Forensics and Malware Analysis

Malware Analysis I

Mihai Ordean / Sujoy Sinha Roy
University of Birmingham

m.ordean@cs.bham.ac.uk

UNIVERSITY OF BIRMINGHAM

Ref: Coursera - Malicious Software and its Underground Economy: Two Sides to Every Story

Very brief history of Malware

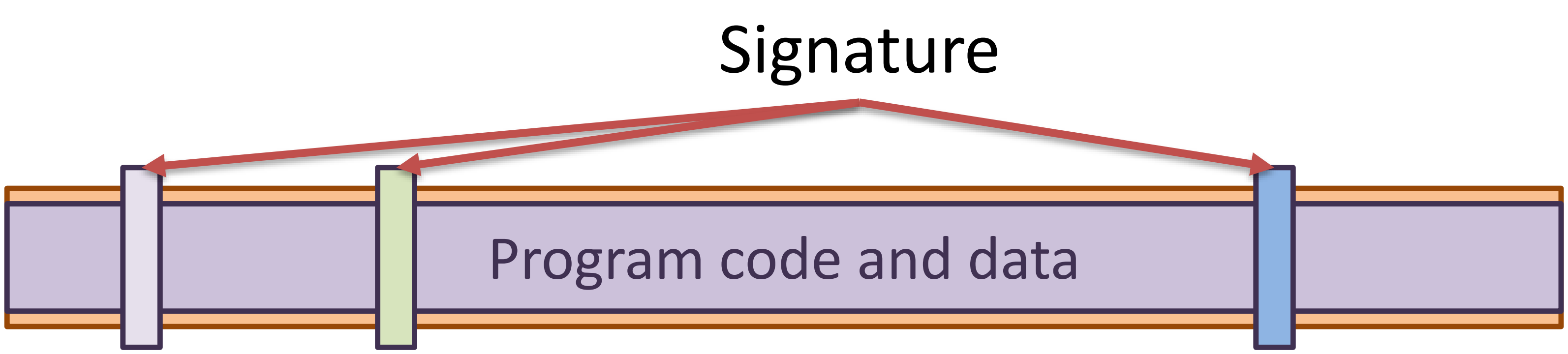
- Era of Discovery
  - Reason for malware: show skills, curiosity, no intentional harm, no financial gain
  - Anti-Malware: small scale reverse engineering
  - Ex: BrainBoot, Internet Worm

Very brief history of Malware

- Era of Transition
  - Reason for malware: show skills, no intentional harm, no financial gains.
  - Ex. Melissa, CIH, ILoveYou, etc.
  - Anti-Malware:
    - reverse engineering
    - signature based detection

Very brief history of Malware

- Era of Transition (cont.)
  - signature based detection
    - Instruction level signatures
    - Heuristics
    - Wildcards monitoring of specific files



Very brief history of Malware

- Current Era
  - Reason for malware: mostly financial gain
    - Malware more difficult to analyse
      - code obfuscation, polymorphism
      - tools to automatically mutate malware
  - Anti-Malware:
    - reverse engineering
    - signature based detection
    - increased computing power and sensors
    - still not behavior based

Architecture Reminder

- Memory description (user space + kernel space, page files, page files permissions, etc.)

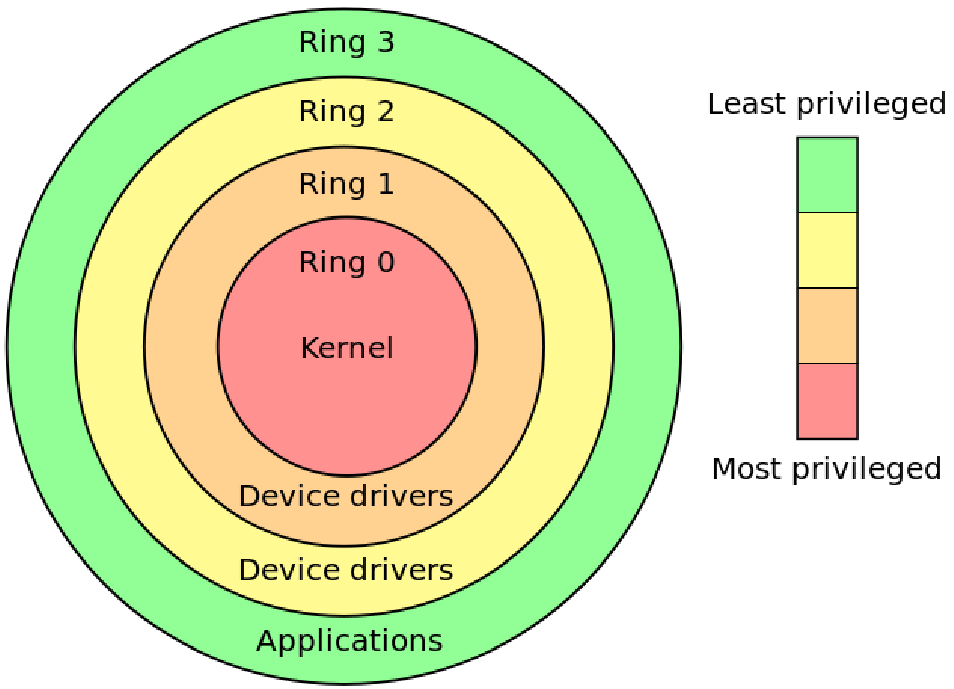
- Privilege levels/rings (0-3 in intel)
- User space at ring 3
- Kernel at ring 0
- System calls transfer control from user space to kernel space (in a OS controlled way)

Architecture Reminder

- Protection rings, are mechanisms to protect data and functionality from faults (by improving fault tolerance) and malicious behavior.

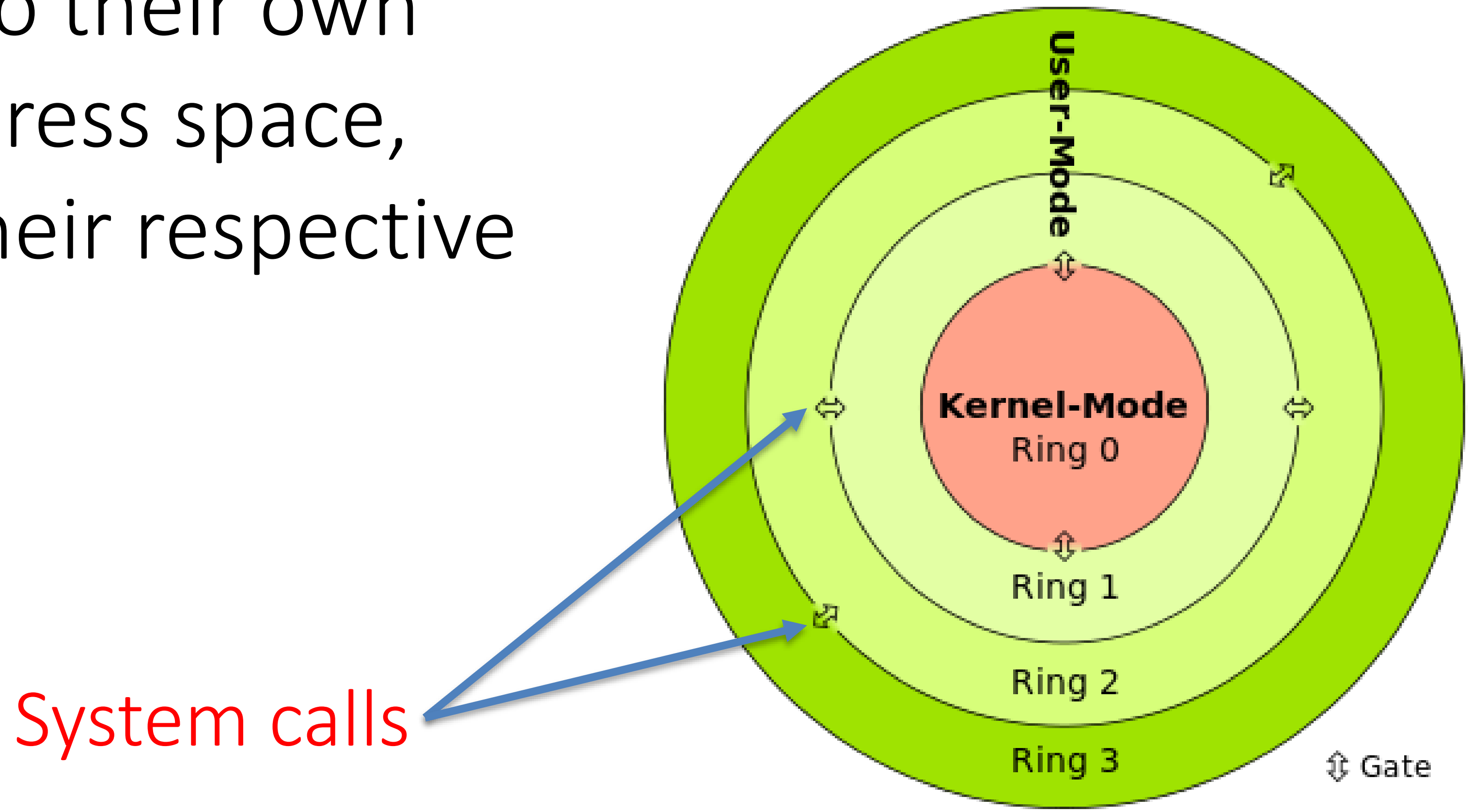
Architecture Reminder

- Ring0:
  - most privileges
  - interacts most directly physical hardware
- ....
- Ring3:
  - least privileges
  - Uses system calls to interact with physical hardware



Architecture Reminder

- Programs are (usually) limited to their own own address space, within their respective ring.



System calls

- System calls: controlled entry points into the kernel, which allow a process to request that the kernel perform some action on the process's behalf.

- The kernel makes a range of services accessible to programs via the system call application programming interface

System calls as APIs

- OS provide a API to access system calls
- API is implemented a library
  - Unix: glibc; Windows: ntdll.dll

- The library wrapper functions use ordinary function calling convention
  - i.e. ASM subroutine calls

Architecture Reminder

- Types of System calls:
  - process control (e.g. start /stop process)
  - file management (e.g. read/write to disk)
  - device management (e.g. interact with hardware)
  - information maintenance (e.g. set time, get proc. info)
  - communication (e.g. establish remote connections)

Function calls

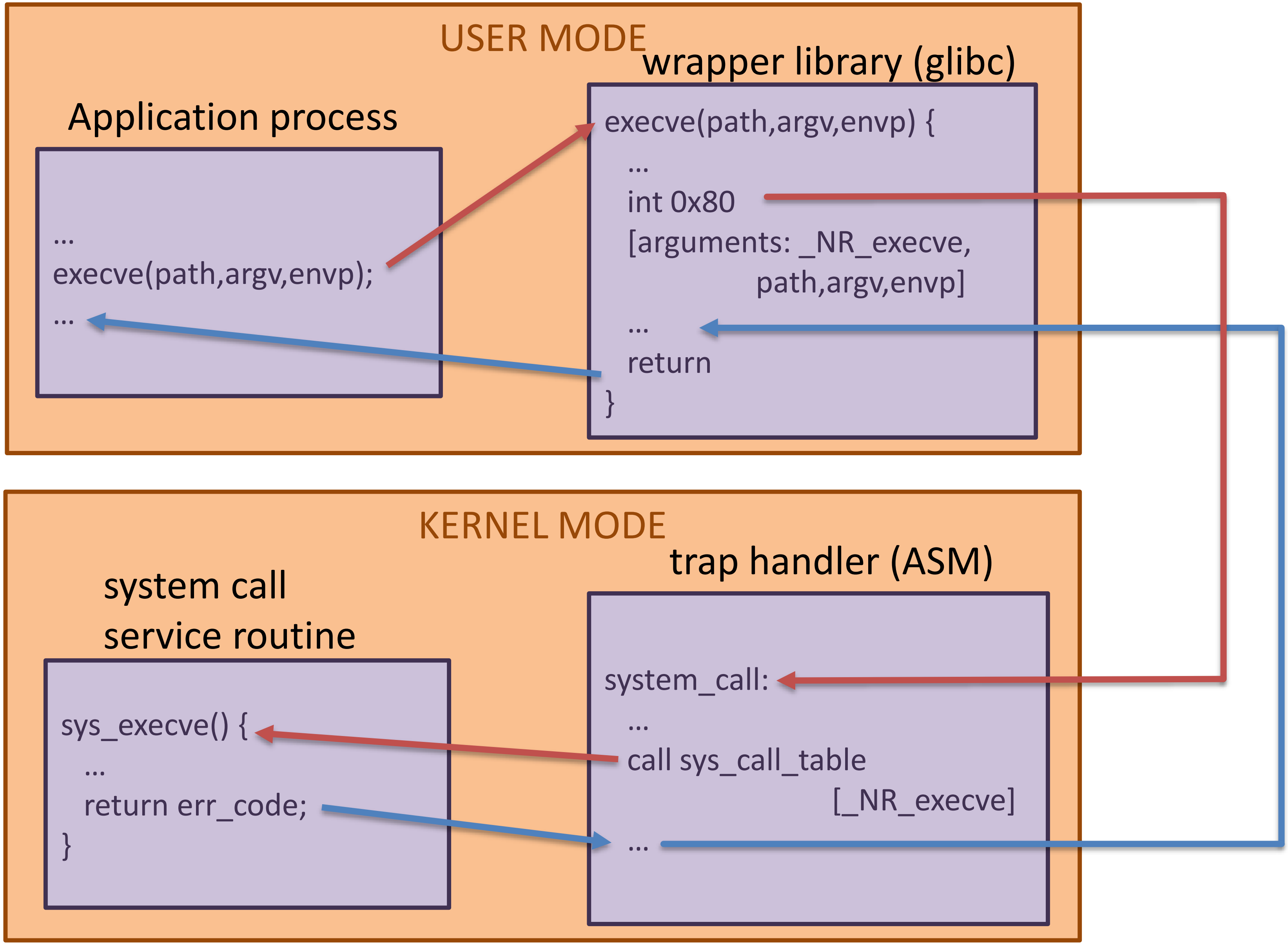
- C library functions like **printf** and **scanf** are wrappers around **system calls**.

- … but not all C library functions execute system calls e.g. **string.h** library functions, including **strcmp**, are **function calls**.

Executing a system call

- **The standard method**: the library function executes a trap machine instruction (int 0x80).
  - this causes the processor to switch from user mode to kernel mode and execute code pointed to by location 0x80 of the system's trap vector.
- **The sysenter instruction:** a faster method of entering kernel mode than the conventional int 0x80 trap instruction.
  - In Linux sysenter is supported in the 2.6 kernel and from glibc 2.3.2 onward.

Executing a system call



- C function **execve:** syscall that launches a new process

Quick intro to reverse engineering

How to analyse malware

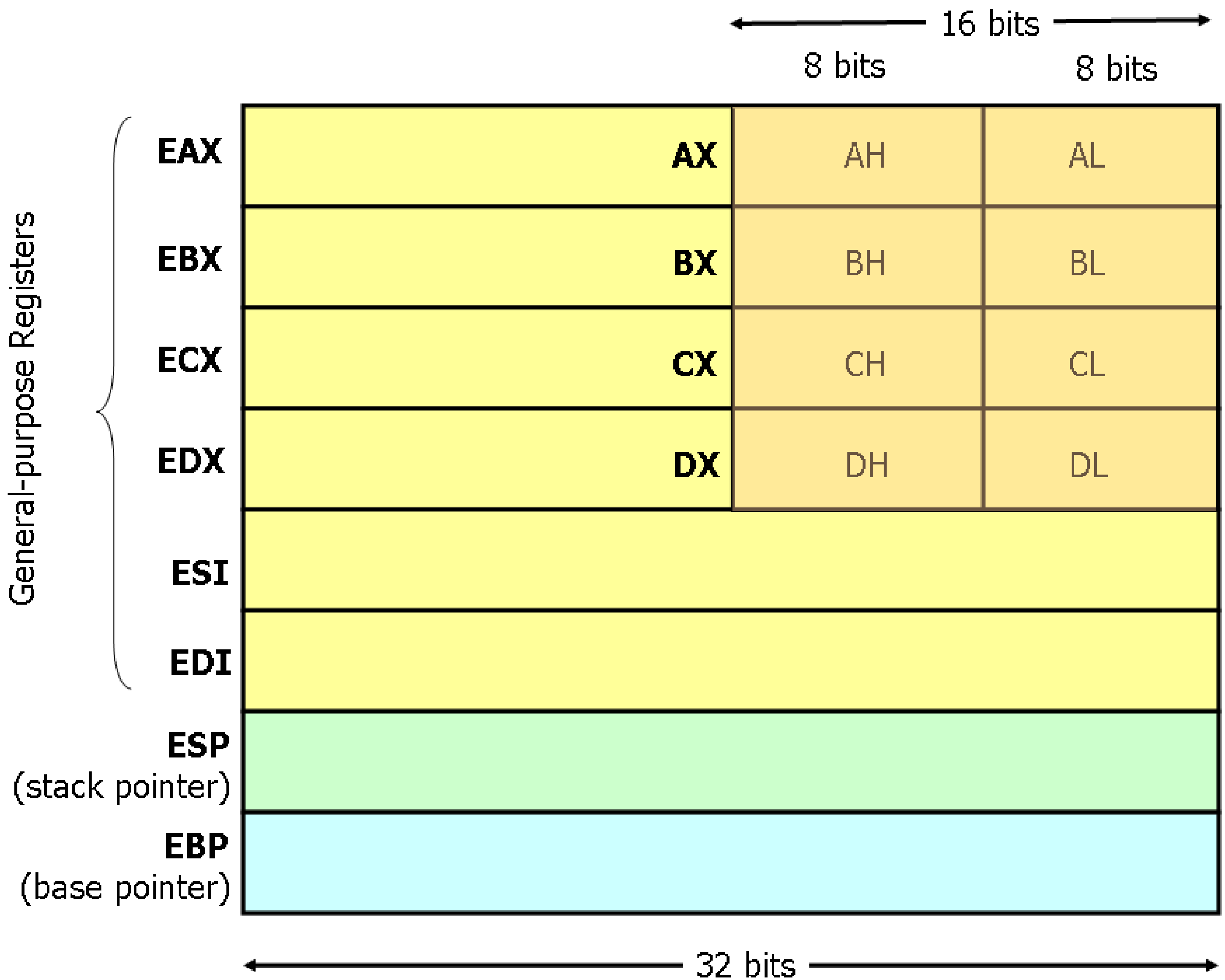
- Malware is usually written in ASM, C, etc.

- ASM
  - Low level symbolic language
  - Processor specific
  - Directly translated into binary format (1 to 1 mapping to machine language)

ASM

- Different types of instructions
  - Data transfer: mov, xchg, push, pop…
  - Binary arithmetic: add, sub, mul, div, inc, dec,…
  - Logical: and, or, xor, not
  - Control transfer: jmp, jne, call, ret, int,…
  - Input/output: in, out
- CPU/HW Registers
  - General purpose registers: (r)/(e)**ax**, (r)/(e)**bx**, (r)/(e)**cx**, (r)/(e)**dx** and **r8** to **r15** in x64 CPUs
  - Instruction pointer: (r)/(e)**ip**
  - Base pointer: (r)/(e)**bp**
  - Stack pointer: (r)/(e)**sp**

CPU registers

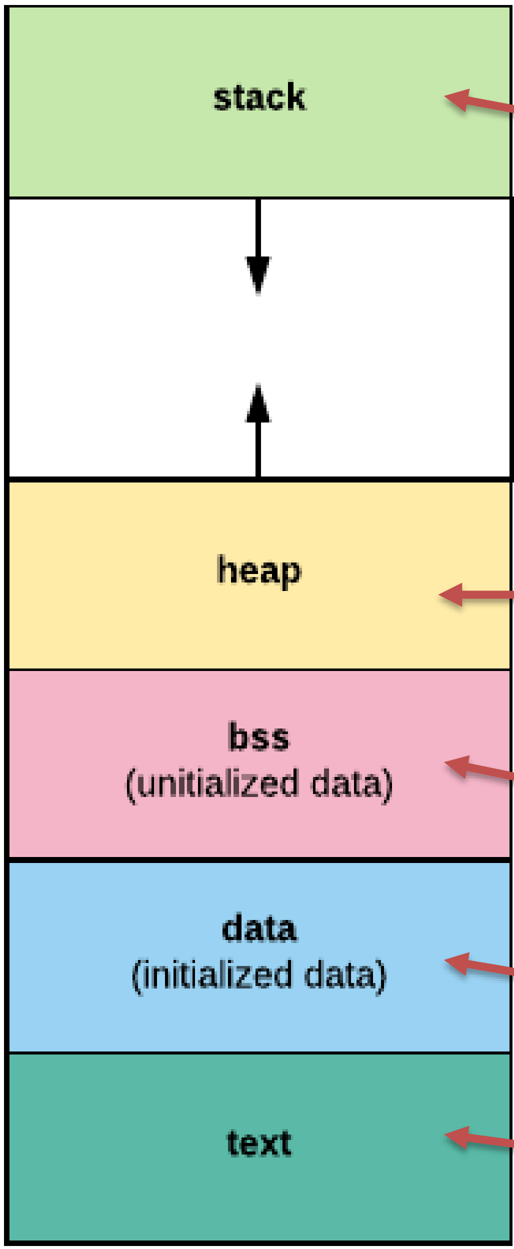


CPU registers & executable files

- CPU/HW Registers (cont.)
  - Flags: ZF, SF, CF…

- Object file Segment Registers:
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES (and FS, GS) – extra segment(s)

Executable files



STACK: Contains the program stack. Has a "stack pointer" register tracks the top of the stack. The stack area is adjoined to the heap area and they grew towards each other.

HEAP: heap area is shared by all threads, shared libraries, and dynamically loaded modules in a process.

BSS segment: global/static variables that are initialized to zero or do not have explicit initialization.

Data segment: contains global or static variables. Contains DS and ES.

Code segment: contains executable instructions, is RO.

ASM Syntax

- ASM – two ways of representing ASM instructions
  1. Intel (usually Windows)
     - syntax

       ```
       label: destination, source ; comment
       ```
     - pointers between [ ]
     - hex prefixed with 0x
  2. AT&T (usually Linux)

ASM Syntax

- ASM – two ways of representing ASM instructions
  1. Intel
  2. AT&T
     - syntax:

       ```
       label: source, destination # comment
       ```
     - pointers between ( )
     - numerical constants prefixed with $
     - hex prefixed with 0x
     - binary prefixed with 0b
     - registers prefixed with %

ASM

- Conditionals

```c
#include <stdio.h>

int main (int argc,
           char **argv){
   int a=1;
   if (a != 0){
       printf("A != 0\n");
   } else {
       printf("A = 0\n");
   }
}
```

```
.A:
   mov dword ptr [ebp - 8], 1 ; input a
.LC0:
   .string "A = 0\n"
.LC1:
   .string "A != 0\n"
main:
   [function prologue]
   mov al, byte ptr [ebp - 8]
   test al,al
   jnz .L2  ; jump if not 0
   mov ESP, [LC1]
   call printf
   jmp .L3  ; jump to end of the if (i.e. "}" )
.L2:
   mov ESP, [LC0]
   call printf
.L3:
   leave
   ret
```

ASM

- Loops

```c
#include <stdio.h>

int main (int argc, char ** argv){
   int i=1;
   while (i<10){
      i++;
   }
}
```

```asm
.main:
   [function prologue]
   mov dword ptr [ebp - 4], 1 ; input i
.LOOP:
   mov eax, dword ptr [ebp - 4]
   cmp eax, 10
   jge .L3
   inc eax
   mov dword ptr [ebp - 4], eax
   jmp LOOP
.L3:
   leave
   ret
```

ASM

- Functions

```
#include <stdio.h>

int main (int argc, char ** argv){
   return myfunc(1,2);
}
```

```
.main:
   ; save current program location &
   ; opt. EAX, ECX EDX (not here)
   push ebp      ; prologue for main
   mov ebp, esp ; equiv to: enter 0,0

   …
   ; prepare to run function
   push 2
   push 1
   call myfunc
   …

   ;restore program to saved location
   mov esp, ebp ;
   pop ebp       ; epilogue for main
```

ASM

- Functions

```c
#include <stdio.h>

int myfunc(a,b){ return a+b; }
int main (int argc, char ** argv){
   myfunc(1,2);
}
```

```
.myfunc:                ; stack situation
   ;[prologue]          ; after [prologue]
   mov eax,[ebp+4]
   mov ebx,[ebp+6]      ; ebp+0 => old ebp
   add eax, ebx         ; ebp+2 => ret addr
   ;[epilogue]          ; ebp+4 => 1
   ret 4                ; ebp+6 => 2
```

```
.main:
   ; save current program location &
   ; opt. EAX, ECX EDX (not here)
   push ebp      ; prologue for main
   mov ebp, esp ; equiv to: enter 0,0

   …
   ; prepare to run function
   push 2
   push 1
   call myfunc
   …

   ;restore program to saved location
   mov esp, ebp ;
   pop ebp      ; epilogue for main
```

ASM

- Functions

```
#include <stdio.h>

int myfunc(a,b){ return a+b; }
int main (int argc, char ** argv){
   myfunc(1,2);
}
```

```
.myfunc:
   ;[prologue]; changes stack point.
   mov eax,[ebp+4]
   mov ebx,[ebp+6]
   add eax, ebx
   ;[epilogue]; changes stack point.
   ret
```

Result is usually placed in EAX

```
.main:
   ; save current program location &
   ; optionally EAX, ECX EDX (not here)
   push ebp      ; prologue for main
   mov ebp, esp ; equiv to: enter 0,0

   …
   ; prepare to run function
   push 2
   push 1
   call myfunc
   …

   ;restore program to saved location
   mov esp, ebp ;
   pop ebp       ; epilogue for main
```

ASM

- Functions

```
Function prologue:

   push ebp ; save base pointer
   mov ebp, esp ; create new stack
   sub esp, N ; create N bytes on stack


or:
   enter N, 0
```

```
Function epilogue:

   mov esp, ebp ; restore space
   pop ebp ; restore base pointer
   ret  M; return to calling func,
           popping the extra m
           arguments
or:
   leave
   ret M
```

Disassembly

- Disassembly involves taking a binary blob, separating code and data and extracting the instructions.
- In a disassembled program we can
  - Locate functions
  - Recognize jumps
  - Identify local variables
  - i.e. understand the program's behaviour

Disassembly

- Disassembling programs is not an easy task
  - Separating code from data is very difficult.
    - e.g bitstream: … 0x8b 0x44 0x24 0x04 …

      ```
      mov eax, [esp+0x04]
      ```

    - 0x8b is data => 0x44 0x24 0x04

      ```
      inc esp
      and al,0x04
      ```

Disassembly

- Two main algorithms for doing disassembly
  - Linear sweep (objdump,…)
  - Recursive traversal (IDApro)

Linear sweep

- Linear sweep:
  - Locates instructions: where one instruction, begins another one ends
  - Assumes that everything that is marked as code is actually machine instructions (of course not true for malware!)
  - Disassembly starts from the first bit and continues in a linear fashion, i.e. instructions are disassembled one after the other until the end of the section is reached.

Linear sweep

- Linear sweep:
  - Pros:
    - Provides complete coverage of a programs code sections: e.g. calls are not followed because eventually the code for that call will be reached.
  - Cons:
    - Oblivious to the flow of the program.
    - Compilers often mix code and data which results in disassembled “junk” (e.g. a switch statement in C can translate to a jmp table in asm).

Linear sweep

08048350	B8 66 83 04 08	mov $0x8048366,%eax
08048355	FF D0	call *%eax
08048357	C3	ret
08048358	48	dec %eax
08048359	65	gs
0804835A	6C	insb (%dx),%es:(%edi)
0804835B	6C	insb (%dx),%es:(%edi)
0804835C	6F	outsl %ds:(%esi),(%dx)
0804835D	20 57 6F	and %dl,0x6f(%edi)
08048360	72 6C	jb 0x80483ce
08048362	64 21 0A	and %ecx,%fs:(%edx)
08048365	0D BA 0E 00 00	or $0xeba,%eax
0804836A	00 B9 58 83 04 08	add %bh,0x8(%ecx)
08048370	BB 00 00 00 00	mov $0x0,%ebx
08048375	B8 04 00 00 00	mov $0x4,%eax
0804837A	CD 80	int $0x80
0804837C	B8 00 00 00 00	mov $0x0,%eax
08048381	C3	ret

Linear sweep

08048350	B8 66 83 04 08	mov $0x8048366,%eax
08048355	FF D0	call *%eax
08048357	C3	ret
08048358	48 65 6C 6C 6F 20 57	*(data)*
0804835F	6F 72 6C 64 21 0A 0D	*(data)*
08048366	BA 0E 00 00 00	mov $0xe,%edx
0804836B	B9 58 83 04 08	mov $0x8048358,%ecx
08048370	BB 00 00 00 00	mov $0x0,%ebx
08048375	B8 04 00 00 00	mov $0x4,%eax
0804837A	CD 80	int $0x80
0804837C	B8 00 00 00 00	mov $0x0,%eax
08048381	C3	ret

Recursive traversal

Ref: Coursera - Malicious Software and its Underground Economy: Two Sides to Every Story

Recursive traversal

- Recursive transversal “follows” the control flows.
- Types of control flows
  - Sequential flow: pass execution to the next instruction that follows immediately (mov, push, add)
  - Conditional branching: set of instructions that follow after a jump instruction
  - Unconditional branching: set of instruction that would happen if jump is not executed.

Recursive traversal

- Types of control flows (cont.)
  - Function calls: set of instructions following a `call` instruction
  - Returns: recursive transversal disassembles programs based on calls and jumps. Every call is placed in a “stack” and once the flow is disassembled completely the disassembly resumes from the stack (i.e. the recursive part).

Recursive traversal

- Recursive traversal:
  - Pros:
    - distinguish code from data
    - enables creation of flow graphs
  - Cons:
    - Some parts of the program may not be disassembled.
    - Inability to follow indirect code paths.

Lets see it in action!

- objdump disassembly

  ```
  objdump –d desktop/examples/reveng/r1 | less
  ```

  1. AT&T syntax
  2. Disassembled using linear sweep

Lets see it in action!

- IDAPro disassembly
  - `Open IDAPro`
  - `Load exercise`
    `desktop/examples/reveng/r1`

1. Intel syntax
2. Disassembled using Recursive transversal: flow graph, code segment, data segment, strings,…
3. Compare IDAPro and objdump output

GHIDRA

- https://ghidra-sre.org/
- https://github.com/NationalSecurityAgency/ghidra