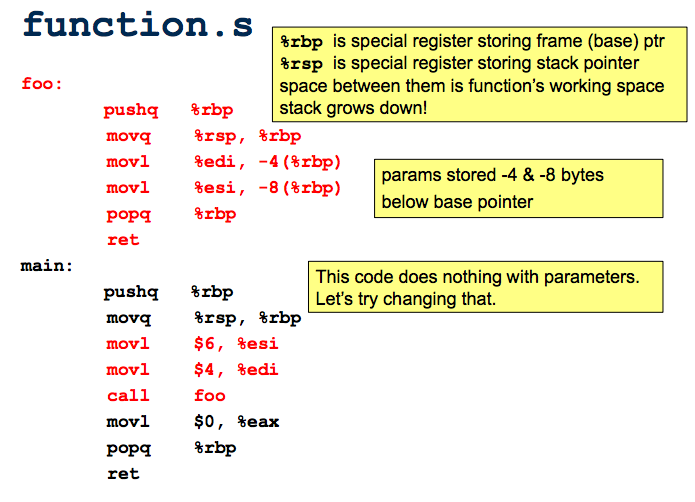
Stack – stores local variables, grows downwards

Heap – malloc, grows upwards; set aside for dynamic allocation / deallocation of blocks

Static

Code



store return val in %eax

x86-64 assembly code

|  |  |
| --- | --- |
| %rax | Result register; also used in imul & idiv |
| %rbx | Miscellaneous register (callee-save) |
| %rcx | 4th argument reg |
| %rdx | 3rd argument reg |
| %rsp | Stack pointer |
| %rbp | Frame pointer (base pointer) (callee-save) |
| %rsi | 2nd argument reg |
| %rdi | 1st argument reg |
| %r8 | 5th argument reg |
| %r9 | 6th argument reg |
| %r10 | Misc reg |
| %r11 | Misc reg |
| %r12-%r15 | Misc reg (callee-save) |
| %rip | Instruction pointer |

**Call** pushes address of next instruction (return address) onto stack & transfers control to operand address

**Leave** sets stack pointer (%rsp) to frame pointer (%rbp) (popped from stack)

**Ret** pops return addresses off stack & jumps to it

**Addressing memory**

|  |  |  |
| --- | --- | --- |
| Syntax | Address | Description |
| (reg) | reg | Base addressing |
| d(reg) | reg + d | Base addressing + displacement |
| d(reg, s) | (s x reg) + d | Scaled index + displacement (s = 2,4, or 8) |
| d(reg1, reg2, s) | reg1 + (s x reg2) + d | Base + scaled index + displacement (s = 2, 4, or 8) |

**Opcodes**: add, and, call, cmp, idiv, imul, jmp, lea (load effective address), mov, nop, or, pop, push, ret, sal, sar, shr (shift codes), sub, xor

**Jump opcodes:** JO (overflow), JNO (no overflow), JE/JZ (if equal - zero), JNE/JNZ (if not equal – not zero), JS (sign), JNS (not sign), JP/JPE (parity - parity even), JNP/JPO (not parity – parity odd)

**Jump opcodes (unsigned:** JB/JNAE/JC (below – carry, not above or equal), JNB/JAE/JNC (not below – not carry, above or equal), JBE/JNA (below/equal - not above), JA/JNBE (

**PROGRAMMING SECURITY**

General errors:

* look for unchecked buffer writes
* off-by-one errors (usually end of buffer indexing)
* look for inputs an attacker controls – what if no \n or \0
* malloc/calloc lengths
  + Is enough space allocated?
  + Include space for null-byte

*strlen(s)* calculates the length of the string s, not including the terminating ‘\0’ character.

*strcpy(dst, src)* copies the string pointed to by *src* to *dst*, including the terminating “\0” character

*sprintf* works exactly like *printf*, but instead writes to the string pointed to by the first argument – terminates the characters written with a “\0”

In C, can access an element before an array with a negative index

**Simple Side Channel Analysis**:

* use characteristics directly visible in single measurement trace
* key has simple, exploitable relationship with operations visible in trace
* typically, vulnerable implementations include key dependent branching

**Differential Side Channel Analysis**

* requires multiple traces, use statistical methods
* targets specific intermediate result in a specific part of the measurement traces
* choose selection function (intermediate result)
* selection function depends on known input/output data and a small # of hypotheses on key value

**Power Attacks**

* Differential Power Attacks (DPA)
* Dynamic current consumption of chip is correlated to gate activity

**Simple Power Analysis: (SPA)**

* directly interprets the power consumption of the device
* trace: a set of power consumptions across a cryptographic process
* 1 millisecond operation sampled at 5MHz yields a trace with 5000 pts

**STACK PROTECTION**

**NX Stack** (Nonexecutable Stack)

* Goal: if someone managed to inject malicious code into the stack, that code wouldn’t run because it would have the nonexecutable bit set
* **Weaknesses:**
  + If heap is not NX, you can put malicious code there
  + Use return-oriented programming: there is already-existing code
    - Disassemble in memory – use already-existing code and jump to them in weird orders rather than jumping to them sequentially

**Stack Canary**

* have canary value somewhere in code – once code is executed, if the canary code was not overwritten and it matches somewhere else where it was stored, then you know the code was not tampered
* **Weaknesses:**
  + 1. If you know where the stack canary is, just copy the value to make sure you do not change it
  + 2. Overwrite the second canary in memory and make sure they match

**ASLR** (Address Space Layout Randomization)

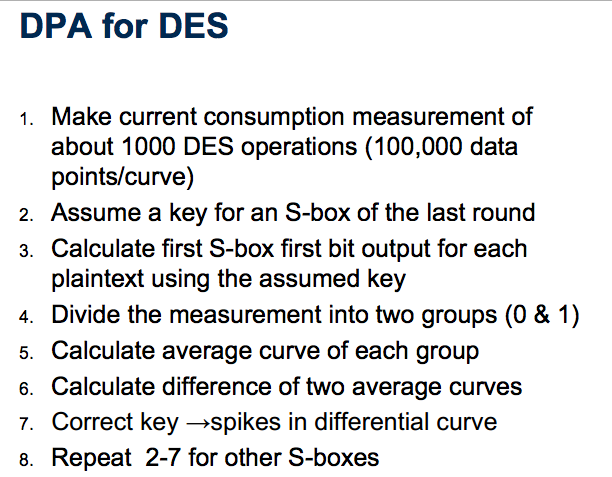
* start stack at random place in memory rather than a fixed point; idea that if someone is using hardcoded malicious code, ASLR can get around it
* **Weaknesses**:
  + Relative addressing: instead of return to a hardcode address, return a relative address to where you are on the stack right now
  + Know the difference between hard and relative addressing

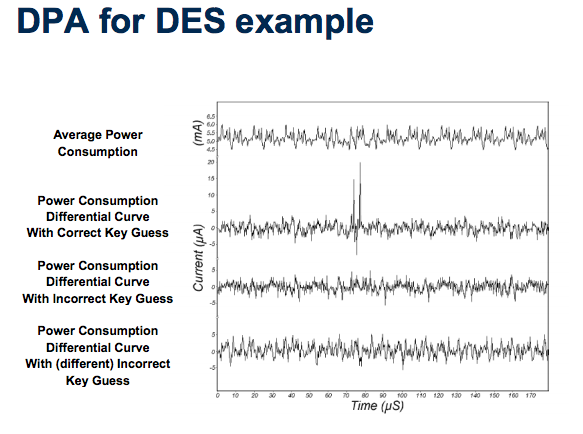
**No-op slide**

* found a buffer vulnerable to overflow, used shell code so that it runs
* if can’t find shell code address, fill with no-ops and jump to no-op slide to get to shell code

**EVALUATING STACK DEFENSES ON HW7**

* **NX stack on hw7**: it would fail because all of our solutions involve injecting malicious code
* **Stack canary**: depends on where the stack canary is located. If the canary is located before the overflowed buffer, then it would not fail. However, if it was after, then it would fail.
* **ASLR**: it would fail since the return address of the shell code would be different each time, and would jump to a fixed address



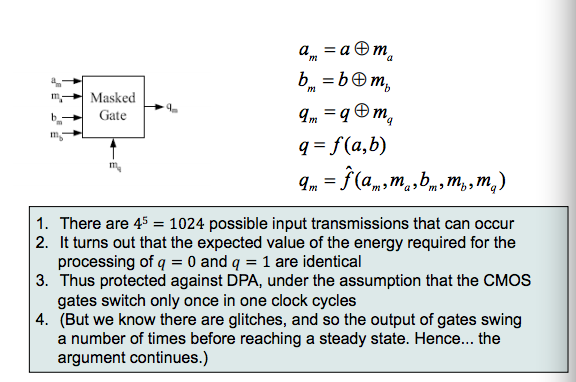


**Countering DPA**

* 1st approach: make power consumption of device independent of the data processed; detached power supplies ; logic styles with a data independent power consumption ; noise generators ; insertion of random delays – methods are costly
* 2nd approach: randomize intermediate results; idea is that power consumption of randomized data uncorrelated to actual intermediate results; involves masking (applied at algorithm or gate level)

**Gate Level Masking**

* no wires store values correlated to intermediate algorithm result
* converting unmasked digital circuit to a masked version can be automated
* **normal gates are susceptible to DPA** because attacker splits traces into two groups when q=0 and when q=1 – since E(q=0) =/= E(q=1), DPA attack possible



Masking is not perfect – xor gates can leak info about unmasked values since they do not change output when both the inputs change value simultaneously or within a small time – masked circuits may still be vulnerable to DPA because of signal delay