Bell LaPadula: (Confidentiality)

Label Compartments

| TOP SECRET | JFK   |
|            | Area 51 |
| SECRET     | FOUO   |
|            | Public |

- Object (L, CS)
- Subject (L’, CS’)
- If L’ >= L && CS’ subset of CS access is allowed.
- No write down Policy.
- **Bell LaPadula is guaranteed to never accidently disclose information.**

Suppose a user open an editor
- Because of I/O to user, may need to label based on user.
- Initial label = min label = Public
- Label floats up as it opens more files.
- Float label on files too.

BIBA Model: (Integrity)

- No Read Down
- No write Up

E.g.

| Theorem         | --- Most Trust Worthy |
|                 | First Hand            |
|                 | Second Hand           |
|                 | Internet              | --- Least Trust Worthy |
**Role Based Access Control (RBAC):**

- User → Roles
- Roles → Permission

E.g.

- Alice → Manager → Withdraw
- Bob → Teller → Deposit
- Carol → Make Loan
- Dave → Auditor → Read Logs

- So in this case if a manager makes a loan for himself then there should be some other authority may be Auditor approving that. A manager should not be able to approve loans requested by himself.

**Isolation with Message Passing:**

- Kernel
- Processes
- Memory Isolation
- Message Passing
Software bugs are source of all security vulnerabilities:

- Buffer Overflow
- Format String
- Race Conditions
- Confused Deputy
- SQL Injections
- XSS Bugs
- CSRF
- Component Hijacking

Buffer Overflow:

- Process Virtual Memory Layout

```
1 GB

Kernel

stuff

Env variables

Stack
(Grows Downwards)

\______________________________________________________________________________

(Grows Upwards)
Heap

.bss

readonly data

.data

.text

Unallocated VM
```
Stack Diagram for an e.g. function call:

```c
void dummy(int x, int z)
{
    int w;
    w = x + z;
}

void foo(void)
{
    int y = 7;
    dummy(y, y+1);
}

char *getuser(void)
{
    char buf[256];
    gets(buf);
    printf("Hello%s\n", buf);
    return buf;
}
```

- In this example we show how a local array variable is allocated on stack. Notice that it grows Upwards within the stack.
- In the above code snippet attacker can supply shell code in buf.
- Attacker Input -> Input Username:
  - `<nops><shell code(256 Bytes)><RET address(4 Bytes)>`
  - We add “nops” as we don’t know the exact address of beginning of buf.
  - Possibly attacker can overwrite the RET address with the address of code he wants to execute.

- **Defenses:**
  - NX Bit: Non Executable bit is set for stack pages.
    - Virtual Memory Pages can be labelled as:
      - Readable
      - Writable
      - Executable
    - Stack can’t be executed
    - So no point of injecting machine code on stack
      - So use code that is already there.
Return to libc Attack:

- Stack diagram of a vulnerable function
- When a new function is called first SP is decremented to allocate space for local variables and AR.
- Over write the return address with the address of System function in libc library.

```
int system(const char *cmd);
system("/bin/ls");
```

- **Shell Code:**
  - `<Garbage> <&system()> <System RET> <Ptr to cmd>`
- A proper (e.g. &exit ()) return address for system () is provided because it needs a clean exit when it returns. Otherwise it will fault.

- Modern Machines expects arguments in registers (64 bit arch).
- How can an attacker influence the registers?
- We can do it with the use of ‘Gadgets’.

Return Oriented Programming (ROP):

- Enables us to perform arbitrary computation in a system in presence of NX bit.
  - Loading registers
  - Computing Addresses
  - Inserting Code off the stack

- Finding Gadgets:
  - Gadgets ends in a ret instruction.
  - Ret instruction are preceded with code to do work
  - X86 has **variable length instruction set.**
    * We can ask to start at any point in the given instruction set.

  e.g.

  f7c707000000 test 0x0007 %edi

  0f9545c3 set nzb %eip

  So in the above example instead of starting the execution at f7 we start at c7 then c707000000f – will become move.

  95 – xchg, 45 – inc, c3 – ret

  So this is how even if compiler did not intend to enter code for ret, we can still find it for our attacks in machine code by using this technique. (Gadgets)

- Use of multiple Gadgets:
  - Pop %rdb
  - move %rda %rdb
  - Ret
  - ret
ROP Summary:

- Find gadgets in original binary code.
- Gadgets are some useful set of instructions with ret ins in them.
- String together gadgets to accomplish attack.

**There are enough gadgets in libc to do any computation.**

Attack:

- Use gadget to call `mmap()` to allocate some writable & executable memory.
- Write some instructions in to newly allocated memory.
- Jump to that memory location.