What have we learned so far about SW security?

- Attacks: buffer overflow, integer overflow, timing issues

How might you protect against some of the buffer overflow (and related) vulnerabilities that we have discussed in class?

- Set the stack region of memory to “non-execute”, so the adversary cannot point the instruction pointer in the stack. (DEP - data execution prevention)
  - If the stack is non-executable, can we still get a program to execute arbitrary code of an attacker’s choosing?
    - There is code compiled into the program (from the text region), so things like “exec” and “strcpy” calls will be compiled from the text region.
    - "Return-to-libc attack": change the return addr so that it points to somewhere in libc.
    - More advanced: return-oriented programming.

- High-level point: DEP is one level of defense, but there are attacks that circumvent it.

- Another defense: ASLR (Addr space layout randomization) - locations of things in the stack are randomized.
  - Not super effective for 32 bit architectures, because the adversary can brute force. Much better with 64 bit.
  - If the stack is executable, we can use a bunch of NOP sleds spread throughout the stack to increase the probability that a random return addr hits one of them.

- These defenses are more common on desktops/laptops, but LESS common on embedded systems.
  - Alternatively: products by big companies that have been attacked already are more likely to implement defenses. But companies that haven’t received adversarial pressure (e.g., car manufacturers, home automation systems) do not widely deploy defenses (yet).

- **Stackguard**: push params, return addr, saved FP, canary, then buffer.
  - Canary: ~4-8 bytes, 1st byte is a 0. pushed onto stack before buffer.
  - The program has logic to check that the canary’s value stays the same as its original value. So if the adversary overwrites the canary with a different value, the program exits.
  - Therefore, a strcpy needs to copy the original value of the canary (including 0 byte) for the program to keep running. However, strcpy will only copy up until a 0 byte. So if strcpy hits the 0 byte from the canary, it will stop (and not overwrite the ret addr or the SFP).
  - A memcpy will copy up to n bytes. So the adversary can try to overwrite the canary, IP, and FP with memcpy.
  - Eg: a function “foo” will set the canary, call strcpy, and then when foo returns it will check its canary.
Defense in Depth

● Multiple layers of defense all serve to increase the barrier to attack in different ways.

defenses?

constant time functions for code that deals with privacy sensitive information (e.g., crypto)

use safe functions (e.g., not strcpy)

use type safe language

fuzzing.

● Generate random inputs to test how a program behaves. If it crashes, there’s a bug.

● this page gives some history: https://en.wikipedia.org/wiki/Fuzz_testing

static analysis tools to see if any part of a program is potentially vulnerable.

non-execute

Following quoted from some webpage:

Data Execution Prevention (DEP) is a security feature included in modern operating systems. It is known to be available in Linux, Mac OS X, iOS, Microsoft Windows and Android operating systems and is intended to prevent an application or service from executing code from a non-executable memory region. This helps prevent certain exploits that store code via a buffer overflow, for example.[1] DEP runs in two modes: hardware-enforced DEP for CPUs that can mark memory pages as nonexecutable, and software-enforced DEP with limited protection for CPUs that do not have hardware support. Software-enforced DEP does not protect against execution of code in data pages, but instead counters SEH overwrite, another type of attack.

Hardware:
The NX bit, which stands for Never eXecute, is a technology used in CPUs to segregate areas of memory for use by either storage of processor instructions (or code) or for storage of data, a feature normally only found in Harvard architecture processors. However, the NX bit is being increasingly used in conventional von Neumann architecture processors, for security reasons.

Intel markets the feature as the XD bit, for eXecute Disable. AMD uses the name Enhanced Virus Protection. The ARM architecture refers to the feature as XN for eXecute Never; it was introduced in ARM v6.[1]

address-space layout randomization

static analysis

ASLR

Address space layout randomization (ASLR) makes this type of attack extremely unlikely to succeed on 64-bit machines as the memory locations of functions are random. For 32-bit systems ASLR provides little benefit since there are only 16 bits available for randomization, and they can be defeated by brute force in a matter of minutes.[1]

General principles:

● Trusted vs. trust worthy: A component may be trusted, but still have bugs in it. We want to know: are trusted components trust worthy? (E.g., do they provide the safeguards we assume they do?)
● Open vs closed source
● Check user inputs - user may be attacker.
● Check all return values
● Least privilege: each individual component of a system should run with the least amount of information necessary to do its job.
   ○ If marketing web page, doesn’t need access to banking and credit card info. If adversary breaks into marketing web page, shouldn’t be able to compromise integrity of accounts, transfer funds, and so on.
• Car radio may not need access to brakes.

• Securely clear memory
• Failsafe defaults
  ○ Flashing red lights for traffic intersections.

  If a firewall fails, it should deny access --
  not let any or all traffic in.

  Log files. If log file full -- one option
  would be to not log anymore (not safe).
  Another option would be to not allow access
  until space is cleared (safe).
  • Attack: fill up log files with tons of errors, if the system just stops logging,
    then the attacker can now mount attack undetected.

• Defense in Depth
• Reduce size of TCB (trusted computing base): what part of a system needs to be secure
  for the system to work properly? Make this part as small as possible.