CS50’s Introduction to Cybersecurity

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 3

- Securing Software
- Phishing
- Code Injection
- Reflected Attack
- Stored Attack
- Character Escapes
- HTTP Headers
- SQL Injection
- Prepared Statements
- Command Injection
- Developer Tools
- Server-Side Validation
- Cross-Site Request Forgery (CSRF)
- Arbitrary Code Execution (ACE)
- Open-Source Software
- Closed-Source Software
- App Stores
- Package Managers
- Bug Bounty

- Identifying Vulnerabilities
- Summing Up

Securing Software

- This is CS50's Introduction to Cybersecurity.
- This week, let's focus on securing software that you use or software that you create.
- Last time, we introduced various attacks that could be used by adversaries to obtain information from you.

Phishing

- We introduced an attack called *phishing*, where an adversary tricks you into providing information of some kind.
- In the source code of a website, for example, in the language of HTML, you may see code like this:

  ```
  <p>...</p>
  ```

  Notice in the code above that a paragraph starts and ends. It begins with an opening tag and a closing tag.
- Similarly, links in web pages use a specific type of tag called an *anchor tag* to take users from one web page to another.
- Such code looks like this:

  ```
  <a href="https://harvard.edu">Harvard</a>
  ```

  Notice that this code is an anchor tag that allows the user to click the word 'Harvard' and visit `harvard.edu`.
- On an actual web page, you could move your mouse over such a link and see where this precise link will take you.
- Adversaries may take advantage of your lack of attention to claim you are linking to one web page when you are actually linking to another one.
- For example, an adversary could provide code like this:

  ```
  <a href="https://yale.edu">https://harvard.edu</a>
  ```

  Notice that this code is an anchor tag that tricks the user into clicking `https://harvard.edu` when it actually browses to `yale.edu`. While the user will think they are clicking a link for Harvard, they are actually browsing to Yale.

- You can imagine how this strategy can be used by an adversary to trick you into thinking you are visiting one website when you are actually visiting another.
- Adversaries often create fake versions of websites for the sole purpose of tricking users into inputting sensitive information into those websites. For example, if you were a Harvard student visiting such a fake Harvard website, you may attempt to log in and provide your username and password to an adversary.

Code Injection

- *Cross-site scripting*, or *XSS*, is a form of attack where a website is tricked into running malicious code via a user's input.
- For example, on Google, when you type a search for the term "cat", notice how the term appears on the screen elsewhere, showing you how many results are present for this search.
- Imagine that an adversary who knows a bit about the web could insert code as input as a way of tricking the website into running such code.
- For example, consider the following code that could be inserted into a search field:

  ```
  <script>alert('attack')</script>
  ```

  Notice how this script displays a notification that says "attack." While the Google website will not display such a notice due to security, this is representative of what an adversary could attempt.
- If a website blindly copies user input and outputs what the adversary typed, this is a major security concern.

Reflected Attack

- A *reflected attack* is one that takes advantage of how websites accept input to trick a user's browser into sending a request for information that results in an attack.
- Imagine that a user could be tricked to click a link structured as follows:

  ```
  <a href="https://www.google.com/search?q=%3Cscript%3Ealert%28%27attack%27%29%3C%2
  ```

  Notice that this link includes the exact script presented above that is intended to create an attack alert on the user's screen.
- The user's actions trick their own web browser into reflecting back an attack upon the user.

Stored Attack

- A website could be vulnerable to an attack where it is tricked into storing malicious code.
- Imagine where one could email malicious code. If an email provider blindly accepts any code sent to it, any person receiving the malicious code may become a victim of an attack.

Character Escapes

- Services use *character escapes* as a way by which to protect against such attacks. Software should escape potentially troublesome characters that represent common coding-based characters.
- For example, code like the following…

  ```
  <p>About 6,420,000,000 <script>alert('attack')</script></p>
  ```

  will be outputted by secured software as…

  ```
  <p>About 6,420,000,000 &lt;script&gt;alert('attack')&lt;/script&gt;</p>
  ```

  While a bit cryptic, notice that `<` is used to escape potential characters that would pose a threat to the software. The output of the above then becomes a text-only representation of the malicious code.
- Commonly escaped characters include:
  - `<`, which is is the less-than sign, “<”
  - `>`, which is is the greater-than sign, “>”
  - `&`, which is the ampersand, “&”
  - `"`, which is the double quote, “, itself
  - `'`, which is the single quote, ‘

HTTP Headers

- Recall that *HTTP headers* are additional instructions that are provided to the browser.
- Consider the following header:

  ```
  Content-Security-Policy: script-src https://example.com/
  ```

  Notice that the above security policy in a website header will only allow Javascript to be loaded via separate files, usually ending in `.js`. Thus, `<script>` tags inside HTML will not be run by the browser when this security policy is in place.

- Similarly, the following header will allow CSS only from `.css` files:

  ```
  Content-Security-Policy: style-src https://example.com/
  ```

  Notice that `style-src` indicates that only CSS that is loaded from a `.css` file will be permitted.

SQL Injection

- *Structured query language* or *SQL* is a programming language that allows for retrieving specific information from a database.
- Consider how an adversary may attempt to trick SQL into executing malicious code.
- Consider the following SQL code:

  ```
  SELECT * FROM users
  WHERE username = '{username}'
  ```

  Notice that here a user's inputted username is inserted into the SQL code.
- Never trust a user's input.
- All input should be scrubbed such that all user input is escaped.
- Suppose that an adversary inserted the following code into the username field:

  ```
  malan'; DELETE FROM users; --
  ```

  Notice that in addition to a username, malicious code is inserted.
- What results because of the above input is the following:

  ```
  SELECT * FROM users
  WHERE username = 'malan'; DELETE FROM users; --'
  ```

  Notice that the adversary's malicious input adds additional code to the query. What results is the deletion of all users from the system. Every account on the system is deleted.
- Suppose that a user is asked for a username and password as follows:

  ```
  SELECT * FROM users
  WHERE username = '{username}' AND password = '{password}'
  ```

  Notice a user is asked for a username and password.
- An adversary could insert the following into the password field:

  ```
  ' OR '1'='1
  ```

- The following SQL code will then execute:

```
SELECT * FROM users
WHERE username = 'malan' AND password = ''
OR '1'='1'
```

Notice grammatically, this results in providing all the users in the database.

- To see this more plainly, notice the additional parentheses added below:

```
SELECT * FROM users
WHERE (username = 'malan' AND password = '')
OR '1'='1'
```

Notice that this code will either show all users where the username and password combination are true `OR` all users.

- Effectively, the above input is always true. Through this security vulnerability, the adversary may have information about all users on the system, including the administrator.

Prepared Statements

- *Prepared statements* are pre-designed snippets of code that correctly handle many database functions, including user input.
- Such statements, for example, ensure that user-inputted data is properly escaped.
- A prepared statement will take code as the following…

```
SELECT * FROM users
WHERE username = '{username}'
```

and replace it with…

```
SELECT * FROM users
WHERE username = ?
```

- Prepared statements will look for any `'` characters and replace them with `''`. Hence, our previous attack shown above would be rendered by the prepared statement:

```
SELECT * FROM users
WHERE username = 'malan''; DELETE FROM users; --'
```

Notice that the `'` at the end of 'malan' is replaced with `''`, rendering the malicious code inoperable.

- What results is that malicious characters are escaped, such that malicious code cannot run.

Command Injection

- A *command line interface* is a method by which to run a computer system using text-based commands, as opposed to clicking on menus and buttons.
- A *command injection* attack is one that issues a command on the underlying system itself.
- Should a command be passed from user input to the command line, the effect could be disastrous.
- Two common places of vulnerability are that of `system` and `eval`, wherein if you pass user input without sanitization, malicious commands could be issued on a system.
- Always read the documentation to learn how to escape the user's input.

Developer Tools

- Let's return to the world of HTML and the web.
- In the context of the browser, *developer tools* allow you to poke around some of the underlying code in a web page.
- Consider what we can do using developer tools. Here is the code for a textbox:

  ```
  <input disabled type="checkbox">
  ```

  Notice that this creates a type of input called a checkbox. Further, notice that this textbox is disabled and not usable via the `disabled` attribute.
- Perhaps a challenge with the security of HTML is that the HTML is resident on their computer. Therefore, the user could be able to make changes to a local file on their computer.
- A user with access to HTML on their own computer via developer tools can change HTML.

  ```
  <input type="checkbox">
  ```

  Notice that a local copy of the HTML here has the `disabled` attribute removed.
- You should never trust *client-side validation* alone.
- Similarly, consider the following HTML:

  ```
  <input required type="text">
  ```

  Notice how this text input is `required`.
- Someone with access to developer tools could remove the requirement of this input as follows:

  ```
  <input type="text">
  ```

  Notice the `required` attribute is removed.
- Again, never trust that client-side validation will ensure the security of your web application.

Server-Side Validation

- *Server-side validation* provides security features to ensure that user input is appropriate and safe.
- While this topic is beyond the scope of this class, simply trust in the principle that user input should be validated on the server-side. Never trust user input.

Cross-Site Request Forgery (CSRF)

- Another threat is called *cross-site request forgery* or *CSRF*.
- Websites use two primary methods to interact with users called `GET` and `POST` methods.
- `GET` gets information from a server.
- You might consider how Amazon uses the `GET` method for the following HTML:

  ```
  <a href="https://www.amazon.com/dp/B07XLQ2FSK">Buy Now</a>
  ```

  Notice how with a single click, one can buy this product.
- You can imagine how one may trick someone into buying something they don't intend.
- One could provide an image that is automatically attempting to buy a product:

  ```
  <img src="https://www.amazon.com/dp/B07XLQ2FSK">
  ```

  Notice that no image is provided here. Instead, the browser will attempt to execute the `GET` method using this web page, making a possibly unauthorized or unwanted purchase.
- Similarly, adversaries can use the `POST` method to make unauthorized purchases.
- Consider the following HTML of the 'Buy Now' button:

  ```
  <form action="https://www.amazon.com/" method="post">
  <input name="dp" type="hidden" value="B07XLQ2FSK">
  <button type="submit">Buy Now</button>
  </form>
  ```

  Notice how a web form, as implemented above, could naively make one believe that one is safe from an unauthorized purchase. Because this form includes a `hidden` value that is used by Amazon, hypothetically, for validation, it may make a programmer think that users are safe.
- However, as is the case with many exploits, this feeling of safety is misplaced.
- Indeed, adding only a few lines of code could subvert the above. Imagine an adversary has the following code on their own website (not Amazon's):

  ```
  <form action="https://www.amazon.com/" method="post">
  <input name="dp" type="hidden" value="B07XLQ2FSK">
  ```

```
<button type="submit">Buy Now</button>
</form>
<script>
document.forms[0].submit();
</script>
```

Notice how a few lines of code on an adversary's website could locate a form and submit it automatically.

- This ability to trick a user into executing commands on another website is the essence of a CSRF.
- One way to protect against an attack such as this is a *CSRF token*,where a secret value is generated by the server for each user. Thus, a server will validate that one's CSRF token presented in their submissions matches the token expected by the server.
- These tokens are often submitted via HTML headers.

Arbitrary Code Execution (ACE)

- *Arbitrary code execution*, or *ACE*, is the act of executing code that is not part of the intended code within software.
- One such threat is called *buffer overflow*, where software is overwhelmed with input. Such input overflows into other areas of memory, causing the program to malfunction. For example, the software may expect input of a short length, but the user inputs an input of a massive length.
- Another similar threat is called a *stack overflow*, where overflows can be used to insert and execute malicious code.
- Sometimes, attacks such as these can be used for *cracking* or bypassing the need to register or pay for a piece of software.
- Further, attacks such as these can be used for *reverse engineering* to see how code functions.

Open-Source Software

- One way to circumvent threats like this is to use and make *open-source software*. Such software's code is published readily online for anyone to see.
- One can audit the code and make sure that there are fewer security threats.
- These pieces of software are still vulnerable to attacks.

Closed-Source Software

- *Closed-source software* is the opposite of open-source software.

- Such software's code is not available to the public and, therefore, may be less vulnerable to adversaries.
- However, there is a tradeoff between open-source software, where thousands of eyes are looking for vulnerabilities in the software, and closed-source software, where code is hidden from public view.

App Stores

- *App stores* are run by entities like Google and Apple, where they monitor submitted code for adversarial intent.
- When you install only authorized software, you are far more protected than installing software from any developer without using an app store.
- App stores employ encryption to accept only software or code that is signed by authorized developers. In turn, app stores sign software with a digital signature. Thus, operating systems can ensure that only authorized, signed software is being installed.

Package Managers

- *Package managers* adopt a similar signing mechanism to ensure that what you download from third parties is trustworthy. However, there is no guarantee that one is entirely safe.
- Still, we are always attempting to raise the bar for adversaries to install adversarial code.

Bug Bounty

- *Bug bounties* are paid opportunities for individuals to discover and report vulnerabilities in software.
- Bounties such as these may effectively influence would-be adversaries to opt to be paid for finding vulnerabilities rather than deploying them as an attacker.

Identifying Vulnerabilities

- Developers can examine a database of *common vulnerabilities and exposures* or *CVE* numbers to see what adversaries are doing worldwide.
- Further, they may examine the *common vulnerabilities scoring system* or *CVSS* to see how severe such threats are.
- There is also an *exploit prediction scoring system* or *EPSS* that allows developers to see the potential for vulnerabilities worldwide to allow them to prioritize their security efforts.

- *Known exploited vulnerabilities* or *KEV* database is a list of known vulnerabilities.

Summing Up

In this lesson, you learned about securing software. You learned…

- How adversaries use attacks such as phishing, code injection, reflected attacks, SQL injection, and stored attacks to infiltrate software;
- How character escapes, HTML headers, prepared statements, and server-side validation may help thwart the attacks of adversaries;
- How app stores, package managers, and developer signatures help protect against the installation of malicious code;
- How experts in the cybersecurity field track exploits.

See you next time!