Forensics, Malware, and Penetration Testing

Malware Analysis

Mihai Ordean
University of Birmingham

m.ordean@cs.bham.ac.uk

Ref: Computer Security by Bishop; Introduction to Malware, UT Dallas

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

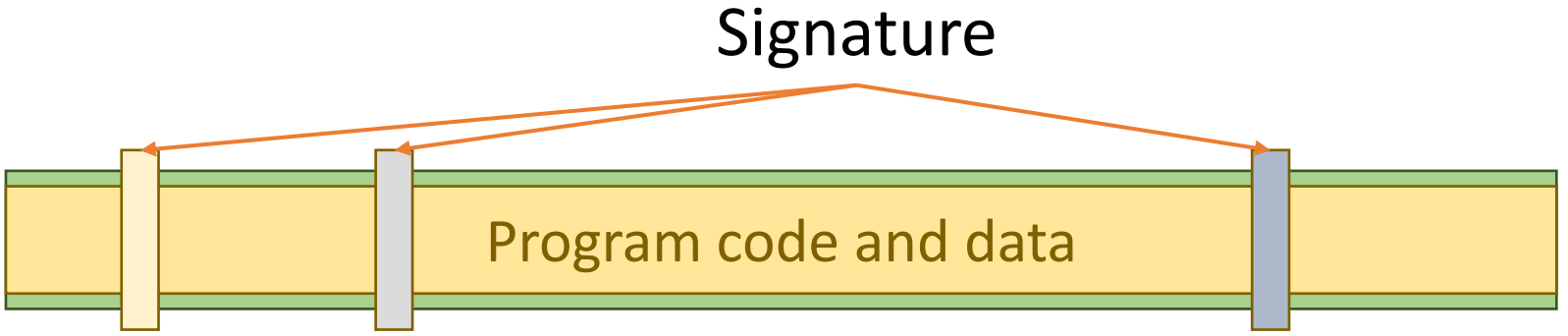
Era of Transition

- 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

Static reverse-engineering

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
```

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.

System calls

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

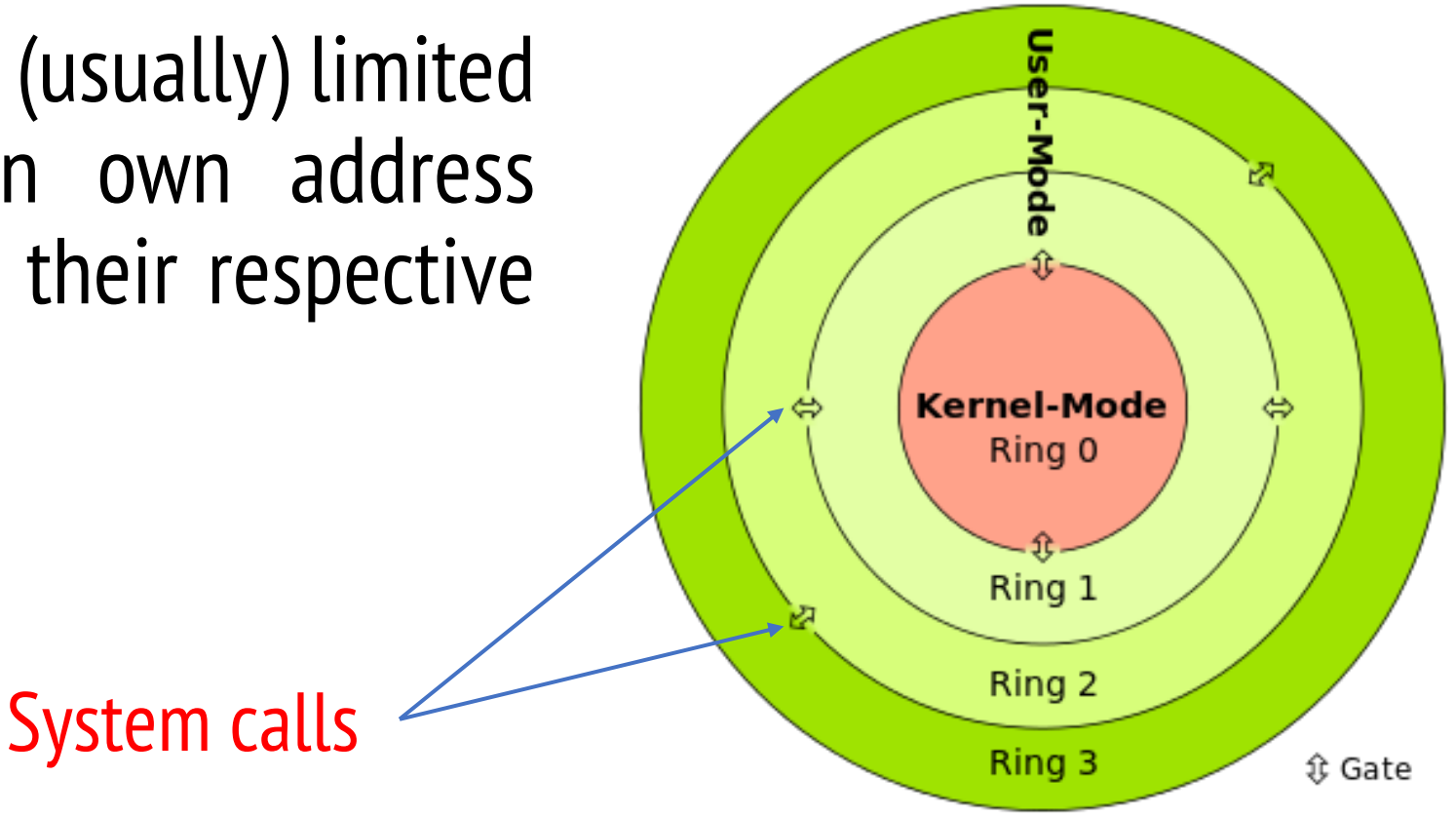
The standard method: the library function executes a trap machine instruction (int 0x80).

- 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.

System calls

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



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

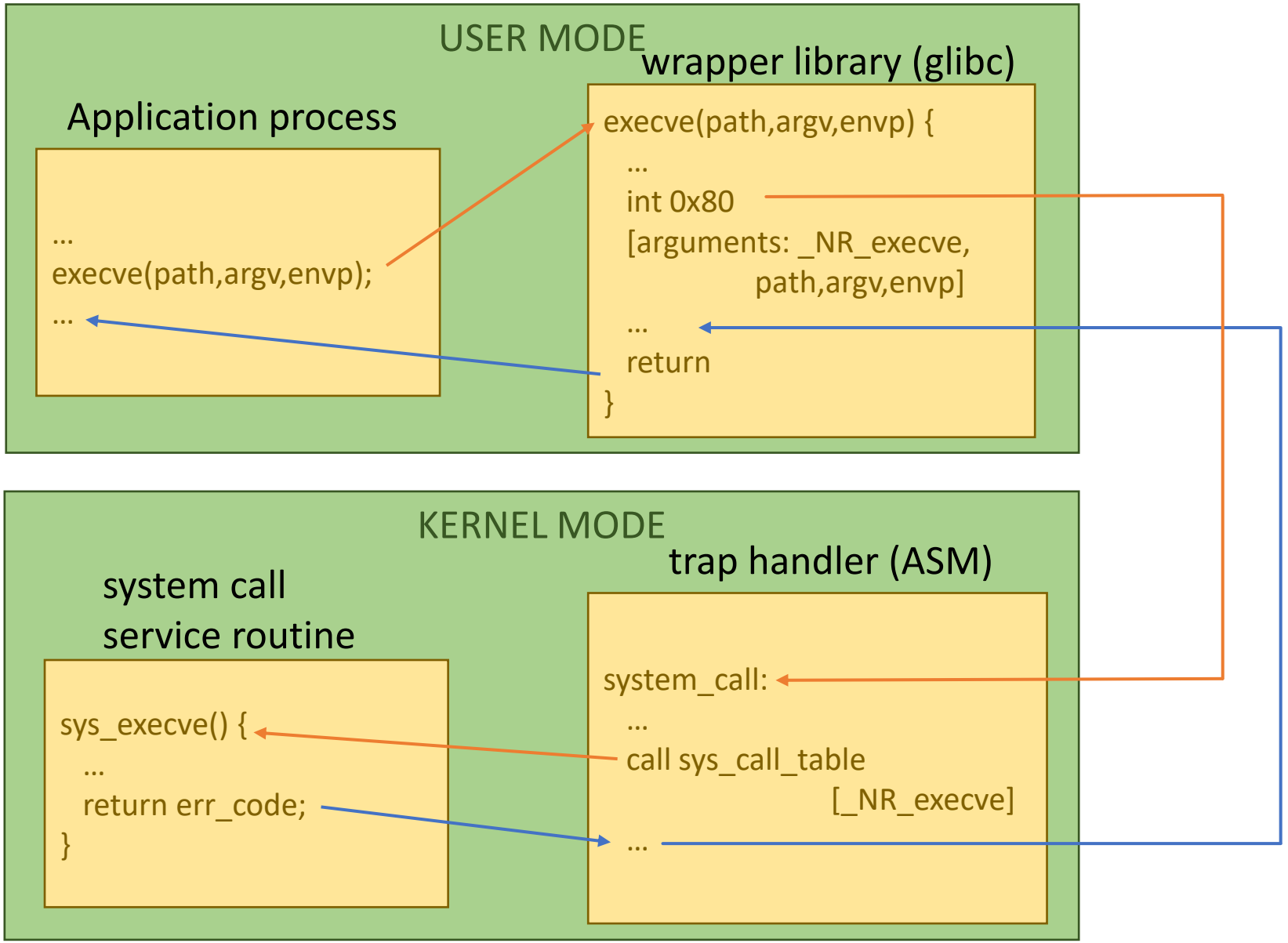
- i.e. ASM subroutine calls

System calls

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)

Executing a system call



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

System calls vs 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**.

Lets see it in action!

objdump disassembly

- In a terminal run:

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

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

Lets see it in action!

IDAPro disassembly

- Load exercise in `IDAPro`
  `desktop/examples/reveng/r1`

1. Intel syntax
2. Disassembled using **recursive transversal**: flow graph, code segment, data segment, strings,…

Limitations

Polymorphism & Metamorphism

Polymorphic malware

- Change layout with each encryption
- Encrypt their payload (using XOR based encryption)
- Use different keys for each infection
- **Makes static string analysis impossible**
- **Encryption routine either:**
  - **changes with each generation**
  - **is very common such that it produces a lot of false positives**

Polymorphism & Metamorphism

Metamorphic malware: the whole malware changes

- each instruction is replaced with another, semantically equivalent instruction
- the instructions are syntactically different
- producing a signature is impossible
  - signatures can be produced for a specific instance of the metamorphic virus but not for the whole “family”

Code obfuscation

- **Code obfuscation** refers to techniques that **preserve the program's semantics and functionality**, but greatly **increase the difficulty of understanding the program's structure.**

Anti static-analysis techniques

Code obfuscation

Anti static-analysis techniques

- Junk insertion: introduce disassembly errors by inserting junk bytes at selected locations into the code stream where the disassembler expects code
- Branch function obfuscation: modify the normal behaviour of a function call
- Overlapping instructions: sharing of code on different levels

Junk insertion

Junk:

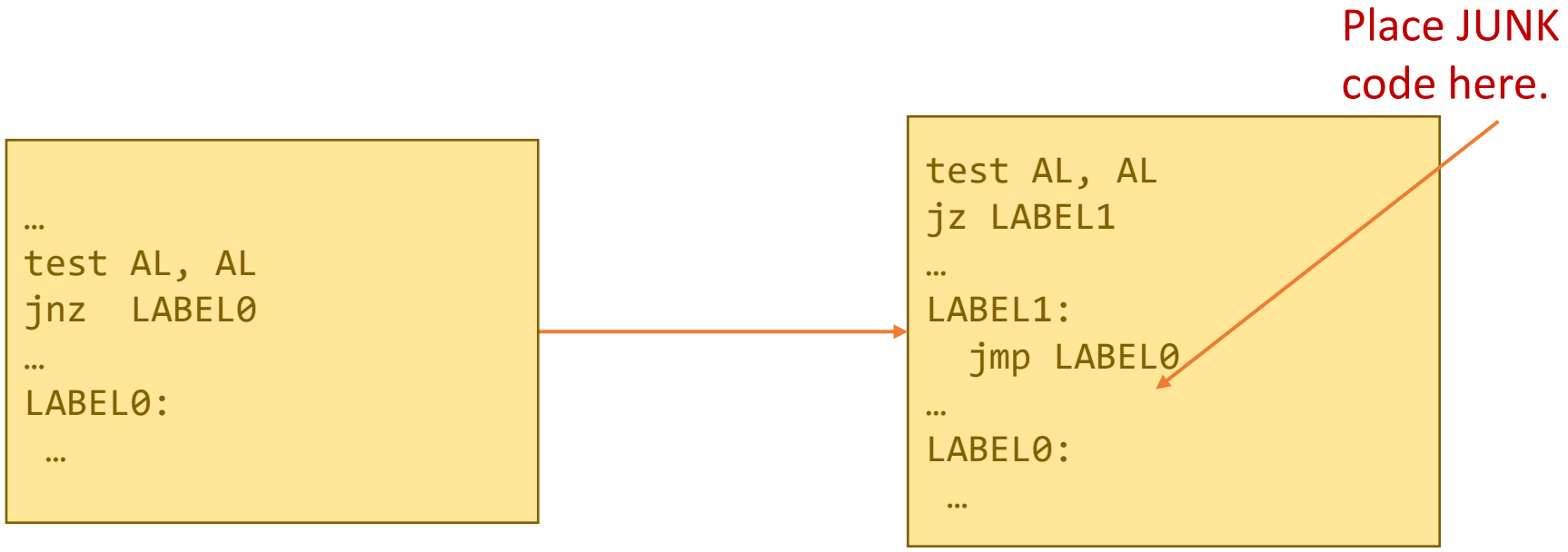
- partial instructions
- inserted such that they the instructions are not reachable at runtime (otherwise SEGFAULT)

Scenario

1. Pick a candidate block “B” to protect
2. Insert junk before block “B”
3. Code immediately before junk must be an unconditional jump

Junk insertion

- Fortunately, ASM contains error correcting functionality.
- Junk insertion "can be maximised" by:
  - inverting conditional jumps, or branch flipping to increase the number of candidates



Junk insertion

Excellent way of interfering with linear sweep

Does not affect recursive flow disassembly (mostly…)

Branch functions obfuscation

Branch functions obfuscation

- Assumption: disassembler assumes reasonable behaviour of control transfer logic (i.e. `JMP` instructions) – 2 targets:
  - target pointed by the label
  - target that follows after the `JMP` instruction
- Exploit: the difficulty of identifying indirect control transfers

Branch functions obfuiscation

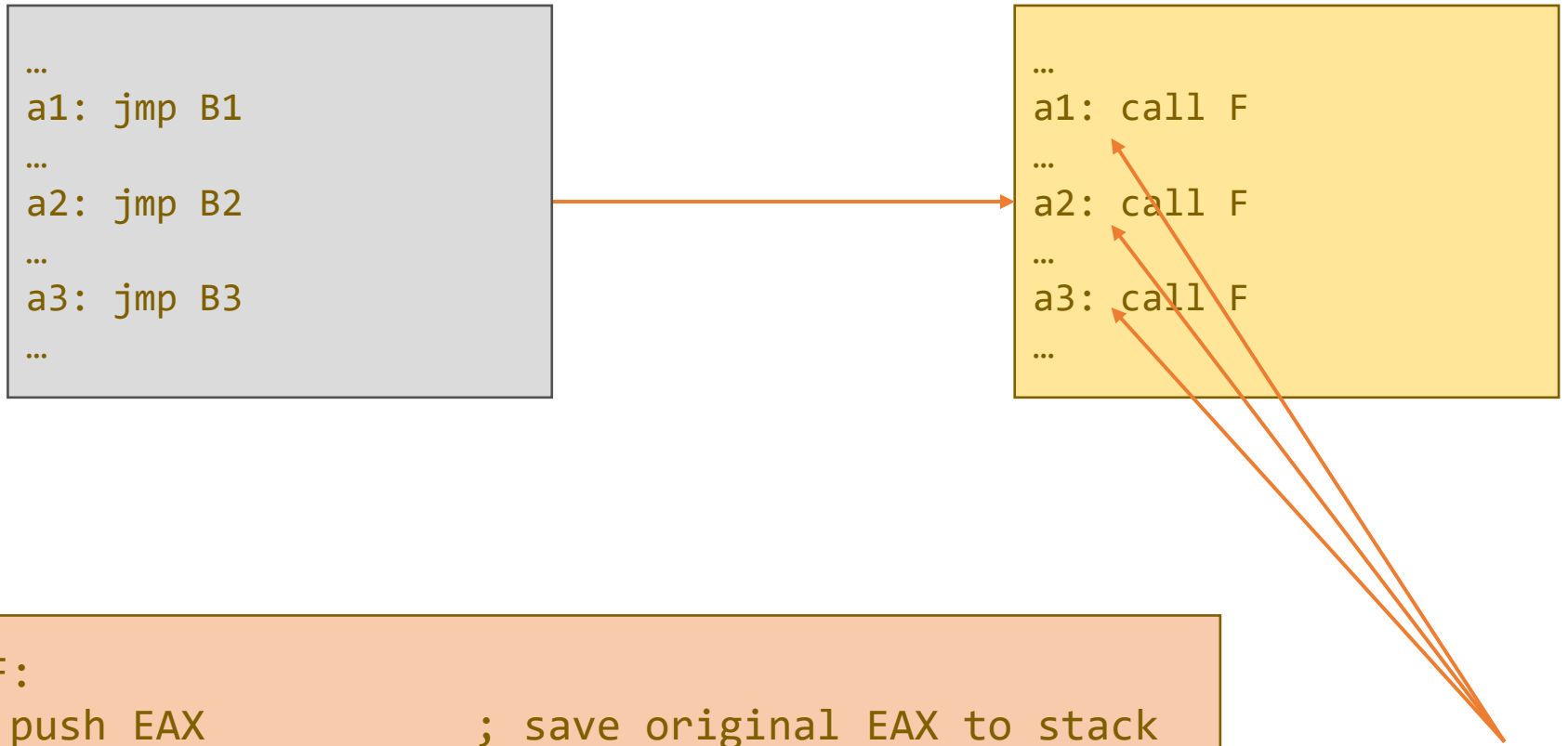
Exploitation techniques:

- opaque predicates: transform unconditional jumps into conditional jumps with predicates that always evaluate to the same value.
- jump table spoofing: insert artificial jump tables to mislead the disassembler (e.g. some jumps might point to junk bytes)
- branch functions: encode the logic of the original JMP instruction into a **branch function**, in a way that can only be evaluated dynamically

Branch functions


```
F:
 push EAX              ; save original EAX to stack
 mov EAX, [ESP+8*n]    ; get offset to target Bn
 add [ESP+4], EAX      ; add offset to rtrn address
 pop EAX               ; restore original EAX
 ret                   ; jump to target i.e.
                       ; ret<-take eip and add esp,4
```

Branch functions




```
F:
 push EAX             ; save original EAX to stack
 mov EAX, [ESP+8*n]   ; get offset to target Bn
 add [ESP+4], EAX     ; add offset to rtrn address
 pop EAX              ; restore original EAX
 ret                  ; jump to target
```

Overlapping instructions

original code

```
binary    asm

C1E102    shl ecx,0x2
41        inc ecx
C3        ret
```

inject
- partial junk instruction byte B801
- jump instruction EB02

```
binary    asm

C1E102    shl ecx,0x2
EB02      jmp short X
B801      db:0xb0,0x01
41        inc ecx
C3        ret
```

Overlapping instructions

original code

```
binary    asm

C1E102    shl ecx,0x2
41        inc ecx
C3        ret
```

inject
- partial junk instruction byte B801
- jump instruction EB02

```
binary    asm

C1E102    shl ecx,0x2
EB02      jmp short X
B801      db:0xb0,0x01
41        inc ecx
C3        ret
```

disassembled code

```
binary    asm

C1E102    shl ecx,0x2
EB02      jmp short X
B8        db:0xb8
0141C3    add [ecx-0x3d], eax
```

Packing techniques

Packing

Malicious code hidden by multiple layers (1+) of compression/encryption

- An evolution of polymorphism
- 80% of malware is packed

Example of open-source exe packer: https://en.wikipedia.org/wiki/UPX

Packing

Malicious code hidden by multiple layers (1+) of compression/encryption

- An evolution of polymorphism
- 80% of malware is packed

Packing

Malicious code hidden by multiple layers (1+) of compression/encryption

- An evolution of polymorphism

Packing

Malicious code hidden by multiple layers (1+) of compression/encryption

- An evolution of polymorphism
- 80% of malware is packed

Anti-malware software must implement unpacking routines to be able to identify the malicious code

- Problem is that there are over 200 families with more than 2000 members in each (reported by Symantec)

Packing

Malicious code hidden by multiple layers (1=) of compression/encryption

- An evolution of polymorphism
- 80% of malware is packed

Anti-malware software must implement unpacking routines to be able to identify the malicious code

- Problem is that there are over 200 families with more than 2000 members in each (reported by Symantec)
- **Packing can be used in conjunction with polymorphism and metamorphism.**

Algorithm agnostic unpacking

Emulate/trace the execution until the unpacking routine finishes (e.g. OmniUnpack).

1. Start monitoring the malware
2. Remove all permissions from all the memory pages belonging to the malware
3. When malware attempts to access one of these pages, intercept the access and reason about the behaviour (execute code, read data, write data)

Algorithm agnostic unpacking

Emulate/trace the execution until the unpacking routine finishes (e.g. OmniUnpack).

Reasoning: analyse the **system calls** of the process

- Two types of system calls:
  - Safe: e.g. read from to pagefile, write to own pagefile (i.e. unpack), read external file
  - Unsafe: e.g. write file, setup socket
- When unsafe behaviour is detected stop the process and run an anti-virus targeting the unsafe code

Dynamic analysis

Dynamic analysis

Problem: signature based anti-malware and static reverse engineering techniques are easy to defeat.

Context knowledge: malware usually targets system calls, mostly by hijacking libraries.

Dynamic analysis

Dynamic analysis:

- Monitor the system calls executed by processes: this is the only way a processes can interact with OS.
- Infer the behavior from these events
- Detect the high-level malicious behaviour

Dynamic analysis can detect novel malware because it characterize generic high-level malicious behaviour (e.g. spam, sensitive information stealing,…)

Dynamic analysis

Allows us to:

- **Monitor** code and data flow as it executes
- **Reason** about actual execution
- **Perform** precise security analysis using the runtime information

Techniques

- **Program instrumentation: add extra semantic preserving code and/or perform taint tracking**
- **Debugging tools: allow programmers to see "what's going on"**

Taint analysis

Taint analysis describes how “interesting” information flows throughout a programs execution using the following components:

- Taint source (the interesting data)
- Propagation rules (“how the flowing happens”)
- Taint sinks

Helps determine the where data arrives, where security policies should be enforced, etc.

Debugging

Debugging: a way to analyze the execution of a program

Debugging tools usually allow:

- **Monitoring of a single process execution**
- **Set multiple debugging levels**: e.g. single step
- **Set breakpoints**: stop the program at a specific point
- **Set watchpoints**: stop a program when the value of an expression changes
- **Set catchpoints**: stop a program when an event occurs
- **Watch CPU registers and environment**

Tools

- **Linux: gdb**
- **Windows: OllyDBG, WinDBG**
- **Other tools:**
  - **PTRACE**: an (Linux) API that allows one process to control another process. Windows has an equivalent API: Event Tracing for Windows (ETW)
  - **ltrace**: (in Linux) intercepts and records dynamic library calls which are called by the executed process and signals which are received by that process
  - **strace**: (in Linux) monitors the system calls and signals of a specific program and provides the execution sequence of a binary from start to end.

Breakpoint

A breakpoint is a software interrupt: int3

- CPU transfers control to the routine associated to the interrupt
- When the routine ends the execution resumes to the instruction following interrupt

Debuggers use break exceptions to suspend execution of a program and allow one to inspect memory locations, registers, etc…

Observing debugging

- Can a process detect if its being traced?

```
void handler(int x){}
int main(){
    signal(SIGTRAP,handler);
    asm("int3;");
    printf ("lets do stuff...\n");
}
```

Run

1. ./antiint3
2. gdb ./antiint3

Yes, by setting up a trap on the int3 interrupt!

Observing debugging

ltrace:

substitutes calls to external libraries with int3 to halt execution and allow ptrace to get information about the traced process.

```
1.  #include <stdio.h>
2.  #include <unistd.h>
4.  int main() {
5.     FILE *fp = fopen("rfile.txt", "w+");
6.     fprintf(fp+1, "Invalid Write\n");
7.     fclose(fp);
8.     return 0;
9.  }
```

Run

```
1. ./ltraceex
2. ltrace ./ltraceex
```

Observing debugging

ltrace:

substitutes calls to external libraries with int3 to halt execution and allow ptrace to get information about the traced process.

```
1.  #include <stdio.h>
2.  #include <unistd.h>
4.  int main() {
5.      FILE *fp = fopen("rfile.txt", "w+");
6.      fprintf(fp+1, "Invalid Write\n");
7.      fclose(fp);
8.      return 0;
9.  }
```

Run

```
1. ./ltraceex
2. ltrace ./ltraceex
```

Find out exactly which library call crashes

Anti-debugging

PTRACE

- Limitation: a process can only have a single `PTRACE` parent.
- Exploit: self trace and check if anything else wants to start tracing. If **yes** PTRACE will return an error.

Run

```
1. ./antiptrace
2. ltrace ./antiptrace 2>/dev/null
```

Tools: ptrace - process trace

- Check if tracing is allowed

  Normal `ptrace` usage

```
void runDebug (const char* prog){
   //enable self tracing
   fork()
   if (ptrace(PTRACE_TRACEME,0,NULL,NULL)<0){
      perror("ptrace")
      return;
   }
   execl(prog,prog,0)
}
```

Whenever a system call is hit the parent process will be informed of what's happening in the traced process

Anti-debugging

PTRACE

- Limitation: a process can only have a single PTRACE parent.
- Exploit: self trace and check if anything else wants to start tracing. If **yes** PTRACE will return an error.

```
#include <stdio.h>
#include <sys/ptrace.h>

int main(){
   if (ptrace(PTRACE_TRACEME,0,NULL,NULL)<0) {
      printf("you are debugging me!\n");
      return 1;
   }
   printf("lets start doing weird stuff...\n");
   return 0;
}
```

Anti-anti-debugging

Can be used if the process implementing anti-debugging uses externally linked libraries.

Intuition (e.g. **PTRACE**):

- Compile a custom `ptrace` library which always returns 0.
- Instruct the debugged program to use the custom file rather than the "real" `ptrace`.

Run:

```
LD_PRELOAD=./ptrace.so ltrace ./antiptrace 2>/dev/null
```

Anti-anti-debugging

Can be used if the process implementing anti-debugging uses externally linked libraries

Intuition (e.g. **PTRACE**):

PTRACE reference header: `ptrace(PTRACE_TRACEME,0,NULL,NULL);`

```
long ptrace(int a, int b, int c, int d){
   return 0;
}
```

Some limitations

Only parts of the code that happen can be analyzed

- Current scientific challenge is to detect “the untaken paths”

Logic bombs can be hidden inside the malware which deactivate the malicious code if analysis is detected

Use code obfuscation to hide malicious code: i.e. by using encrypted code to activate a branch

Conclusion

We looked at a few techniques to perform dynamic analysis

We saw techniques to prevent dynamic analysis

Manny more ways to do it

- Sandboxing
- VM debugging
- ....

Dynamic analysis also has some limitations