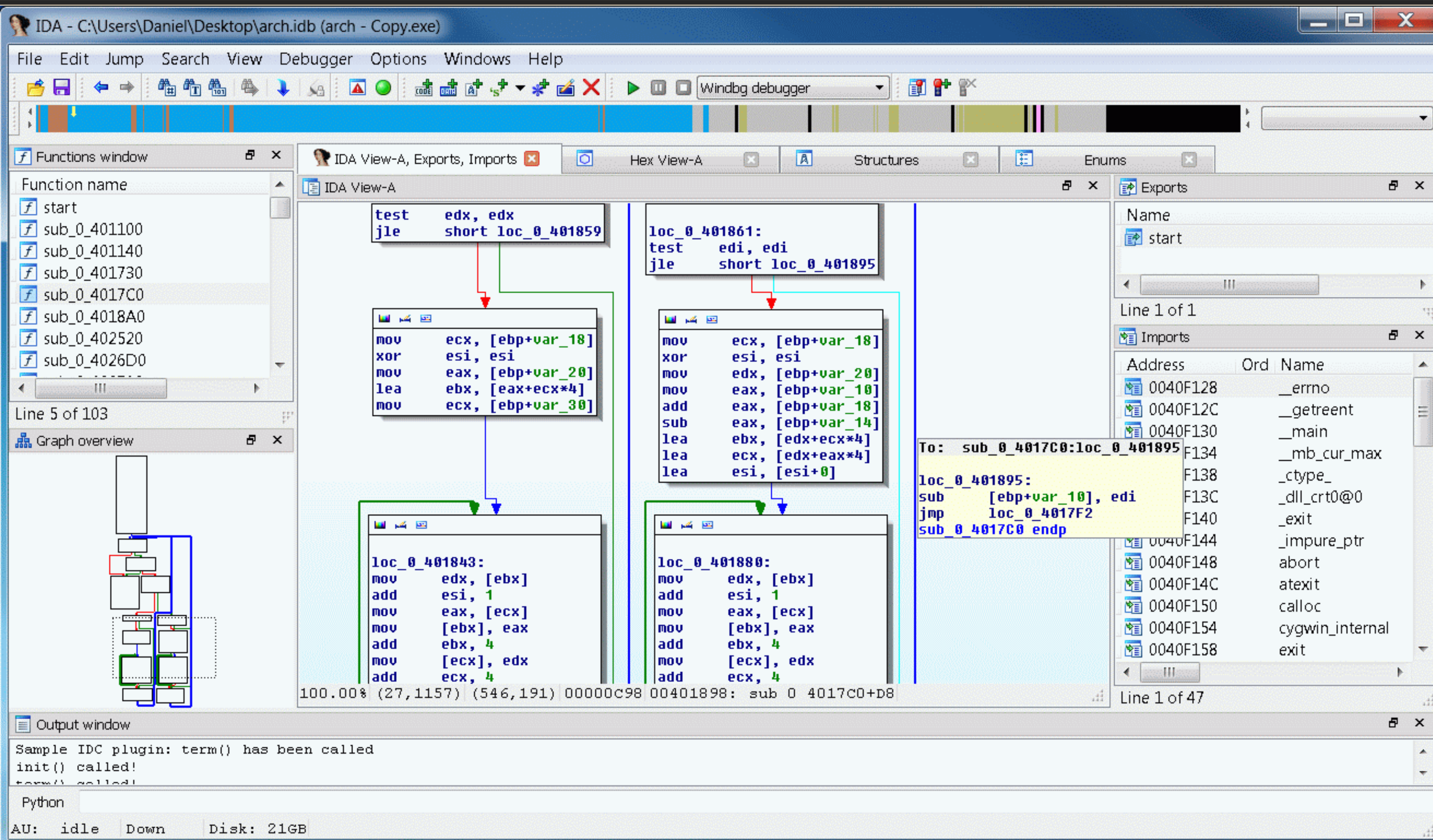
(gdb) help focus

## Compile with debugging symbols

gcc -ggdb

```
 ┌─hello.c──────────────────────────────────────────────────────────────┐
 │1       #include <stdio.h>
 │2       #include <unistd.h>
 │3
 │4       int main(void)
 │5       {
B+>│6               int i = 1;
 │7
 │8               while (i < 60) {
 │9                       i++;
 │10                      sleep(1);
 │11              }
 │12
 │13              return 0;
 │14      }
 │15
 │16

   0x8048384 <main>        lea    0x4(%esp),%ecx
   0x8048388 <main+4>      and    $0xfffffff0,%esp
   0x804838b <main+7>      pushl  -0x4(%ecx)
   0x804838e <main+10>     push   %ebp
   0x804838f <main+11>     mov    %esp,%ebp
   0x8048391 <main+13>     push   %ecx
   0x8048392 <main+14>     sub    $0x14,%esp
B+>0x8048395 <main+17>     movl   $0x1,-0x8(%ebp)
   0x804839c <main+24>     jmp    0x80483ae <main+42>
   0x804839e <main+26>     incl   -0x8(%ebp)
   0x80483a1 <main+29>     sub    $0xc,%esp
   0x80483a4 <main+32>     push   $0x1
   0x80483a6 <main+34>     call   0x80482b8 <sleep@plt>
   0x80483ab <main+39>     add    $0x10,%esp
   0x80483ae <main+42>     cmpl   $0x3b,-0x8(%ebp)
   0x80483b2 <main+46>     jle    0x804839e <main+26>
   0x80483b4 <main+48>     mov    $0x0,%eax

child process 9865 In: main                          Line: 6    PC: 0x8048395

Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...
(gdb) b main
Breakpoint 1 at 0x8048395: file hello.c, line 6.
(gdb) r
Starting program:  /home/beej/hello

Breakpoint 1, main () at hello.c:6
(gdb)
```

# Disassemblers And Debuggers



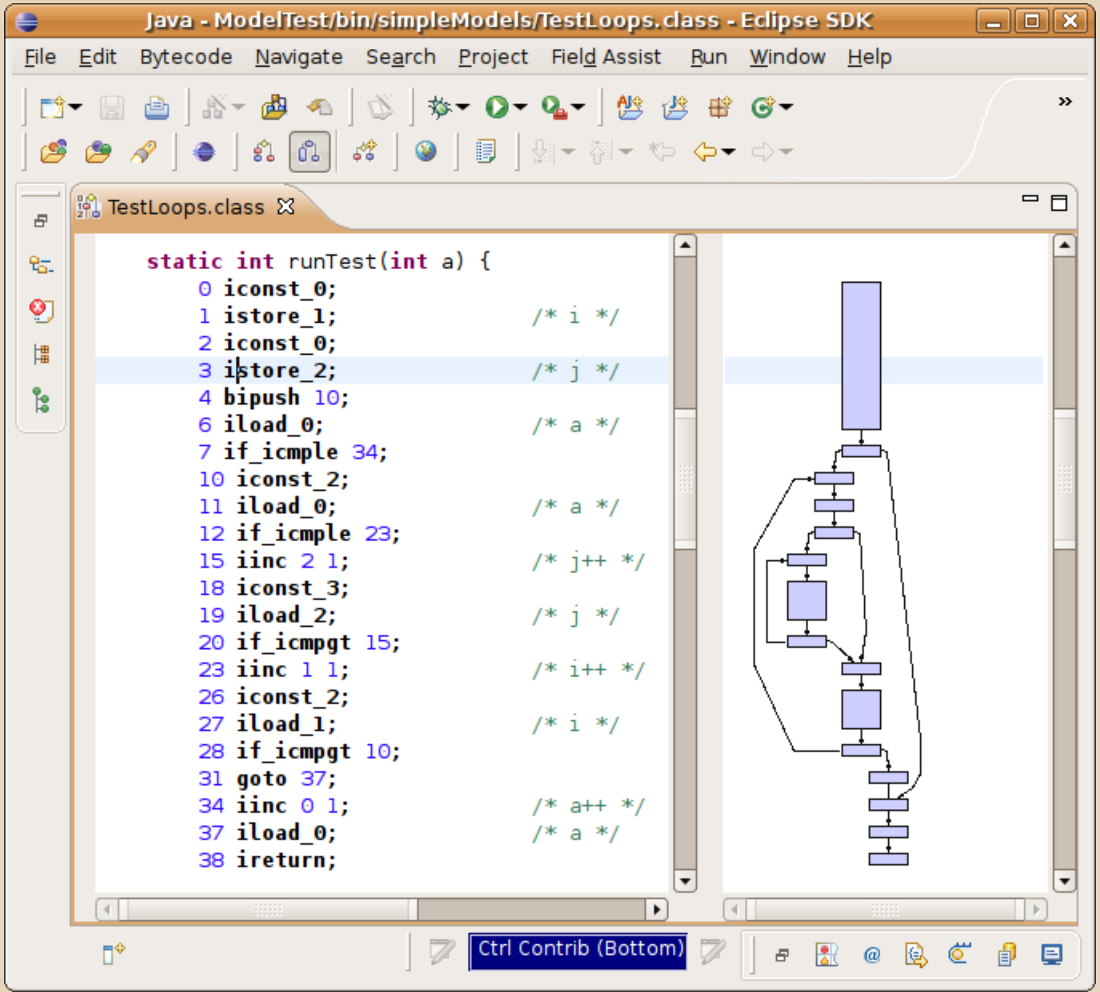
# Disassemblers And Debuggers

## JDB does not support disas

```
javac -g HelloWorld.java
jdb HelloWorld
    1    public class HelloWorld{
    2
    3    public static void main(String[] args){
    4
    5 =>      System.out.println("Hello, world!");
    6
    7    }
    8    }
    9
```

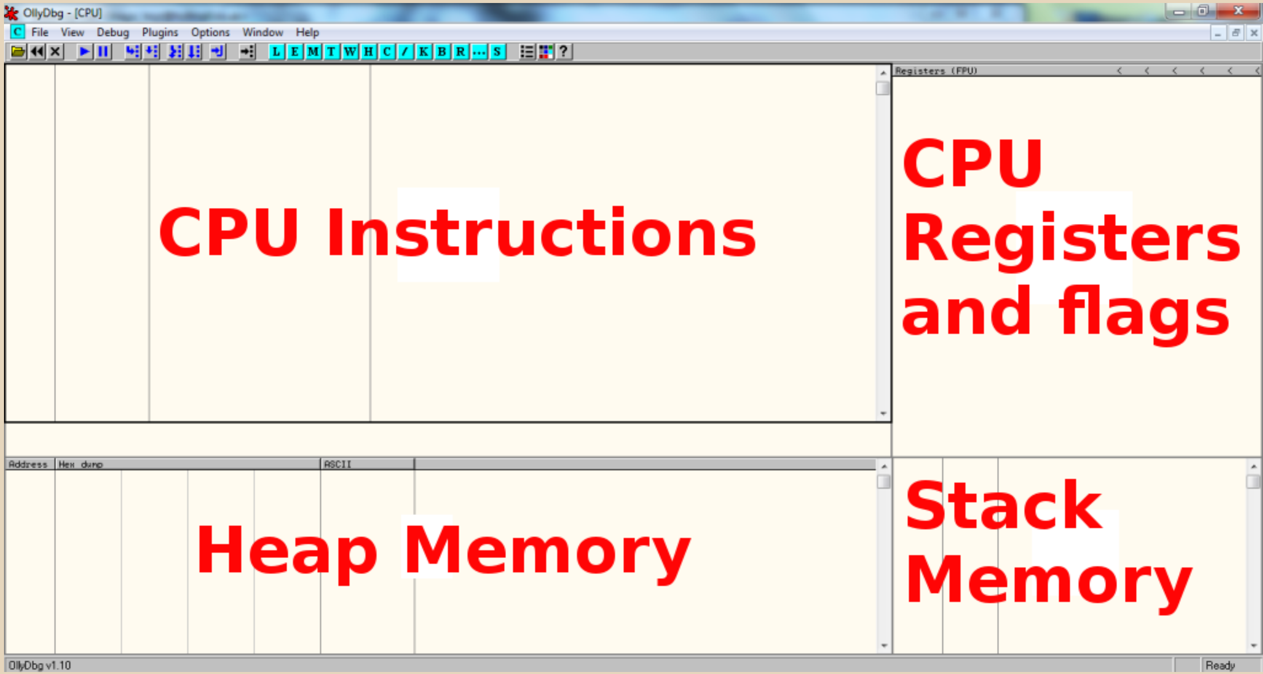
# Disassemblers And Debuggers

## Bytecode Visualizer (Eclipse Market)



# Program Sections

Lets take a look at the 3 types of memory segments in a debugger



# Program Sections

1. Text Segment -> instructions
2. Stack Segment -> variables and return address
3. Heap Segment -> malloc()

# Program Sections

## Text Segment

This is where the compiled code of the program itself resides. It is the machine code, the computer representation of the program instructions. This includes all user defined as well as system functions.

# Program Sections

## Stack Segment

This is a section of memory that is allocated for automatic variables (such as primitives int, float, string) within functions. It is also used to store the return address. Data is stored in stack using the Last In First Out (LIFO) method. This means that storage in the memory is automatically allocated and deallocated at only one end of the memory called the top of the stack.

# Program Sections

## Heap Segment

This is an area of memory used for dynamic memory allocation. Blocks of memory are allocated and freed in this case in an arbitrary order. The pattern of allocation and size of blocks is not known until run time. Heap is usually being used by a program for many different purposes. The malloc() function typically uses heap memory.

# Stack Frames

## Stack Frames part 1

The stack is divided up into contiguous pieces called stack frames, or frames for short; a frame is created each time a function is called. The frame contains the arguments given to the function, the function's local variables, the address at which the function is executing, and the address at which to return to after executing.

# Stack Frames

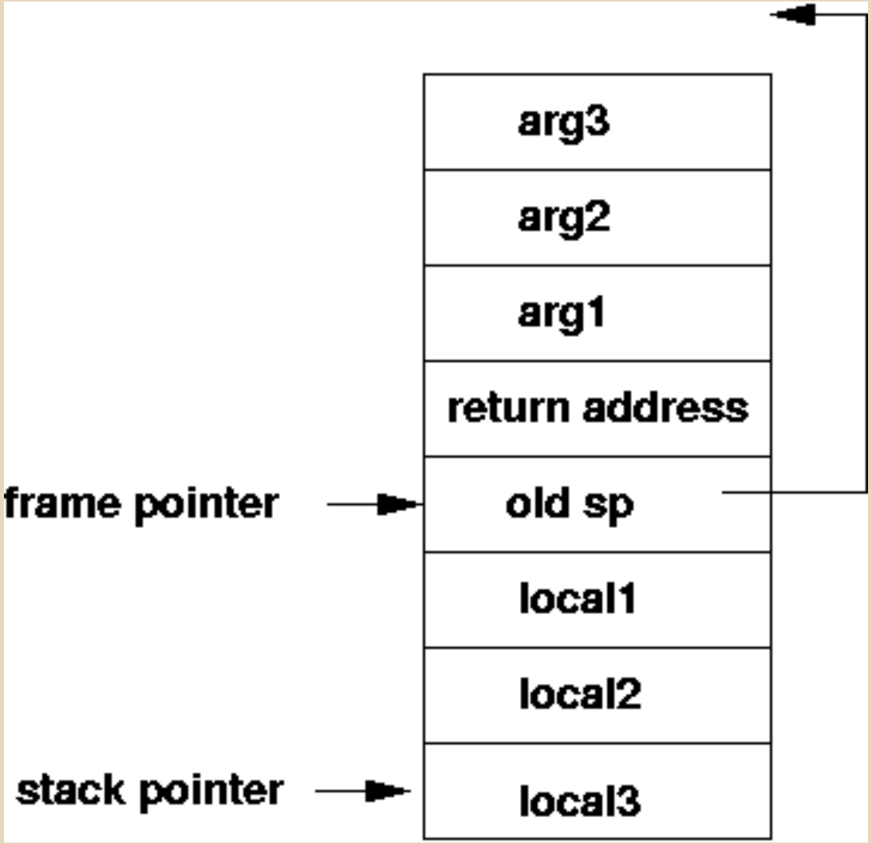
## Stack Frames part 2

When your program is started, the stack has only one frame, that of the function main(). This is called the initial frame or the outermost frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the innermost frame. This is the most recently created of all the stack frames that still exist.

# Stack Frames

## How stack frames are created

```
//example-function.h
int f(int a1, int a2, int a3){
    int local1;
    int local2
    int local3;
}
```



# Stack Frames

## How stack frames are created

The return address is current address + size of the calling instruction

```
push eip + 2
jmp _MyFunction2
```

## These are the same

```
call _MyFunction2
```

# Stack Frames

## How stack frames are destroyed

When a function is finished it uses the "RET" instruction and all the data is removed from the top of the frame and the value of the instruction pointer (EIP) is set as the contents of the last byte in the frame (the return address).

### Unwinding the Stack

When an exception is thrown, the runtime mechanism first searches for an appropriate handler (a catch statement) for it in the current scope. If such a handler does not exist, the current scope is exited and the function that is higher in the calling chain is entered into scope. This process is iterative; it continues until an appropriate handler has been found. An exception is considered to be handled upon its entry to a handler. At this point, all the local objects that were constructed on the path from a try block to a throw-expression have been destroyed. In other words, the stack has been unwound.

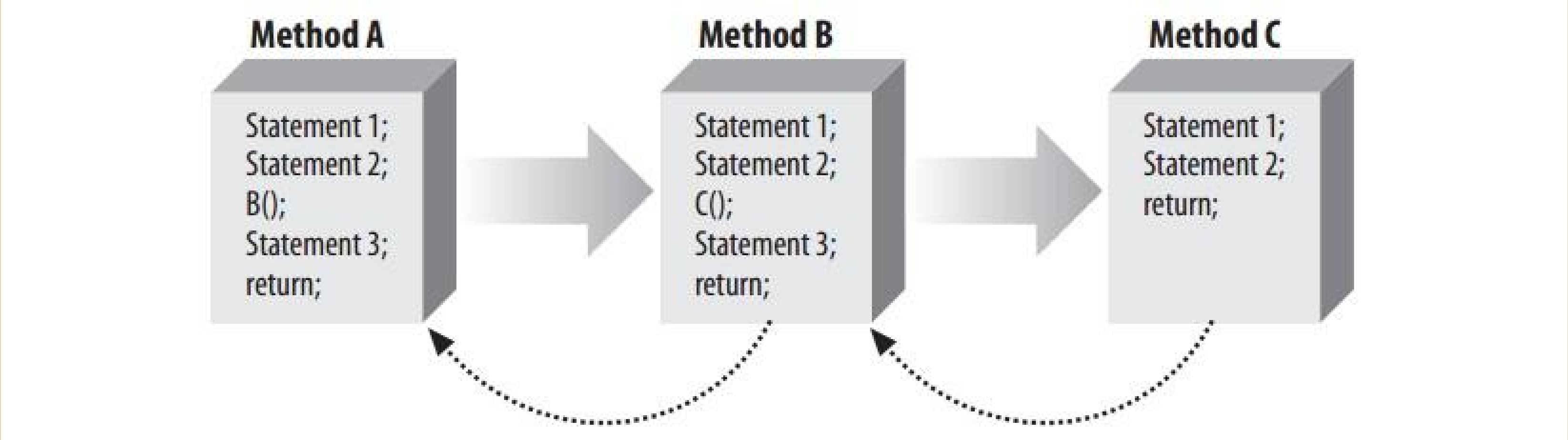
# Payloads

## Normally

-"Function A" executes and calls "function B"

-"Function B" finishes completely and then returns

-"Function A" continues executing from where it stopped



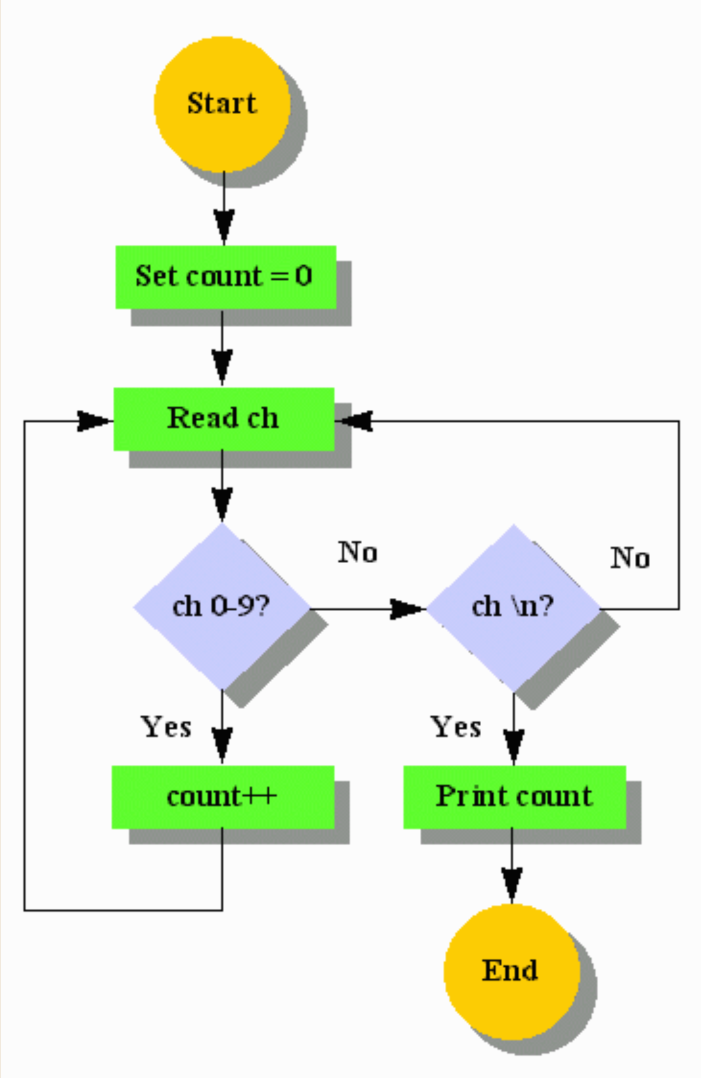
# Payloads

## Buffer Overflow

Find the address of other functions that are included in the program

[AAAA][AAAA][AAAA][AAAA]
[AAAA][function return address]

Function B does not return to A
Function B returns to X



# Payloads

## Smashing The Stack

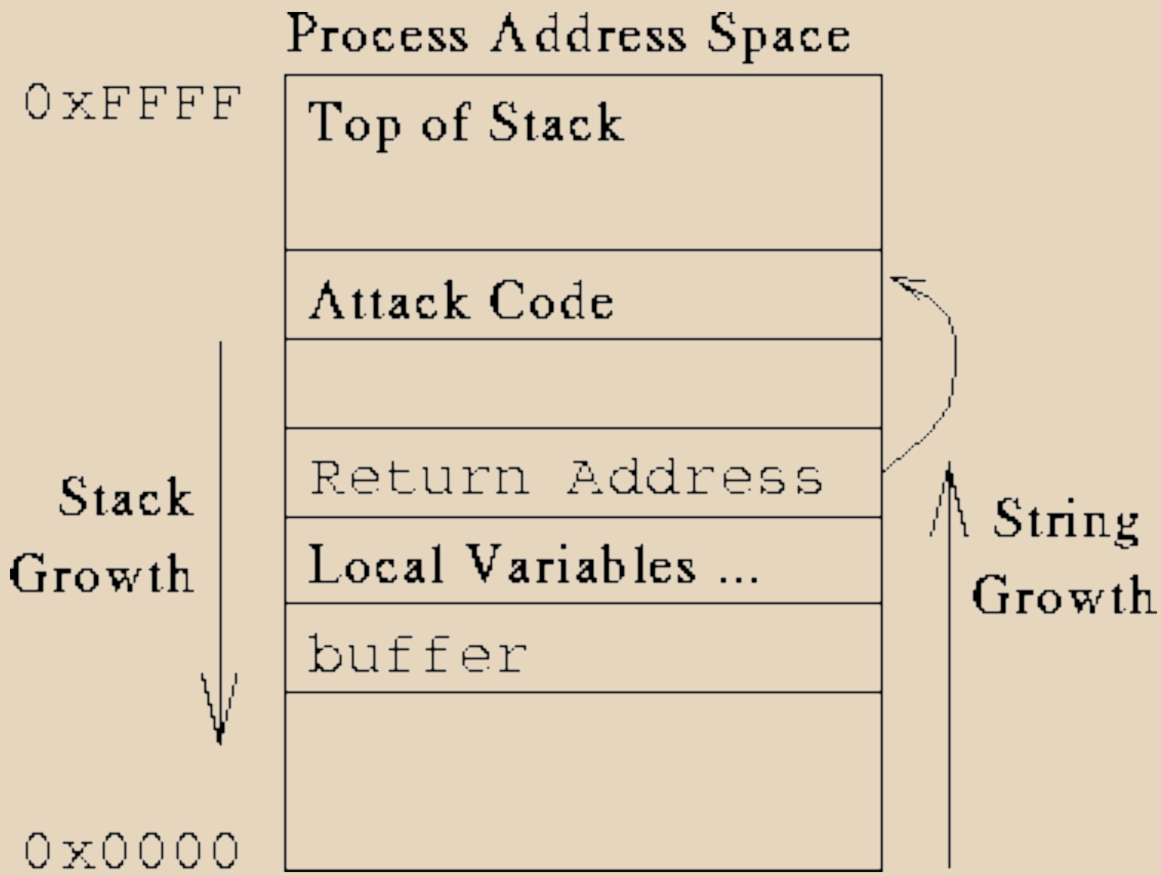
A buffer overflow attack is a general definition for a class of attacks to put more data in a buffer than it can hold thus overwriting the return address.

Smashing the stack is a specific buffer overflow attack where the return address is the same as the address on the stack for storing the variable information. Thus you are executing the stack.

```
[instructions][instructions][return address]
  ^
   |______________________________|
```

# Payloads

## Smashing The Stack



# Payloads

## Using Objdump to craft a payload

Objdump is a program for displaying information about object files. It can be used as a disassembler to view an executable in a human readable form.

Note: Sometimes endian-ness can be a problem

# Payloads

## Lets create a syscall exit() Payload

```
main(){
    exit(0);
}
```

```
gcc -static exit.c -o exit.out
```

# Payloads

## objdump -d ./exit

```
08048f14 <main>:
 8048f14:       55                      push   %ebp
 8048f15:       89 e5                   mov    %esp,%ebp
 8048f17:       83 e4 f0                and    $0xfffffff0,%esp
 8048f1a:       83 ec 10                sub    $0x10,%esp
 8048f1d:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048f24:       e8 77 08 00 00          call   80497a0 <exit>
 8048f29:       66 90                   xchg   %ax,%ax
 8048f2b:       66 90                   xchg   %ax,%ax
 8048f2d:       66 90                   xchg   %ax,%ax
 8048f2f:       90                      nop

08053a0c <_exit>:
 8053a0c:   8b 5c 24 04             mov    0x4(%esp),%ebx
 8053a10:   b8 fc 00 00 00          mov    $0xfc,%eax
 8053a15:   ff 15 a4 f5 0e 08       call   *0x80ef5a4
 8053a1b:   b8 01 00 00 00          mov    $0x1,%eax
 8053a20:   cd 80                   int    $0x80
 8053a22:   f4                      hlt
 8053a23:   90                      nop
 8053a24:   66 90                   xchg   %ax,%ax
 8053a26:   66 90                   xchg   %ax,%ax
 8053a28:   66 90                   xchg   %ax,%ax
 8053a2a:   66 90                   xchg   %ax,%ax
 8053a2c:   66 90                   xchg   %ax,%ax
 8053a2e:   66 90                   xchg   %ax,%ax
```

# Payloads

**Three columns**

**Virtual address; Opcodes; Mnemonic**

We want the opcodes to create our payload

We will be storing our shellcode inside a string data type so we cannot have null values.

To use a system call - execute the hex number 0x80. Depending on the values in the registers, different functions will be called.

# Payloads

## Virtual address; Opcodes; Mnemonic

```
08048f14 <main>:
 8048f14:       55                      push   %ebp
 8048f15:       89 e5                   mov    %esp,%ebp
 8048f17:       83 e4 f0                and    $0xfffffff0,%esp
 8048f1a:       83 ec 10                sub    $0x10,%esp
 8048f1d:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048f24:       e8 77 08 00 00          call   80497a0 <exit>
 8048f29:       66 90                   xchg   %ax,%ax
 8048f2b:       66 90                   xchg   %ax,%ax
 8048f2d:       66 90                   xchg   %ax,%ax
 8048f2f:       90                      nop

08053a0c <_exit>:
 8053a0c:   8b 5c 24 04             mov    0x4(%esp),%ebx
 8053a10:   b8 fc 00 00 00          mov    $0xfc,%eax
 8053a15:   ff 15 a4 f5 0e 08       call   *0x80ef5a4
 8053a1b:   b8 01 00 00 00          mov    $0x1,%eax
 8053a20:   cd 80                   int    $0x80
 8053a22:   f4                      hlt
 8053a23:   90                      nop
 8053a24:   66 90                   xchg   %ax,%ax
 8053a26:   66 90                   xchg   %ax,%ax
 8053a28:   66 90                   xchg   %ax,%ax
 8053a2a:   66 90                   xchg   %ax,%ax
 8053a2c:   66 90                   xchg   %ax,%ax
 8053a2e:   66 90                   xchg   %ax,%ax
```

# Payloads

**Opcodes**

8b 5c 24 04 b8 fc 00 00 00 ff 15 a4 f5 0e 08 b8 01 00 00 00 cd 80

Paste this into a text editor and use the "find and replace" feature to replace space with "\x" to make a hex string out of it in C. Don't forget to surround it with quotes

"\x8b\x5c\x24\x04\xb8\xfc\x00\x00\x00\xff\x15\xa4\xf5\x0e\x08\xb8\x01\x00\x00\x00\xcd\x80"

We now have a string containing machine instructions.

# Payloads

**Simply execute the string to test it**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* shellcode = "\x8b\x5c\x24....."
int main(){
    void (*f)();
    f = (void (*)())shellcode;
    (void)(*f)();
}
```

# Non-Executable Stack

Linux has several inbuilt protection mechanisms to deal with malicious buffer overflow attacks. Some of them are built into kernel while some of them are part of compiler tools such as gcc.

A Non-Executable Stack prevents the Stack Smashing Attack. But it does not prevent all buffer overflow attacks such as heap execution.

# Back to payloads

## Return to LibC

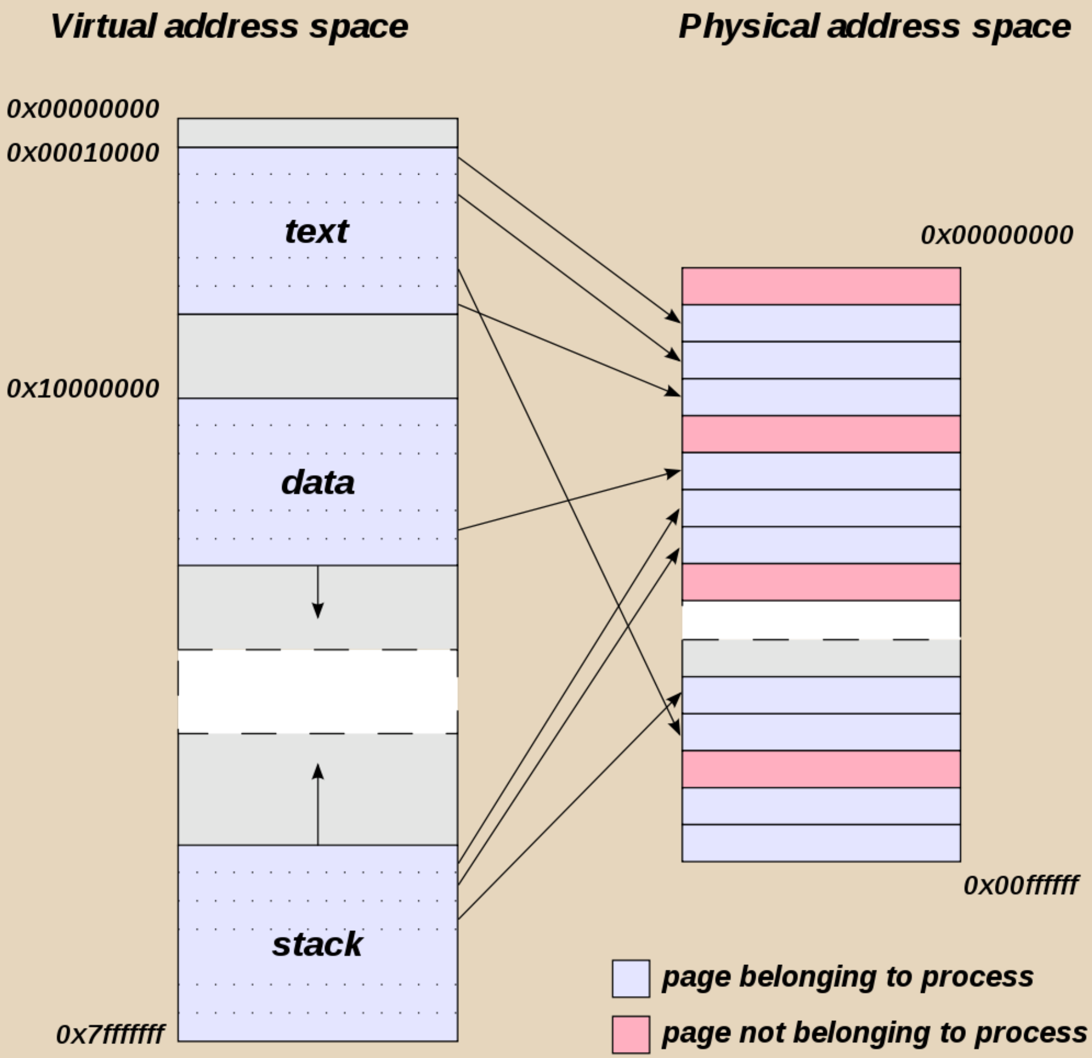
A return-to-libc attack is a computer security attack usually starting with a buffer overflow in which the return address on the call stack is replaced. The shared library called "libc" provides the C runtime on UNIX style systems. Although the attacker could make the code return anywhere, libc is the target as it is always linked to a natively compiled program, and it provides useful calls for an attacker (such as the system() call to execute an arbitrary program.

# Address Space Layout Randomization (ASLR)

Buffer overflows work by modifying the return address of a function on the stack. ASLR complicates that by randomizing the starting point of the virtual addresses. This virtual address starting point is known as the image base.

ASLR makes buffer overflows more difficult since the image base changes each and every time the program runs. The attacker cannot predict that the payload will return to an executable region of memory, or that the payload will even return to valid instructions.

# Address Space



# Address Space

Each process has its own independent virtual address space. Two processes can have a virtual memory page starting at an arbitrary number (for example 1) which maps to different physical pages. Processes can participate in shared memory, in which case they each have a virtual page map to the same physical page.

ASLR does not mean chunks of the application are all over the place, this is already done in physical address space. ASLR means the virtual address does not always start from the same beginning. The starting virtual address of a binary executable is called the image base.

# Stack Canaries

In a typical buffer overflow, the stack is overwritten in an attempt to overwrite the saved return address (EIP). But before the return address is overwritten, the cookie is overwritten as well, rendering the exploit useless.

This technique works by simply setting a secret random number when a function starts and checking if the same number is still there before the function returns. Remember integers are data types that can automatically allocate regions of memory on the stack.

# Disabling Security Features

**For RHEL temporarily disable these security features using the following**

**commands:**

```
sudo sysctl -w kernel.randomize_va_space=0
sudo sysctl -w kernel.exec-shield=0
```

**To compile without security features use the following command:**

```
gcc -ggdb -fno-stack-protector -z execstack -D_FORTIFY_SOURCE=0 -mpreferred-stack-boundary=4 program.c -o program.out
```

# Affected Systems

**Windows ASLR**

ASLR for Windows was first implemented in Windows Vista beta 2. All subsequent workstation and server releases have included the feature including Vista, Server 2008, Server 2008 R2, and Windows 7.

Prior to Windows 8 ASLR is only implemented for files specifically linked to enable ASLR by default.

Windows 8 includes a feature called Force ASLR. With Force ASLR enabled all non-ASLR modules injected into a process are forced to use ASLR as well, thus ensuring the entire process is randomized.

# Affected Systems

**Linux Kernel ASLR**

Linux has enabled a weak form of ASLR by default since kernel version 2.6.12 which was released in 2005.

Over time as more bits have been added to the stack entropy by kernel developers, the period size has increased and as a result giving attackers a larger attack space.

**Android ASLR**

Android has supported ASLR since v4.0 Ice Cream Sandwich.

# Affected Systems

**Windows Data Execution Prevention (DEP)**

DEP has been present on Microsoft Windows since XP SP2 and Windows Server 2003 SP1.

**Linux Kernel DEP**

DEP has been in the stable Linux kernel since release 2.6.8 in August 2004.

**Android DEP**

Android has supported non-executable pages (including non-executable stack and heap) since v2.3 and later.

# Affected Systems

To ensure there is no ASLR, non-executable stack or stack canaries look for systems that fit the following criteria

1. Linux kernel prior to 2.6.8
2. Windows prior to XP SP2
3. Most Routers are shipped with Linux kernel 2.4
4. Custom Firmware with no OS
5. Devices with a power cycle of 12, 24, 36, 48 months as they can not be easily patched or updated in a maintenance window of a few minutes

Read: Protecting Industrial Control Systems from Electronic Threats by Joe Weiss

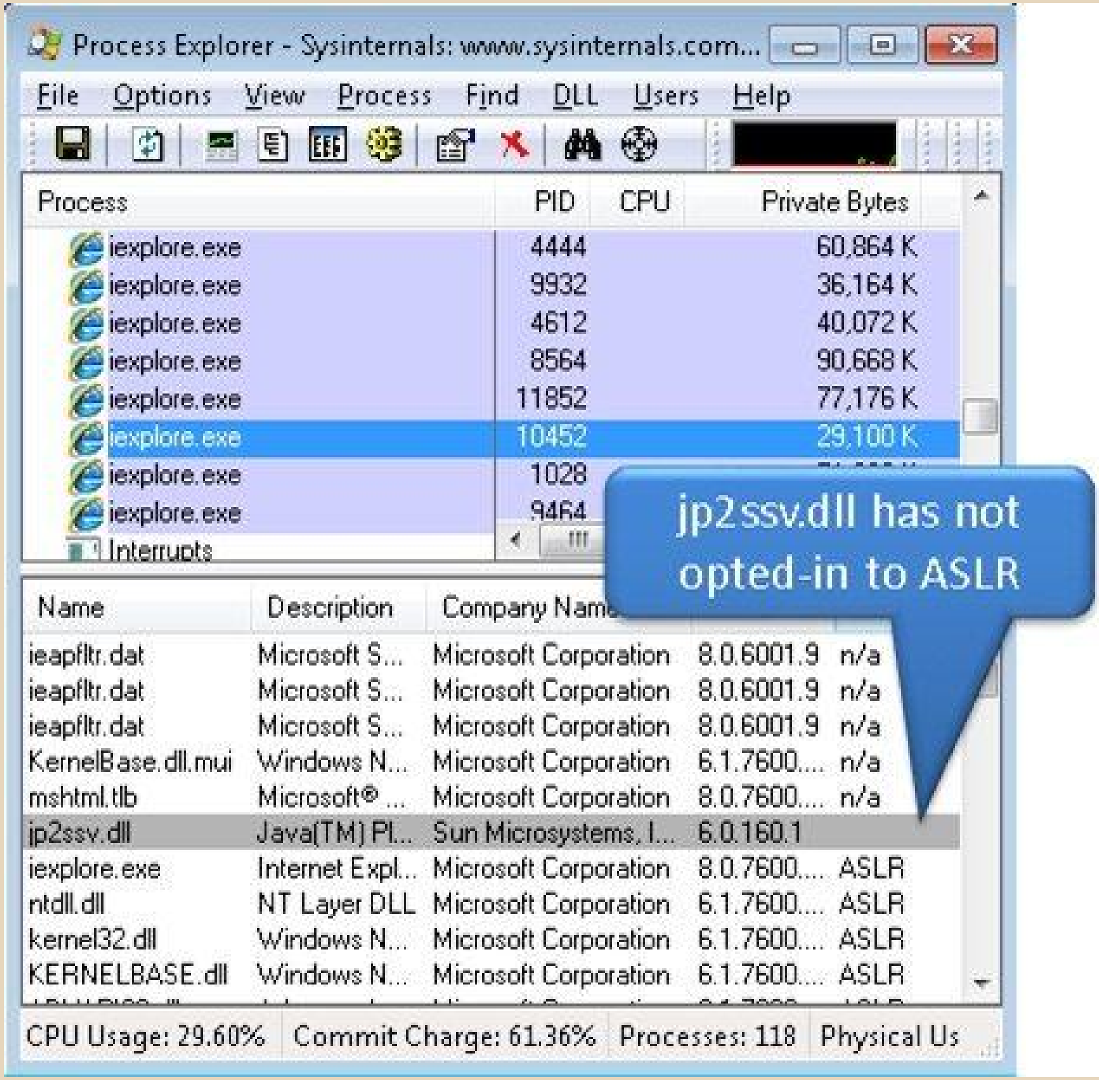
# Affected Systems

## Is ASLR mandatory?

"Prior to Windows 8 ASLR is only implemented for files specifically linked to enable ASLR by default"

Prior to Windows 8, ASLR was opt-in policy

# Affected Systems



# Affected Systems

| Application | DEP (7) | DEP (XP) | Full ASLR |
|---|---|---|---|
| Flash Player | N/A | N/A | YES |
| Sun Java JRE | no | no | no |
| Adobe Reader | YES* | YES* | no |
| Mozilla Firefox | YES | YES | no |
| Apple Quicktime | no | no | no |
| VLC Media Player | no | no | no |
| Apple iTunes | YES | no | no |
| Google Chrome | YES | YES | YES |
| Shockwave Player | N/A | N/A | no |
| OpenOffice.org | no | no | no |
| Google Picasa | no | no | no |
| Foxit Reader | no | no | no |
| Opera | YES | YES | no |
| Winamp | no | no | no |
| RealPlayer | no | no | no |
| Apple Safari | YES | YES | no |

DEP & ASLR (June 2010)

* *Exploitation techniques defeating the feature are publicly known*

# Finding Victims

## Internet Scans

- Shodan
- internetcensus2012.bitbucket.org
- critical.io by HD Moore

## Firmware Analysis

- binwalk
- firmware-mod-kit

# Creating your own exploits

## User Interaction

Find a buffer overflow vulnerability in a piece of software that is commonly used around the world such as an operating system: The Linux Kernel, Windows Kernel, Mac OS Kernel.

Also try finding a vulnerability in applications that are accessible over a network such as Apache web server, SSHD, Portal Servers.

Combining prolificness and network connectivity is the perfect breeding ground for worms.

# Creating your own exploits

## User Interaction

If you target file parsers/readers such as Adobe Acrobat, Flash Player, The Microsoft CHM Reader, Microsoft Word

Then user interaction may be required.

Lets face it, this may or may not happen

# Creating your own exploits

## Regression Tests

Run program on many normal inputs after changes to prevent normal users from encountering errors.

## Code Coverage Tools

Code Coverage Tools are used to determine what lines of your source code are used at runtime. Some of them will even give the line execution frequency.

# Creating your own exploits

## Open Source Regression Tests

Lots of open source packages come with test scripts to ensure the integrity of the build.

## High Value Targets

Combining regression test scripts with code coverage tools will enable vulnerability researchers to find portions of code that are not being regularly tested and portions of code that are frequently executed. Finding bugs should be easy in uncovered regions. Bugs in frequently executed regions should be more valuable.

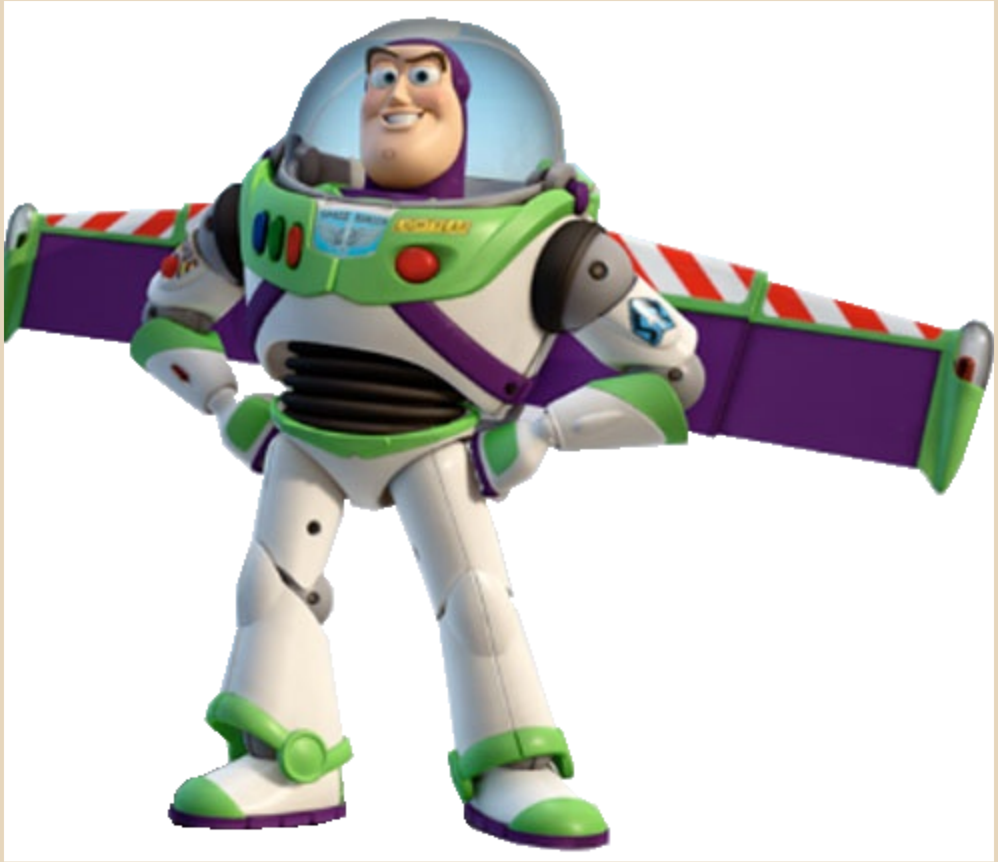
# Creating your own exploits

Some organisations will pay a finders fee

CASH4GOLD.COM

Vupen get paid millions by military organizations for selling their exploits

# Beyond a buffer overflow



# Hooks

## API Hooking

Hooking is a technique for adding extra code to a program/environment for monitoring or changing some program behaviour.

If we have two functions, function A and function B, how do we redirect execution from function A to function B? Well, obviously enough we will be using a jump instruction at some point.

# Hooks

## Environment Variables

Hooking can be performed by simply writing a function with the same name and parameters as the function you are trying to intercept. This code is then compiled to a shared library.

The function in the shared library can override the victims function by simply loading it first. Using this technique there is no patching required.

The LD_PRELOAD environment variable on linux allows libraries to be loaded before the executable.

# Hooks

**GCC Links Dynamically By Default**

gcc prog.c -o dynamically-compiled-elf

**Show Dynamically Linked Libraries**

ldd dynamically-compiled-elf

```
    linux-vdso.so.1 =>  (0x00007fff12bff000)
    libc.so.6 => /usr/lib64/libc.so.6 (0x00000039f3c00000)
    /lib64/ld-linux-x86-64.so.2 (0x00000039f3800000)
```

**LD_PRELOAD Environment Variable**

env LD_PRELOAD=$PWD/libevil.so ./dynamically-compiled-elf

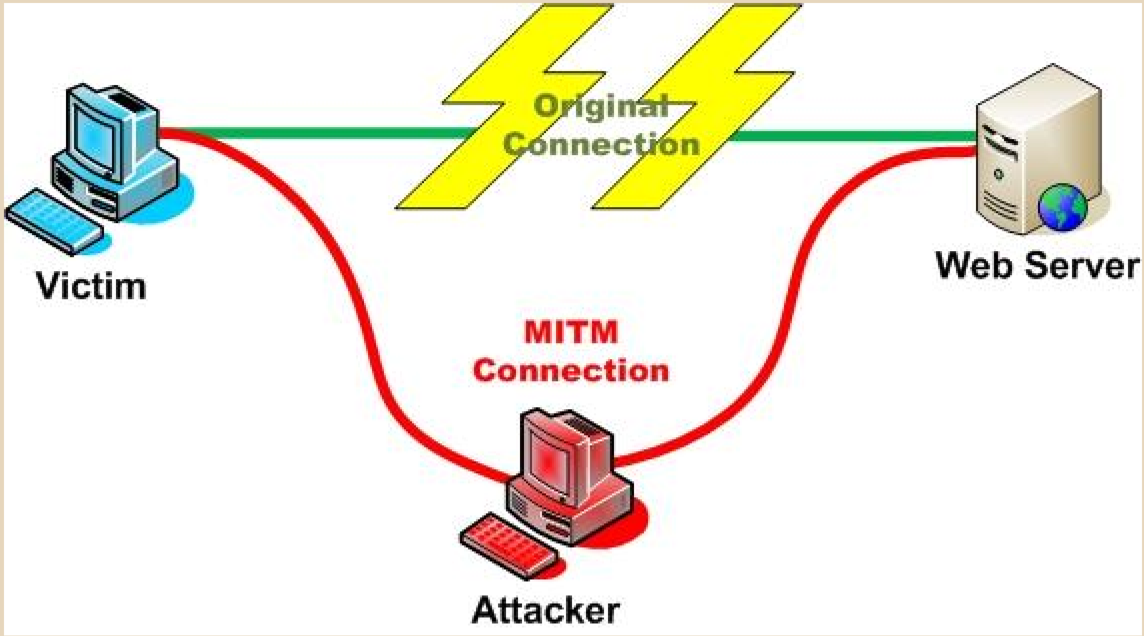
# Hooks

## Patching

By patching an application we can alter its functionality. There exist two families of patching techniques. Hot and cold.

# Hooks

Api hooking sounds very similar to a man in the middle attack against network connections... But for functions!



# Hooks

## Tools

**Pin -** Intel's a multi-platform DLL instrumentation framework for the IA-32 and x86-64 architectures. Pin supports attaching to already running processes.

**DynamoRIO** - a multi-platform runtime code manipulation system that supports code transformations on any part of a program, while it executes.

**DynInst -** a multi-platform runtime code-patching library. DynInst supports static and dynamic instrumentation.

**Valgrind** - a flexible program for debugging and profiling Unix executables.

**bsdiff and bspatch** - tools for building and applying patches to binary files.

**Katana** - a Hot Patching Framework for ELF Executables.

**Detours** - Library for hooking Win32 APIs underneath applications.

# Hooks

## More Tools

**libelf** - lets you read, modify or create ELF files

**pefile** - Python library for reading PE files

**Elfesteem** - Library to read / modify / generate PE/ELF 32/64

**pev** - multiplatform PE analysis toolkit

**LibPE** - emulates the Windows application loader by mapping PE files into memory

**MinGW** - gcc, gdb, objdump for windows

**SSP** - Cygwin Single Step Profiler can generate runtime performance profiles using virtual addresses to control the scope

**Pannus** - kernel patch, library and tools for overwriting runtime code

**Kaho** - provides a binary-patch function upon runtime

# Hooks

## Detection Tools

There are over 18, 000 function pointers (most of them long-lived) existing within the Windows kernel. This is a very large attack surface which means automation is necessary. Api hook detection is generally considered an area of research. There exist only a few research projects.

**Hook Analyser** - Beenu Arora

**RootKit Hook Analyzer** - Resplendence Software

**HookScout** - Berkeley University

# Bytecode Hooking

## JVM Data Types

Byte - b

Short - s

Integer - i   (booleans are mapped to ints)

Long - l

Character - c

Single float - f

double float - d

References - a   (to Classes, Interfaces, Arrays)

Note: These are use as prefixes in opcodes (iadd, astore...)

# Bytecode Hooking

## JVM Mnemonics

Shuffling (pop, swap, dup ...)

Calculating (iadd, isub, imul, idiv, ineg...)

Conversion (d2i, i2b, d2f, i2z...)

Local storage operation (iload, istore...)

Array Operation (arraylength, newarray...)

Object management (get/putfield, invokevirtual, new)

Push operation (aconst_null, iconst_m1....)

Control flow (nop, goto, jsr, ret, tableswitch...)

Threading (monitorenter, monitorexit...)

# Bytecode Hooking

## javap -verbose -c -private HelloWorld

```
public class HelloWorld
{
    public static void main(String[] args){
        System.out.println("Hello, world!");
    }
}
```

# Bytecode Hooking

```
{
public HelloWorld();
 Code:
  Stack=1, Locals=1, Args_size=1
  0: aload_0
  1: invokespecial #1; //Method java/lang/Object."<init>":()V
  4: return

 LineNumberTable:
  line 1: 0

public static void main(java.lang.String[]);
 Code:
  Stack=2, Locals=1, Args_size=1
  0: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc #3; //String Hello, world!
  5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: return
 LineNumberTable:
  line 5: 0
  line 6: 8
}
```

# Bytecode Hooking

## Recognising Constructs

Try doing this for a few basic lines of code and you will begin to start recognising the layout of constructs such as memory allocation and flow control.

# Bytecode Hooking

## Java Attach API

Java 6.0 contains the Attach API feature that allows seamless, inter-process modification of a running JVM. The Attach API is a Sun extension that provides a way for a Java process to “attach” to another JVM at runtime. This bridge can be used to load Java agents onto remote virtual machines. Those agents can then redefine classes or retrieve information about the JVM to which it’s attached

# Bytecode Instrumentation

## Java Instrumentation Interface

This class provides services needed to instrument Java programming language code. Instrumentation is the addition of byte-codes to methods for the purpose of gathering data to be utilized by tools. Since the changes are purely additive, these tools do not modify application state or behavior. This can be useful for logging events.

# Bytecode Complexities

## The Java JIT Compiler

Java programs are stored in memory as bytecode, but the code segment currently running is preparatively re-compiled by the Java JIT compiler from bytecode to physical machine code in order to run faster. JIT includes on-the-fly hardware specific optimizations and optimizations specific to the class files.

JIT is incompatible with the mprotect feature of PaX included with the Linux Grsecurity kernel patch. Mprotect adds rules to memory segments such as read, write, execute.

# Bytecode Hacking Tools

## Java Tools

**javap** - The Java Class File Disassembler

**JavaSnoop** - Hack Java Applications using Attach

**Belch (Burpsuite plugin)** - Intercept java serialization

**Javassist** - Lib for inserting bytecode into class files

**ObjectWeb ASM** - API for decomposing and modifying bytecode

**Apache BCEL** - analyze, create, and manipulate class files

**JOIE** - framework for Java bytecode transformation

**reJ** - visualize, search, compare, modify, refactor Java Class files

**Serp** - framework for manipulating Java bytecode.

**JMangler** - Framework for Load-Time Transformation of Class Files

**ReFrameworker** - produce modified binaries to perform tasks not indented originally by the software developer.

# Bytecode Hacking Tools

## More Java Tools

**Omniscient Debugger (ODB)** - Java Debugger by Bil Lambda

**jClassLib bytecode viewer** - read, modify and write Java bytecode

**JProfiler** - find bottlenecks, memory leaks and understand threading

**JDB** - Command line debugger for Java

**javah** - generates C header needed to implement JNI for native code

**P6Spy** - Framework for intercepting JDBC statements

# Bytecode Hacking Tools

## Tools for people who use .NET

**MBEL -** parse, create, edit, and rewrite .NET files

**Runtime Assembly Instrumentation Library** - Instrumentation

# Bytecode Hacking Tools

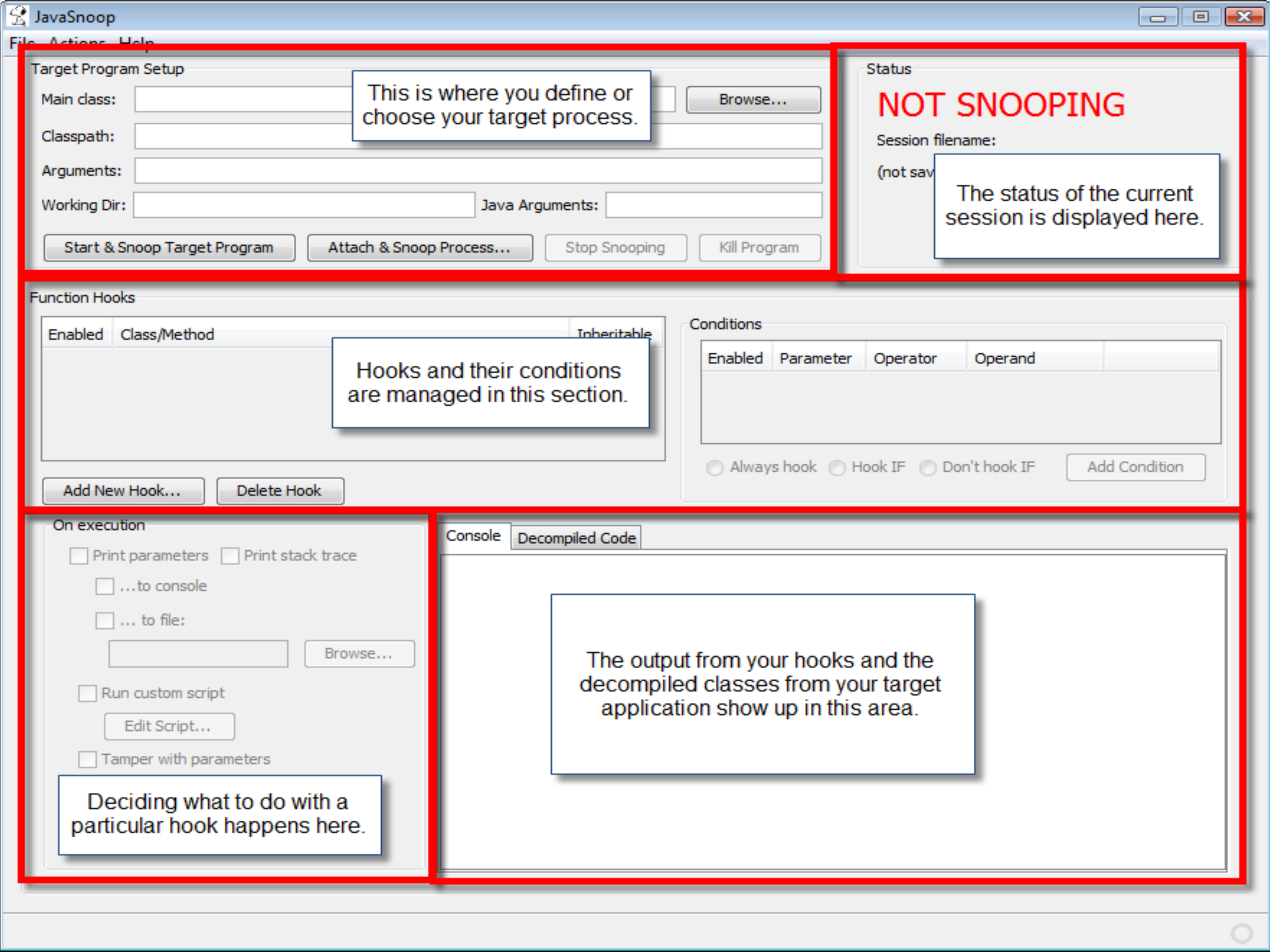
## JavaSnoop

JavaSnoop can use the Attach API and the Instrumentation class (helps in modification of a JVM during runtime) to jump into another JVM on the machine and install various “hooks” throughout class methods on that system. These hooks are then used by an agent to communicate with a GUI that allows the JavaSnoop user to “intercept” calls within the JVM.

# Bytecode Hacking Tools

The hooking technique used by JavaSnoop can perform the following actions:

- Edit method parameters
- Edit method return value
- Pause method
- Execute user-supplied script at the beginning of the method
- Execute user-supplied script at the end of the method
- Print the parameters to the console (or to a file)


JavaSnoop
File Actions Help
Target Program Setup
Main class:
Browse...
Classpath:
Arguments:
Working Dir:
Java Arguments:
Start & Snoop Target Program
Attach & Snoop Process...
Stop Snooping
Kill Program
This is where you define or choose your target process.
Status
NOT SNOOPING
Session filename:
The status of the current session is displayed here.
Function Hooks
Enabled
Class/Method
Hooks and their conditions are managed in this section.
Conditions
Enabled
Parameter
Operator
Operand
Always hook
Hook IF
Don't hook IF
Add Condition
Add New Hook...
Delete Hook
On execution
Print parameters
Print stack trace
...to console
... to file:
Browse...
Run custom script
Edit Script...
Tamper with parameters
Deciding what to do with a particular hook happens here.
Console
Decompiled Code
The output from your hooks and the decompiled classes from your target application show up in this area.

# Extra Reading

## Instrumentation on a virtual machine from a host

Bochs - a portable x86 and x86-64 IBM PC compatible emulator and debugger.

./configure [...] --enable-instrumentation
./configure [...] --enable-instrumentation="instrument/stubs"

C++ callbacks occur when certain events happen:

- Poweron/Reset/Shutdown
- Branch Taken/Not Taken/Unconditional
- Opcode Decode (All relevant fields, lengths)
- Interrupt /Exception
- Cache /TLB Flush/Prefetch
- Memory Read/Write

See also "bochs-python-instrumentation" patch

# Extra Reading

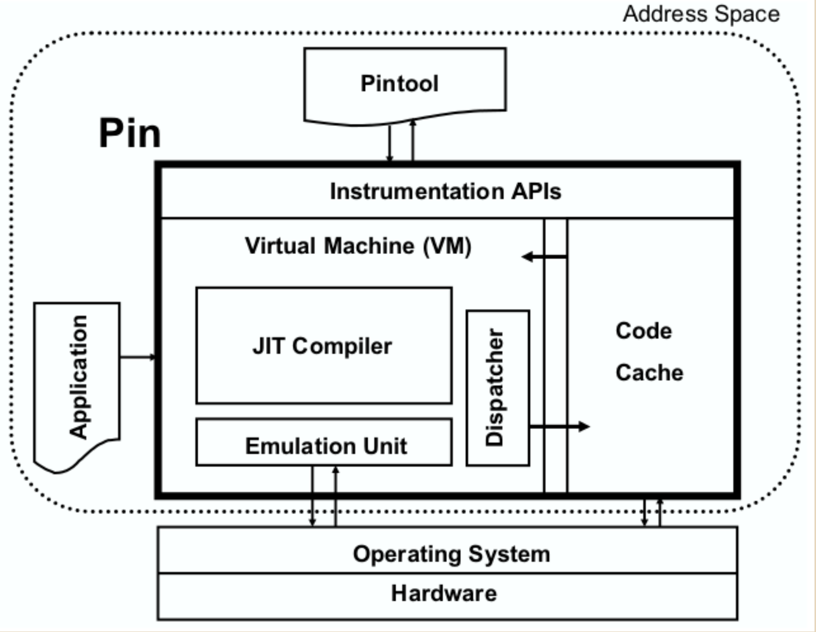
## Building and running Pin on Android

Intel released several documents for building and running Pin on Android titled "A Dynamic Binary Instrumentation Engine for the ARM Architecture" and "Pin for Android (Pindroid)".

Download pin-2.12-56759-gcc.4.4.3-android.tar.gz

Look inside "./source/tools"
and look at the tools they provide

Take a look at the software architecture of pindroid



# Questions

## Questions?

## Try some exercises at home

http://goo.gl/wPab4

Email: hughpearse@gmail.com
Twitter: https://twitter.com/hughpearse
LinkedIn: http://ie.linkedin.com/in/hughpearse