ENPM 809W
Introduction to Secure Software Engineering

Gananand Kini

Lecture 4

Input related security bugs - Defenses

MITRE | SOLVING PROBLEMS FOR A SAFER WORLD™

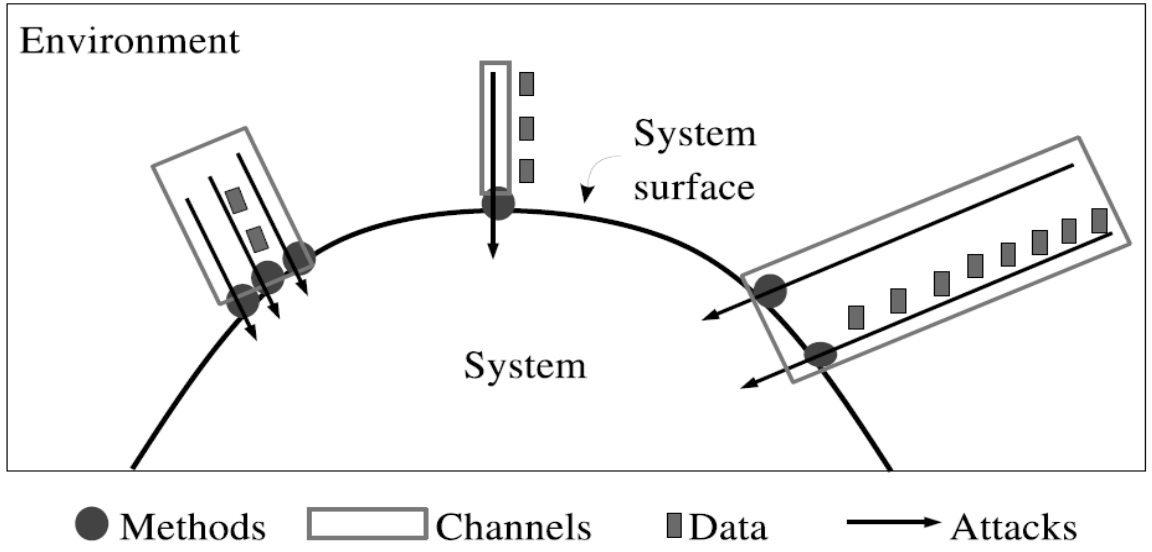
Outline

- Input Validation Issues
- Input Encoding Issues
- Input Filtering (Allow List vs Deny List approaches)
- Input Trust – Zero Trust Policy
- Regular Expressions
- Hide sensitive information
- Planned recovery
- LANGSEC – Hammer Parser Validator

Input Validation

Review: Attack Surface – Formal Definition

- Attacker can attack using channels (e.g., ports, sockets), invoke methods (e.g., API), and send data items (input strings or indirectly via persistent or stored data)
- A system's attack surface – Subset of the system's resources (channels, methods, and data) that can be used in attacks on the system.
- More attack surfaces likely means it is easier to exploit and cause more damage.



Source: *Attack Surface Metric*, Pratyusa K. Manadhata, CMU-CS-08-152, November 2008

What should a defender do?

- **Applying the security principle of reducing the attack surface:**
  - Defender can reduce or disable channels (e.g. ports, files etc.) and methods (APIs etc.).
  - Prevent and Restrict access for attackers to the channels and ports by adding authorization, authentication, validation or filtering.
- **Understand what attack surfaces are possible with the current or proposed system design**
  - Make sure you understand all possible entry points
  - That means knowing all possible channels, methods and types of data coming into or going out of the system.
- **Add impediments to attackers to prevent them from getting unauthorized access while allowing and enabling authorized and legitimate users access to the system functionality.**

Sources:

Potential Improper Input Validation weaknesses

- CWE-20: Improper Input Validation
- CWE-790: Improper Filtering of Special Elements (Poor filtering weakness)
- CWE-121/CWE-122: Buffer overflow Weakness
- CWE-89: SQL Injection Weakness
- CWE-79: Cross-site Scripting (XSS) Weakness
- CWE-77/CWE-78: Command Injection/OS Command Injection Weakness
- CWE-94: Code Injection Weakness
- CWE-22: Path Traversal Weakness (Path to restricted directory allowed weakness)

Input Filtering

- **You can filter your inputs.**
- **Two approaches:**
  - A “deny” list approach – Make a list of dis-allowed inputs and filter to remove and deny those inputs.
  - An “allow” list approach – Make a list of only valid allowed inputs and filter to allow those inputs while removing everything else.
- **While a “deny” list approach sounds really cool, it really isn’t!**
  - Why? Attackers are many much cleverer than you and can easily find ways around your “deny” list.
- **An “allow” list should be the default approach.**
  - This gives attackers very little maneuvering room.
  - If you are too strict, users may complain.
- **Only use a “deny” list approach if you are working with a small enumeration of options and are confident it cannot be thwarted easily.**

Overflow and Underflow weaknesses

- Use the appropriate data type for numerical variables that accommodate the range of the numbers being represented.
- Use unsigned data types for numerical variables that hold only positive numbers.
- If fractions are not allowed, use integral types wherever possible.
- The floating point standard IEEE-754 may work ok for the general case, but drawbacks do exists for other cases.
- .NET handling of floating points: Floating Point Numeric Types
- Many proposals like UNUM or Bounded Floating Point that are attempting to present alternative floating point representations that could be backwards compatible with IEEE-754.
- You can force an “allowed” number range when taking input. Again, an allow list approach can help avoid such issues.

Validate untrusted input (.NET)

- **You should validate any input that comes into the software system.**

- **For numbers, there are:**
  - System.Configuration.LongValidator .NET class.
  - System.Configuration.IntegerValidator .NET class.
- **Many more validators exist in the System.Configuration namespace.**

- **For input file types like XML in .NET:**
  - Use the System.Xml.XmlDocument.Validate method.

Server-side versus Client-side input validation

- **Imagine a web application sends this HTML to a web browser as part of a form:**

```
<input name="lastname" type="text" id="lastname" maxlength="100" />
```

- **Does this HTML provide security-relevant input validation (e.g., to ensure that last names are no more than 100 characters long)?**

Server-side versus Client-side input validation

- **Imagine a web application sends this JavaScript to a web browser:**

```
function regularExpression() {
  var a=null;
  var first = document.forms["form1"]["firstname"].value;
  var firstname_pattern = /^[A-Z][a-z]{1,30}$/;
  if(first==null || first=="") {
    alert("First name cannot be null");
    return false;
  } else {
  a=first.match(firstname);
  if (a==null || a=="") {
    alert("First name must be of form Xxxxxx");
    return false;
    }
  }
```

- **and also sends this HTML that activates it:**

```
<form action="register.jsp" name="form1" onsubmit="return regularExpression()" method="post" >
```

- **Does this JavaScript provide security-relevant input validation?**

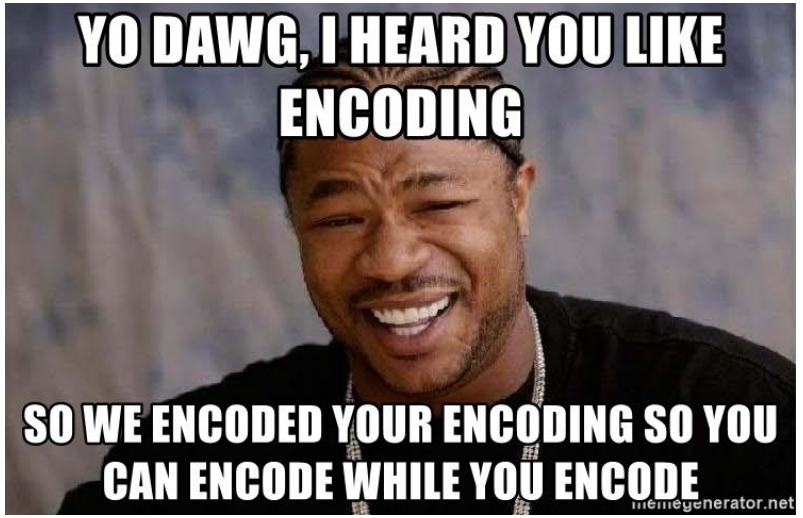
Server-side versus Client-side input validation

- Input validation in web application and client-server applications can be performed on the client-side or the server-side.
- Does that mean you should only rely on one validation? No!
- Assume the attacker controls the client-side and can directly craft requests bypassing the client-side code completely.
- Client-side validation may be helpful to the user of the application, but by no means is it secure nor is it to be relied upon.
- It is ok to do client-side input validation, but should be performed regardless on the server-side.

Neutralizing Inputs using Encoders and Escapes (Metacharacters)

- Idea here is to encode inputs received by users such that when the content is included as part of the executing code, it hopefully won't execute as code.
- Can use Base64 encoding to capture null bytes. Lazy approach when you cannot validate input strictly.
- Other languages have similar support.
- Cannot use on content that is displayed. Ex.: VGhpcyBpcyB0aGUgYmVzdCEK.

- .NET Framework 4.5+ provides an Anti-XSS library (not enabled by default).
- Use System.Web.Security.AntiXSSEncoder (Encodes a string for use in HTML, XML, CSS and URL strings).



Source:

1. "Yo Dawg, I heard you like encoding so we encoded your encoding so you can encode while you encode." Meme Generator. https://memegenerator.net/instance/43802280/yo-dawg-yo-dawg-i-heard-you-like-encoding-so-we-encoded-your-encoding-so-you-can-encode-while-you-en.

Neutralizing Inputs using Encoders and Escapes (Metacharacters)

- **Escape special metacharacters such that they do not impact the software system. We will look at special cases of this as part of attacks on Logging and Error Handling.**
- **This can be performed:**
  - When accepting input or,
  - Right before the output is sent to a target interpreter. (E.g. The function in the webserver right before rendering the HTML) or,
  - Before storing in a database or any kind of data store.
- **Cost and performance considerations might affect this.**

Multi-stage input filters

- **You can check your input in stages. For example:**
  - First try maximum length and encoding (UTF-8) check
  - Next can do a regex based “allow” list check
  - Then, parse “values”. If a numerical value exists, check range. If it’s a Boolean, do a sanity check against other inputs.
  - Finally, you can do any complex checks as needed. For example, only allow Monday, Tuesday, Wednesday, and Saturday.
- **This kind of “allow” list approach is much better than encoding or escapes if the impact to the software system design/implementation performance or cost is minimal.**

Using Regular Expressions

- **For .NET don’t use System.Text.RegularExpressions for validation purposes. Only use it for string searching and only if extremely necessary.**

- **Use System.Configuration.RegexStringValidator class instead.**

Countering Code Injection

- **As seen previously, environment variables can be dangerous (especially since they are data as code)**
  - Extract the variable data, validate it and then erase the environment variable.
  - Did you know you can unset an environment variable?
- **Do use an ‘allow’ list of commands that are allowed to be run by users.**
- **For those commands, ensure users and roles are authenticated and authorized before executing the command. Assume the environment does not do this or check this. You can explicitly document this as well.**
- **Separate out channels that require use of commands and isolate only to necessary parts of the software system. Attackers can use commands in combination to achieve their goals.**

Countering Cross-Site Scripting (XSS)

- Escape Outputs
- Apply Input Filtering seen previously
- Set HttpOnly on cookies (not a cure, but helps a little)
- Setting Content Security Policy (CSP) HTTP Header (again not a cure, but helps)

Countering Cross-Site Scripting (XSS): Content Security Policy

- **Content Security Policy (CSP) is W3C Recommendation**
- **Defines "Content-Security-Policy" HTTP header**
  - If used, creates ***allow*** list of sources of trusted content for this webpage
  - Compliant browsers only execute/render items (Javascript) from those sources
    - Chrome 16+, Safari 6+, and Firefox 4+. IE 10 has very limited support
  - Twitter and Facebook have deployed this
- **Typically must modify website design to fully use it**
  - E.g., move JavaScript into separate files, otherwise receiving browser can't distinguish 'allow' listed & malicious JavaScript
  - Also blocks eval() function. Can be a bummer if you have third party JavaScript libraries relying on this function. But you shouldn't be using such insecure libraries anyway…
- **Only works when users use compliant browsers**
- **CSP is now up to version 3 (June 2021). See links below.**
- **More info:**
  - https://developers.google.com/web/fundamentals/security/csp
  - http://www.w3.org/TR/CSP/

MITRE

Web Applications - Hardening headers on web applications

- **Server-side web application receive HTTP headers from untrusted users.**
- **Do not trust data from untrusted users including HTTP headers!**
- **Some web applications, frameworks or programmers use these header values to formulate URL links because:**
  - It is convenient to use these header variables, since all the pages are hosted on the same webserver usually.
- **However, this is insecure, because the attacker can provide/change the HTTP header value!**
- **Examples of *INSECURE* coding in:**
  - PHP:

```
<a href="<?=$_SERVER['HTTP_HOST']?>/login">Login</a>
```

  - ASP.NET:

```
public class WorkersController : ApiController
{
        // GET: api/Worker
        [Route("api/Workers")]
        [HttpGet]
        public HttpResponseMessage Get()
        {
         string host= "";
         System.Net.Http.Headers.HttpRequestHeaders headers = this.Request.Headers;
            HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.Created);
            response.Headers.Add("Access-Control-Allow-Origin", "*");
         if (headers.Contains("Host"))
         {
             host = headers.GetValues("Host").First();
             response.Body.Add("Refresh", "Refresh: 5; url=" + host + “/redirect");
         }
        }
    }
}
```

Web Applications - Hardening headers on web applications

- **Content Security Policy (CSP) – already noted**
- **HTTP Strict Transport Security (HSTS)**
  - “Only use HTTPS from now on for this site”
- **X-Content-Type-Options (as "nosniff")**
  - Don’t guess MIME type (& confusion it can bring)
- **X-Frame-Options**
  - Clickjacking protection, limits how frames may be used
- **X-XSS-Protection**
  - Force enabling filter to detect likely (reflected) XSS by monitoring requests / responses
  - On by default in most browsers
- **Quick scan tool: https://securityheaders.io/**

Same-Origin Policy

- **Security model of JavaScript running in a web browser. For example:**

```
<script type="text/javascript" src="https://example.com/script.js" />
```

- **Pages may only access data in a different web page if their origin is same (URI scheme + host name + port#). In the example: Scripts from https://example.com are not allowed by the browser if current application is being served from or running from https://example2.com.**
- **Primary exceptions of same-origin policy:**
  - Some old operations (generally predating JavaScript), e.g., img src, submitting POST forms, and including scripts
  - Corner cases for pseudo-protocol URI schemes (e.g. file://)
  - Windows (or frames) with scripts that set document.domain to the same value [can interact]
  - Message-passing text via postMessage() [can interact].
  - WebSockets (request sets origin, server checks if origin ok).
  - Cross-Origin Resource Sharing (CORS).

Source:

1. Vickie Li. Hacking the same origin policy. https://medium.com/swlh/hacking-the-same-origin-policy-f9f49ad592fc. Retrieved July 10, 2021.

Cross-Origin Resource Sharing (CORS)

- Enabling this feature relaxes single-origin policy of web browsers (be careful!). See https://code-maze.com/enabling-cors-in-asp-net-core/.
- When JavaScript makes cross-origin request, “Origin:” sent
- Server examines origin & replies in HTTP header if allowed:
  - Access-Control-Allow-Origin: permitted origin or “*” (or null)
  - Access-Control-Allow-Methods: permitted methods (e.g., GET)
  - Access-Control-Allow-Credentials: if true, share credentials (!)
- Web browser checks if origin specifically allowed (default “no”)
- Thus, both server & web browser must agree access permitted
- Sometimes GET & POST are directly requested (see W3C spec)
- In many cases web browser first makes “preflight” request to server using OPTIONS to determine permission before making actual request
- ***Beware*** of Access-Control-Allow-Credentials
  - If “true” then credentials directly shared – only use if you totally trust other site & it’s absolutely necessary!

For more information, see “Exploiting CORS Misconfigurations for Bitcoins and Bounties” by Jim Kettle, Oct. 14, 2016, http://blog.portswigger.net/2016/10/exploiting-cors-misconfigurations-for.html

Avoiding Path Manipulation

- **Don’t allow user to control the path.**
- **Use of pre-configured directories in configuration files in conjunction with absolute paths.**
- **Applying the least privilege principle, isolate use of directories to only the necessary software components and with only the privileges required.**
- **Use of generative temporary directory to store temporarily during validation or filtering of content. However, beware of resource exhaustion attacks.**
  - Dotnet - https://stackoverflow.com/questions/278439/creating-a-temporary-directory-in-windows
  - Linux – mkdtemp()
- **Ensure permissions are set correctly on the temporary directory with close to NO privileges.**
- **If the application is managing several such files between different users, ensure path isolation within directories so one malicious user cannot reach another user’s files within the directory.**

Web Application Path Redirections Check

- **Web apps frequently redirect and forward users to other pages, sites or resources.**
- **Do not use unvalidated or untrusted user input to determine the target location or destination of such redirects! See the example in the slide on 'Web Applications - Hardening headers on web applications'**
- **Solutions:**
  - Simply avoid using redirects or forwards.
  - If not, do not use user provided parameters to determine target or destination. Think of 'allow' list strategy here.
  - Applications can override OWASP ESAPI (Java only, .NET exists but may not be working) to override the sendRedirect() function to ensure all redirects are safe. This method again follows the 'allow' list strategy.
  - Use LocalRedirect in .NET. See Prevent open redirect attacks in ASP.NET Core | Microsoft Docs.

How to avoid SQL Injection

- Validate, Filter and/or encode user input parameters (as discussed during the Input Validation lecture)
- Use parameterized queries with appropriate termination.
- Utilize the principle of least privilege and only allow queries involving user input to be run as a non-privileged user.
- Do not use shared database accounts across multiple software systems.
- Follow guidance here for .NET: https://www.veracode.com/blog/secure-development/top-security-anti-patterns-aspnet-core-applications

Countering command injection

- Use an ”allow” list approach to a set of allowed commands to be run.
- Again, follow the principle of least privilege and only limit the privileges to the ones necessary for that command.
- Limit the number of characters accepted as input for parameters being passed to the command where possible.
- Validate, filter and encode user provided input if possible to neutralize any meta-characters that have meaning in the OS or environment the command is being run under.s

Program Defensively

- **Developing secure software is not just knowing and countering common weaknesses.**
- **In addition to prevention as seen so far, focus should be emphasized also on detection of faults and recovery!**
- **Make the application resilient to failure.**
- **The weakness lists provided are a good starting point, however its up the development team to ensure:**
  - Avoiding common past mistakes (build on past reviews of the code)
  - Provide evidence for assurance of security properties
  - Provide for adequate recovery measures

Labs

How to do the defense lab for the WebGoat .NET application

- **Please read the lab handout carefully.**
- **The lab handout has specific instructions on which exercises to perform and in what order.**
- **Remember the weakness you identify has to be as specific as possible.**

- **When you find a weakness, please provide the following as part of a document writeup and in the given format as part of your submission:**
  - Question Number.
  - CWE identifier for the discovered weakness.
  - The web application URL.
  - Any relevant headers or parameters for the request/response.
  - Where is the weakness in the source code? (Filename, Line number)
  - What is the weakness in the source code? Describe the coding weakness.

Fix Lab Submission Criteria

- **You will be submitting the following:**
  - A writeup containing the information on previous slide.
  - A text file containing:
    - Your UMD directory id.
    - The last commit id for your code that implements all the fixes.
  - Example:

```
jdoe
f01234cad
```

Next time …

- **Cryptography related security bugs - Attacks**