Logistics

• Next week, my OH will be 10-11 vs 11-12

• HW6 (= writing + Monad.hs) counts!
  ➤ You HW grade = max of 5 HWs

• CAPEs!
Special-topic: constant-time programming
Today + Week 10

- What is constant-time programming?
- Constant-time programming in C
- Constant-time programming in FaCT
What’s wrong with this picture?

Log in as Nancy Smith
alice@aol.com · Not You?

alice@aol.com

Password

Log In

Recover Your Account

Sign up for Facebook

The password you’ve entered is incorrect. Forgot Password?
What’s wrong with this picture?

It’s leaking that alice@aol.com is a valid user on FB!
Fix this by hiding users!

```javascript
function login(req) {
  if (!isValid(req.user)) {
    // reply “Invalid user or password”
  } else {
    const pHash = findPass(req.user);
    if (pHash !== hash(req.password)) {
      // reply “Invalid user or password”
    } else {
      // succesfull login!
    }
  }
}
```
function login(req) {
    if (!isValid(req.user)) {
        // reply “Invalid user or password”
    } else {
        const pHash = findPass(req.user);
        if (pHash !== hash(req.password)) {
            // reply “Invalid user or password”
        } else {
            // succesfull login!
            // succesfull login!
        }
    }
}

Is this secure? A: yes, B: no
function login(req) {
    if (!isValid(req.user)) {
        // reply “Invalid user or password”
    } else {
        const pHash = findPass(req.user);
        if (pHash !== hash(req.password)) {
            // reply “Invalid user or password”
        } else {
            // succesfull login!
        }
    }
}

Need to always do the password hashing!
Hashing takes take

• What is a cryptographic hash function?
  ➤ **Pre-image resistant:** Given output, cannot find input such that output = hash(input)
  ➤ **Second pre-image resistant:** Given input₁, cannot find input₂ such that hash(input₁) = hash(input₂)
  ➤ **Collision resistant:** Cannot find input₁, input₂ such that hash(input₁) = hash(input₂)

• How are hash functions implemented?
  ➤ Lots of bit mixing
E.g., SHA2, 60-80x:
Don’t use hash functions for passwords

• In practice you should NOT hash passwords

• You should use algorithms like bcrypt and scrypt
  ➤ Internally use hash functions
  ➤ Designed to be resistant to brute-force attacks that try to guess passwords even with Moore’s law
function login(req) {
    const uHash = hash(req.password);
    if (!isValid(req.user)) {
        // reply “Invalid user or password”
    } else {
        const pHash = findPass(req.user);
        if (pHash !== uHash) {
            // reply “Invalid user or password”
        } else {
            // succesfull login!
    }
}
function login(req) {
    const uHash = hash(req.password);
    if (!isValid(req.user)) {
        // reply “Invalid user or password”
    } else {
        const pHash = findPass(req.user);
        if (pHash !== uHash) {
            // reply “Invalid user or password”
        } else {
            // succesfull login!
        }
    }
}

Are we done? A: yes, B: no
function login(req) {
  const user = findUser(req.user);
  const uHash = hash(req.password);

  if (!user || uHash !== user.pHash) {
    // reply “Invalid user or password”
  } else {
    // succesfull login!
  }
}
Always run isValid & findPass

```javascript
function login(req) {
    const user = findUser(req.user);
    const uHash = hash(req.password);

    if (!user || uHash !== user.pHash) {
        // reply "Invalid user or password"
    } else {
        // succesfull login!
    }
}
```

Finally, are we done? A: yes, B: no
Short circuit operators

• How do we evaluate \( \text{exp}_1 \lor \text{exp}_2 \) ?

  ➤ \( \text{exp}_1 \rightarrow \ldots \rightarrow \text{value} \)
  ➤ if \( \text{value}_1 = \text{true} \)
      then \text{true} else \( \text{exp}_2 \rightarrow \ldots \rightarrow \text{value}_2 \)

• How do we evaluate \( \text{exp}_1 \land \text{exp}_2 \) ?

  ➤ \( \text{exp}_1 \rightarrow \ldots \rightarrow \text{value} \)
  ➤ if \( \text{value}_1 = \text{false} \)
      then \text{false} else \( \text{exp}_2 \rightarrow \ldots \rightarrow \text{value}_2 \)
Don’t use short-circuiting operators

```javascript
function login(req) {
    const user = findUser(req.user);
    const uHash = hash(req.password);

    if (i2b(b2i(!user) | b2i(uHash !== user.pHash))) {
        // reply “Invalid user or password”
    } else {
        // succesfull login!
    }
}
```

\textbf{b2i} :: \textbf{Bool} $\rightarrow$ \textbf{Int}
\textbf{i2b} :: \textbf{Int} $\rightarrow$ \textbf{Bool}
function login(req) {
    const user = findUser(req.user);
    const uHash = hash(req.password);

    if (i2b(b2i(!user) | b2i(uHash !== user.pHash))) {
        // reply "Invalid user or password"
    } else {
        // succesfull login!
    }
}

OMG, now are we done? A: yes, B: no
uHash !== user.pHash

May be leaking information about the stored password!
String comparison is dangerous

- What is !== doing internally when comparing strings?
  - Probably using C’s strcmp

- How does strcmp work?

```c
strcmp( "" , "" );
```
Don't use strcmp

```javascript
function login(req) {
  const user = findUser(req.user);
  const uHash = hash(req.password);

  if (i2b(b2i(!user) | b2i(cmp(uHash, user.pHash)))) {
    // reply "Invalid user or password"
  } else {
    // succesfull login!
  }
}
```

`cmp :: String -> String -> Bool`

cmp doesn't terminate early, it loops to end of list
This is a lot of work..

do we really need to do this?
• Implementing RSA algorithm by the book is unsafe
  ➤ Attacker can learn secret keys via timing:

  \[
  t_0 \leftarrow \text{getTime} \\
  \text{rsa.decrypt(...)} \\
  t_1 \leftarrow \text{getTime}
  \]
Crypto at the heart of SSL/TLS

- Secure Socket Layer/Transport Layer Security
  - Protocol used to provide secure pipe between networked computers
  - What does secure mean here?
• Can extract secret keys across the network
  ➤ Is this still relevant today?

• Can extract secret keys across process
  ➤ Where is this relevant today?

• Can extract secret keys across VMs
  ➤ Where is this relevant today?
How can we avoid leaks?

• Randomization, randomization, randomization!
  ➤ Every time server responds, delay by some random amount
  ➤ Every time you do operation on integer, mask it with some random integer

• Challenges with this?
Constant-time

• Implement algorithms to run in constant-time

  ➤ l.e., make sure that $t_1 - t_0 = c$

  $t_0 \leftarrow \text{getTime}$
  $\text{rsa.decrypt}(\ldots)$
  $t_1 \leftarrow \text{getTime}$

  ➤ Does this mean we can’t use operators like $\&\&$ or $\|\|$?

  ➤ A: yes, B: no
Constant-time (in secrets)

• Only parts of computations that deal with secrets need to run in constant-time
  ➤ Don’t care about public parts!
How do we write constant-time code?

• We’re going to look at two ways:
  ➤ In a general-purpose language, C
  ➤ In a domain-specific language, FaCT