CS 6301-004: Language-based Security

Dr. Kevin Hamlen
Fall 2018

Prerequisites: none*

*But if you’ve ever programmed in a functional language (ML, Haskell, Lisp, OCaml, etc.) then that will be a helpful skill. Also, if you know assembly language, that will be quite useful too.
Outline

• Course logistics
  – course objectives
  – homework grading, etc.
  – about me

• What is “Language-based Security”?

• Tentative course schedule (list of topics)

• Demo: Program-proof co-development
Course Information

• Course webpage:
  – http://www.utdallas.edu/~hamlen/cs6301fa18.html
  – google “hamlen”, click top link, scroll to bottom

• Instructor:
  – Dr. Kevin Hamlen
  – ECSS 3.704
  – Office hours: After class (MW 2:15-3:15)
Course Objectives

• Cutting-edge research
  – Learn how to extract (the important) info from security-related research articles
  – Learn about modern efforts toward a science of computer security
  – Learn basics of programming language theory, functional programming, automated theorem-proving, etc.
  – Get your hands dirty: Implement and formally verify something

• Warning: This is a research-level class!
  – Many problems/questions are open-ended. We will be exploring the known issues together.
  – Not only is the software extremely beta, the whole concept behind the software is extremely beta!
Grading

• Homework (30%)
  – programming exercises – learn to program in Coq
  – **first one (“Basics”) due next Wednesday 8/29**
  – see online schedule for the other six due dates
  – Recommendation: Complete them far in advance! Then you’ll be done!
  – If you have trouble, do some exercises in the online text (Pierce et al.)

• Quizzes (30%)
  – start of most class sessions (see schedule) (~15 min.)
  – covers assigned reading for the day
  – first one next Wednesday (8/29)

• Class participation (10%)
  – discuss article, ask questions

• Projects (30%)
  – formally validate a native code algorithm in Coq
  – project proposals due around mid-semester (tentatively 10/29)
  – implement during last 6 weeks of course

• No exams
Quizzes

• Approximately 8 questions each
  – multiple-choice / short answer

• Difficulty level
  – multiple-choice != obvious-choice
  – main concepts (e.g., “What is this paper (really) about?”)
  – feasibility critique: main limitations, pros/cons
  – a few harder in-depth questions to test whether you caught subtle but essential details

• Warning: These articles are HARD to understand!
  – contain many tiny technical details
  – I don’t test on minutiae. Don’t memorize everything.
  – But hard questions might focus on a seemingly minor item that you didn’t realize is very significant.
Comprehending Papers

• Ability to read and digest research articles (at a reasonable pace) is a learned and very valuable skill.
  – articles are extremely dense!
  – most assume background knowledge that you lack
  – I expect you to look up terms you don’t understand on your own initiative.
  – I don’t expect you to understand everything, even after doing your best to look things up.

• After reading, be sure you can answer the following:
  – What’s the MAIN discovery?
  – Why is this better/worse than alternatives?
  – What are the system’s weaknesses? How can I break it?
  – Do you understand the main definitions / notations?
About Me

• originally from the northeastern US (Buffalo, NY)
• Undergrad
  – Carnegie Mellon (computer science and math)
  – Senior thesis: Proof-Carrying Code
• Masters (’02) & Ph.D. (’06)
  – Cornell (computer science)
  – Dissertation: certifying in-lined reference monitors
• Government experience
  – Principal Investigator for over 20 US Federal cyber-security contracts with Navy, Air Force, Army, NSF, NSA, and DARPA
• Industry experience
  – Microsoft Research (Redmond & Cambridge)
  – language-based security for .NET and F#
• Personal
  – married, 5-year-old son + 3-year-old twin boys
  – Christian
What is LBS?

• Leveraging theory of programming language design and compiler construction to enforce software security

• Two domains of research:
  – new languages/tools for creating secure software from scratch
  – securing legacy code (e.g., written in C)

• Three stages of enforcement
  – static (find & fix vulnerabilities before runtime)
  – dynamic (detect and block attacks at runtime)
  – audit (recover and assign blame after an attack)
Grand Challenge: Secure Program Development

• Is it possible to develop secure software that is guaranteed to be vulnerability-free?

• Scenario: You are hired to write the control software for a nuclear reactor.
  – it must NEVER fail (millions of lives at stake)
  – it must cope with adversarial conditions (prime target)
  – it must be efficient (too slow = meltdown)

• Traditional approaches
  – test a lot ("It didn’t crash today...")
  – write a proof (consisting of about 10K pages of math)
  • How do we know there isn’t a bug in the proof??
Grand Challenge:
Securing Legacy Code

• Scenario: NSA wants secure software on their office workstations.
  – need web browsers, document readers, etc.
  – need internet connectivity
  – stores and/or reads top secret documents
  – not feasible to rebuild the entire universe of software from the ground up
  – software is proprietary (and usually closed-source)

• How to stop secrets from leaking?
Grand Challenge: A Science of Security

- Can we develop a science of security like we have for math or physics?
  - Are there iron-clad “proofs” of security?
  - What does it even mean for a system to be “secure”?
  - Are there metrics for security? Can we determine that one software system is “more secure” than other? Can we prove that it’s “80% secure”?
  - Are there some security policies that are provably unenforceable? Can we prove that certain enforcement mechanisms can enforce certain classes of policies and not others?
Tentative List of Topics

• First 4 weeks:
  – Developing machine-verified software with Coq
  – basis for homework and projects
• Next 2 weeks: LBS foundations
• After that, move into cutting-edge research:
  – Software Model-checking
  – Software Fault Isolation
  – Code-injection and code-reuse attacks & defenses
  – Artificial Software Diversity and Obfuscation
  – Cyber offense (“active defense”)
  – Information flow controls (confidentiality enforcement)
  – Web scripting security
  – In-lined Reference Monitoring
  – Cyber-deceptive Software Engineering
A Tale of Security Woes:

THREE VULNERABILITY STORIES
Tale #1: Linux GHOST

- Bug in the Linux glibc library
- Discovered by Qualys researchers during a routine code audit in 2015
- Affects all code that uses glibc for host-lookups (i.e., nearly all Linux networking software) between 2000-2013
- Can you spot the bug?

```c
int __nss_hostname_digits_dots( ...) {
    ...
    size_needed = sizeof(*host_addr) + sizeof(*h_addr_ptrs) + strlen(name) + 1;
    *buffer = (char*) malloc(size_needed);

    ... 35 lines of code ...

    host_addr = (host_addr_t*) *buffer;
    h_addr_ptrs = (host_addr_list_t*) ((char*) host_addr + sizeof(*host_addr));
    h_alias_ptr = (char**) ((char*) h_addr_ptrs + sizeof(*h_addr_ptrs));
    hostname = (char*) h_alias_ptr + sizeof(*h_alias_ptr);
    ...
```
Tale #1: Linux GHOST

- Bug in the Linux glibc library
- Discovered by Qualys researchers during a routine code audit in 2015
- Affects all code that uses glibc for host-lookups (i.e., nearly all Linux networking software) between 2000-2013
- Can you spot the bug?

```c
int __nss_hostname_digits_dots( ... ) {
 ...

  size_needed = sizeof(*host_addr) + sizeof(*h_addr_ptrs) + strlen(name) + 1;
  *buffer = (char*) malloc(size_needed);

  ... 35 lines of code ...

  host_addr = (host_addr_t*) *buffer;
  h_addr_ptrs = (host_addr_list_t*)((char*) host_addr + sizeof(*host_addr));
  h_alias_ptr = (char**) ((char*) h_addr_ptrs + sizeof(*h_addr_ptrs));
  hostname = (char*) h_alias_ptr + sizeof(*h_alias_ptr);

  ...
```
Is it really that big a deal?

• Qualys was able to take complete remote control of affected Linux machines merely by sending them a maliciously crafted email (unread!).
• Can you figure out how they did it?
Is it really that big a deal?

... if (isdigit(name[0])) {
  for (cp=name;; ++cp) {
    if (*cp == '\0') {
      if (*--cp == '.') break;
      if ((af == AF_INET) ? inet_aton(name, host_addr) : inet_pton(af, name, host_addr))
        result_buf->h_name = strcpy(hostname, name);
      goto done;
    }
  }
  if (!isdigit(*cp) && *cp != '.') break;
}... 

- Qualys was able to take complete remote control of affected Linux machines merely by sending them a maliciously crafted email (unread!).
- Can you figure out how they did it?
Tale #2: Heartbleed

• Bug in the OpenSSL (secure web communications!) library discovered by Codenomicon in 2014
• Buffer over-read error in implementation of Heartbeat TLS protocol:
  – read-loop trusts length bound provided by user
  – over-read data sent directly back to attacker
• Vulnerability exposed ~66% of the internet to theft of encryption keys between 2011-2014.
• Still highly exploitable because OpenSSL is so pervasive, cannot always be patched in the wild.
• Heartbeat packets deemed so innocuous, they were not even logged during the zero-day window.
Tale #3: Shellshock

- Bug (undocumented feature?) discovered in Linux bash shell (by IT manager Stephane Chazelas in his spare time!) in 2014
- Bash command-parser interprets certain text in environment variables as code and executes it during parsing(?!)
- Impact: All Linux software storing user-provided data in environment variables susceptible to complete remote compromise.
- Zero-day window: 25 years(!!) (198?-2014)
Is secure code development even possible?

• Open-source failed in all these instances.
  – questionable whether open-source model actually provides greater security
• Unit testing didn’t work in these cases either.
  – input space is just too large to cover with tests
• What about better programming languages?
  – But Shellshock was a misguided design choice.
• What’s the answer?
Coq: Programming with Proofs

• Coq
  – stands for “Calculus of Constructions” (the underlying type theory of the system)
    • named after mathematician Thierry Coquand
  – developed by INRIA, France over last decade
  – most powerful secure software development system to date (in my opinion)

• Specification language based on ML/OCaml
  – all loops are recursive (no while/for loops)
  – immutable variables (variables are assign-once!)
  – first-class functions
  – parametrically polymorphic
  – higher-order, dependent type system (!)

• Demo
Homework

• Download and install Coq
  – see links to Coq page from course web page
  – use version 8.8

• Read for next time:
  – “Preface” of the Software Foundations online text (see course web page).
  – Read the “Basics” chapter up to first exercise
  – Solve first two exercises (nandb, andb3)