The Plan

• Administrivia
• Project specification updates
• Networking
• Scanning and Parsing
• Project submission (and git) demo
• In-class build time! (maybe)
Administrivia

• Each team should all have…
  • …access to umdcmsc388n.slack.com with a channel per team
  • …a team repo on gitlab.cs.umd.edu
• Daily status reports start today: ter.ps/388Nreport
Project Specification Updates
Problem Spec Update

- Passwords
- Access control
- Assume network is secure
Passwords

as principal admin password "admin" do
  create principal bob "B0BPWxxd"
  change password admin "BetterPassword"
  set rule too_hot if temperature >= 80 then set air_conditioning = 2
  activate rule too_hot
  set delegation air_conditioning admin read -> bob
return temperature.0

***
Access Control - Grammar

<prim_cmd> ::=  
  create principal p s  
  | change password p s  
  | set x = <expr>  
  | local set x = <expr>  
  | if <cond> then <prim_cmd>  
  | set delegation <tgt> q <right> -&gt; p  
  | delete delegation <tgt> q <right> -&gt; p  
  | default delegator p  
  | print <expr>  
  | set rule x = if <cond> then <prim_cmd>  
  | activate rule x  
  | deactivate rule x

<tgt> ::= all | x

<right> ::= read | write | delegate | toggle
Access Control - Language Description

`set x = <expr>`

Sets x’s value to the result of evaluating `<expr>`, where x is a variable. If x does not exist this command creates it. If x is created by this command, and the current principal is not admin, then the current principal is delegated `read`, `write`, and `delegate` rights from the `admin` on x (equivalent to executing `set delegation x admin read -> p` and `set delegation x admin write -> p`, etc. where p is the current principal).

**Failure conditions:**

- Fails or exhibits security violation if evaluating `<expr>` does
- Fails if x is already set to a rule
- Security violation x exists and the current principal does not have `write` permission on x.
  (DENIED_WRITE)

**Successful status code:** SET
Access Control - Enforcement

1. Admin has <right> on x (for all rights <right> on variables $x^*$)

2. A principal $p$ has <right> on x if principal anyone has <right> on x.

3. A principal $p$ has <right> on x if there exists some $q$ that has <right> on x and $S_d$ includes a delegation assertion $q \ x \ <right> \ \rightarrow \ p$. 
Networking
(Abbreviated)
What is the Internet?

A collection of independently operated autonomous systems (ASes)

Src=Alice, SrcPort=1234, Dest=Bob, DestPort=80, Content=“Hello world”
Network Programming
(aka Sockets)

- The operating system provides an interface for sending/receiving network packets
- A socket is a descriptor for network communication
- As a client, you connect your socket to a remote host, and read/write to that socket as you would a file
- As a server, you listen and accept incoming connections, and read/write to that socket as you would a file
- read()/recv() is a blocking operation; to wait for input from multiple sources, use select
```python
import socket, select

listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listen_socket.bind(('', 9998))
listen_socket.listen(10)
client_sockets = []  # an empty list

while True:
    read_list = [listen_socket] + client_sockets
    (ready_read, _, _) = select.select(read_list, [], []).

    for sock in ready_read:
        if sock is listen_socket:
            new_conn, addr = sock.accept()  # accept the connection
            client_sockets.append(new_conn)
        else:
            data = sock.recv(1024)  # read up to 1K of data
            if data != "":  # the connection is open
                sock.send("Go away.\n")  # I'm not very nice
            else:
                client_sockets.remove(sock)
                sock.close()
```

Select example
What can go wrong?

Man-in-the-Middle can…

• …listen in
• …change data
• …replay
What should we do?

Bob

Man-in-the-Middle can...
• ...listen in
• ...change data
• ...replay

bob balance: 650
Encryption and Decryption

\[ C = E(M) \]
\[ M = D(C) \]

i.e.,
\[ M = D(E(M)) \]

where

- \( M \) = plaintext
- \( C \) = ciphertext
- \( E(x) \) = encryption function
- \( D(y) \) = decryption function
Symmetric and Asymmetric Crypto

- **Symmetric crypto**: (also called private key crypto)
  - Alice and Bob share the same key (K=K1=K2)
  - K used for both encrypting and decrypting
  - Doesn't imply that encrypting and decrypting are the same algorithm
  - Also called private key or secret key cryptography, since knowledge of the key reveals the plaintext

- **Asymmetric crypto**: (also called public key crypto)
  - Alice and Bob have different keys
  - Alice encrypts with K1 and Bob decrypts with K2
  - Also called public key cryptography, since Alice and Bob can publicly post their public keys

### Approved
- AES, Triple DES
- RSA, ECDSA
What should we do?

Man-in-the-Middle can...

- ...listen in
- ...change data
- ...replay

Message Authentication Code

RSA Digital Signature
Message Authentication Codes (MACs)

- MACs provide message **integrity** and **authenticity**
- $\text{MAC}_K(M)$ – use symmetric encryption to produce short sequence of bits that depends on both the message ($M$) and the key ($K$)
- MACs should be resistant to **existential forgery**: Eve should not be able to produce a valid MAC for a message $M'$ without knowing $K$
- To provide confidentiality, authenticity, and integrity of a message, Alice sends
  - $E_K(M, \text{MAC}_K(M))$ where $E_K(X)$ is the encryption of $X$ using key $K$;
  - or
  - $E_K(M), \text{MAC}_K(E_K(M))$

Proves that $M$ was encrypted (confidentiality) by someone who knew $K$ (authenticity) and hasn’t been changed (integrity)
Without knowing $k_1$, Eve can't read Alice's message.

Without knowing $k_2$, Eve can't compute a valid MAC for her forged message!
Asymmetric Crypto

Alice

\[ K_1 = B_{\text{public}} \]
\[ S_1 = A_{\text{private}} \]

Bob

\[ K_2 = B_{\text{private}} \]
\[ S_2 = A_{\text{public}} \]

RSA, ECDSA

Approved
What should we do?

Bob

Man-in-the Middle can…

• …listen in
• …change data
• …replay

Nonces!

ATM

Internet

deposit $50
balance is $1050
withdraw $400
balance is $650

bob balance: 650

BANK

$
Nonces

\[ \text{Src} = \text{Alice}, \text{Dest} = \text{Bob} \]
\[ \text{Msg} = E_{k_1}\{\text{“network security is fun”, Nonce}\}, \]
\[ MAC_{k_2}(E_{k_1}\{\text{“network security is fun”, Nonce}\}) \]
What should we do?

Bob

ATM

Internet

BANK

$50

balance is $1050

withdraw $400

balance is $650

bob balance: 650

Man-in-the Middle can…

• …listen in
• …change data
• …replay

TLS and PKI
Scanning and Parsing

(Abbreviated)
Scanner and Parser

- **Scanner / lexer / tokenizer** converts program source into **tokens** (keywords, variable names, operators, numbers, etc.) with regular expressions.
- **Parser** converts tokens into an **AST** (abstract syntax tree) based on a context free grammar (CFG).

Source → Scanner → Token Stream → Parser → AST
Scanning (“tokenizing”)

Converts textual input into a stream of **tokens**
- These are the **terminals** in the parser’s CFG
- Example tokens are **keywords**, **identifiers**, **numbers**, **punctuation**, etc.

Tokens determined with regular expressions
- Identifiers match regexp `[a-zA-Z_]\[a-zA-Z0-9_]\*`
- Non-negative integers match `[0-9]+`
- Etc.

Scanner typically ignores/eliminates whitespace
Implementing Parsers

Many efficient techniques for parsing
- LL(k), SLR(k), LR(k), LALR(k)…
- Take CMSC 430 for more details

One simple technique: recursive descent parsing
- This is a top-down parsing algorithm

Other algorithms are bottom-up
Recursive Descent - Intuition

Non-terminal

Terminal

E → id = n | { L }
L → E ; L | ε

(Assume: id is variable name, n is integer)

Show parse tree for
{ x = 3 ; { y = 4 ; } ; }

lookahead

Start at the top, try productions in order
Recursive Descent - Example

E → id = n | { L }
L → E ; L | ε

(Assume: id is variable name, n is integer)

Show parse tree for
{ x = 3 ; { y = 4 ; } ; }
Recursive Descent

At each step, we'll keep track of two facts

• What grammar element are we trying to match/expand?
• What is the lookahead (next token of the input string)?

At each step, apply one of three possible cases

• If we’re trying to match a terminal
  ➢ If the lookahead is that token, then succeed, advance the lookahead, and continue

• If we’re trying to match a nonterminal
  ➢ Pick which production to apply based on the lookahead

• Otherwise fail with a parsing error
Additional Material

- Video series by Alex Aiken
  - https://www.youtube.com/playlist?list=PLDcmCggUL9rxPoVn2ykJFc8TOpLyDU5gx
- 6.3 - Recursive Descent Overview
- 6.4 - Recursive Descent Implementation
- Other parsing algorithms
- Parsing slides by Michael Hicks (CMSC 330)
Project Submission Demo
Summary

- Project Specification Updates
  - Passwords
  - Access Control
  - Network is secure!
- Networking basics
  - Socket programming tutorials on website
- Scanning and Parsing basics
  - Additional materials on website
- Project submission demo
- Daily status reports start today: ter.ps/388Nreport
In-class Build Time!

- Divide up into teams and spread out
  - You can leave this room, but stay on this floor
  - Send us a message in Slack with where you go

- Some possible-todos:
  - Merge design documents
  - Discuss logistics
    - Ex: language, libraries, divide-and-conquer vs. pair programming
  - Start writing code!

- Instructors will come around to talk