CISC 327: Software Quality Assurance

Lecture 19–2 (29a in 2017)

More Insecurity
Heartbleed in context

• In 327 we have learned about process models, including XP, and inspections

• The OpenSSL project was not a real-life Bogosys but had many problems:
  – understaffed
  – underfunded
  – inadequate inspections
  – excessive generality
Heartbleed in context

• Excessive generality:
  – Being portable is good, especially for security-critical code: you want one code base that you can test thoroughly, and inspect fully (leaving aside that the developers didn’t seem to do inspections)
  – However, for performance and portability on “weird” platforms, the OpenSSL code was full of conditional compilation (C’s #if, #ifdef)
    • makes the code hard to follow, and hard to test—large projects often have a dedicated server or two for (e.g.) regression testing, but with #ifs based on OS or machine architectures, you need one server for each OS/architecture combination
Heartbleed in context

• Excessive generality:
  – After Heartbleed, the OpenBSD developers yelled at the OpenSSL developers for this, and immediately forked OpenSSL → “LibreSSL”
    • ripped out many of the #ifs, because OpenBSD developers only really care about OpenBSD, a Unix-like system, and not about older Windows versions and other “weird” systems
  – OpenSSL developers were also criticized for seeming to worry more about performance than security
  – A question of priorities
    • portability is good
    • performance is good
    • security is good
Linux

• In 2017, Linus Torvalds wrote on the Linux kernel mailing list:

As a security person, you need to repeat this mantra:

"security problems are just bugs"

and you need to _internalize_ it, instead of scoff at it.
Language-Based Security

- What can we do to prevent buffer overruns, generally?
Language-Based Security

• C allows arbitrary memory access
  – Saying `long buf[128];` tells C to allocate 128 longs’ worth of space, but C does not prevent reading (and writing) before `buf[0]` or after `buf[127]`
  – C also allows arbitrary casts between integers and pointers, and between different pointer types, so any sequence of bits can be treated as a pointer—including a function pointer
Language-Based Security

• Java and Python (for example) do not allow such operations
  – What, precisely, is not being allowed?
  – Or: what, precisely, is allowed in Java and Python?
Memory Safety

• **Memory safety** says, informally:
  If a program thinks it has a pointer $P$ to type $T$, reading the data stored at address $P$ will result in a value $V$ of type $T$.

• To get memory safety, you need **type safety**:
  If a program has a variable $X$ of declared type $T$, then for all possible program executions, the value of $X$ has type $T$. 
Memory Safety

• Traditionally, memory safety and type safety require **bounds checking**
  – otherwise, you can write arbitrary bits to arbitrary (for a buffer overrun attack, carefully chosen) memory locations

• Traditionally, they also require **automatic memory management**
  – in C, you allocate and deallocate (free) blocks of memory yourself
  – you can deallocate a block, let another part of the program allocate a block (possibly at the same address), and then read from the block—if the types of the allocated structures aren’t the same, you have violated memory safety
Memory Safety

• Back in the 1990s, Java was controversial because it had automatic memory management (only used by academics and other weird people who used languages like Lisp)

• But memory safety was seen as essential for Java’s original purpose: web applets
Overcoming Tradition

- **Bounds checking** is problematic for arrays of non-constant length
  - the argument to fgets() may not be known at compile-time
  - leads to a run-time check

- **Refinement type systems**, developed in the late ’90s and 2000s, allow most run-time checks to be omitted—but the language is still memory-safe!
  - example: Liquid Haskell
Overcoming Tradition

• **Automatic memory management** is problematic for low-level code, such as OS kernels
  – Who watches the watchmen?
  – Who loads the bootloader?
  – Who allocates the data structures for the memory allocator?

• Since some programs **must** manage memory manually, we have to give up memory safety and write them in C! Right?
Overcoming Tradition

• **Alternative:** Manual memory management only where needed *and* checked *statically* by the compiler
  
  – Mozilla’s Rust language
  
  – As of November 2017, Firefox is now partly written in Rust

  – I believe, however, that at the moment this is not for security reasons but for performance on multicore CPUs
Conclusion

• In general, the relationship between choice of implementation language and software quality is complicated, and hard to study
  – some companies make a point of hiring programmers with Haskell experience, even if they don’t use Haskell at the company
  – but extensive knowledge of Haskell is correlated with attending certain “fancy” universities
  – maybe, on average, people who go to fancy universities are better programmers
  – but is it because they learned Haskell?
Conclusion

• One of the few statements we can make is:

Programs written in memory-safe languages do not cause buffer overrun vulnerabilities.

• Note the word “cause”: a program in a memory-safe language that calls a library/OS function written in C could still have a buffer overrun! But then it’s C’s fault.