Last time: Buffer Overflow

TBD: Return to libc, ROP, ASLR

NX bit

- the stack can be read and writable. but not executed unless the NX bit is unset
- the data segments is read / write
- the program executable can be seen just as a shared library which happens to have a main() function as well.
- the stack would have an activation record. arguments, return address, (stack) frame pointer

Return to libc attack

- if there is NX bit set on the stack memory page tables, we cannot execute injected shell code
- if we cannot inject machine code, use the code that's already there in system libraries. Like system("/bin/sh");
- We need to: put our string in memory; To pass the parameter in to memory; pass pointer to the buffer; transfer control to system

eg: fun inside libc – system()

```
int system(const char * cmd);
```

```
system("/bin/sh");
```
So we merge the two figures and overwrite the ret address in the vulnerable program:

shellcode:
<garbage><&system><system’s ret address><ptr to cmd>"

---

**Issues in this attack:**
- In real, systems have different calling procedure.

Alternative?
- In x64 system, arguments are expected on registers (RDI), not on stack
- the stack pointer would keep pointing to the return address and the RDI would point to "/bin/sh"
- find a snippet of code somewhere in the executable that will move something off the stack and into the RDI and then do a return.
- look for something like

Gadget
```
Pop %rdi
ret
```

- point the ret address of vuln function to the gadget, then pop the address from the stack to the RDI
- the gadget returns to the &system which is also stored on the stack.
- gadget in this case assumed to take its parameters from the stack itself. In a way acts as an activation record to the gadget

Assumption: address space randomization is disabled.
in some architectures, unless a program uses system(), its not loaded into the address space
Return Oriented Programming

Why?
- enables us to perform arbitrary computation in a system with the NX bit set
- Example – loading registers; computing addresses; inserting code off the stack.

How?
- if we don’t have something like system(), we use code snippets like the gadget to construct our own version of system or execute exec.
- Finding gadgets
  - gadgets end in “ret”. Single byte instruction.
  - ret inst is preceded by the code to do useful work
  - x86 instructions are variable length. Sequence of inst can be interpreted to meaningful code.
  - Example
    - f7 c7 07 00 00 00 test 0x0007%edi
    - 0x95 45 c3 setnz 61%edx

    We can tell cpu to not jump to f7 but to c7 as its variable length instruction.
- Even an obscure pattern of commands may end up having bit patters that would look like a <code>+ret gadget
- so we load eip in such a way that the proc starts executing from the next byte of a large instruction and it turns out to be a different instruction or rather even a gadget
- we keep using the gadgets and returns to executing code and on

Address space (Layout) Randomization:
ASLR
- move stuff around as much as efficiently possible