Validation and Sanitization of User-Entered Data for Security Purposes

Elisabed Asabashvili
University of Georgia
Tbilisi, Georgia
e-mail: z.asabashvili@ug.edu.ge

***Abstract*—The integration of digital devices and the internet has transformed modern life, improving communication, work, and daily routines. However, this digital shift introduces new cybersecurity challenges. The increasing dependence on online platforms has made individuals, businesses, and organizations more vulnerable to cyber threats, elevating data security to a critical concern. The pandemic, the rise of cryptocurrencies, and the shift to remote work have further created ideal conditions for cybercriminals to exploit, making cybersecurity an essential component of the modern business environment.**

Cybersecurity encompasses the use of technologies, processes, and controls to defend systems, networks, software, and data from cyberattacks, such as phishing, data breaches, identity theft, financial fraud, and ransomware.

The Ruby programming language—known for its flexibility and rich feature set—plays a valuable role in cybersecurity by enabling the development of tools and scripts for threat detection and vulnerability assessment. This article showcases code written in Ruby, utilizing validation and sanitization processes to enhance application security. It explains the importance of validation, which ensures that user input meets predefined formats, such as valid email addresses or phone numbers, thereby preventing the entry of invalid or malicious data. The article also introduces sanitization, a process that cleans input by removing harmful characters to guard against threats like cross-site scripting (XSS) and SQL injection.

Together, validation and sanitization form a robust defense mechanism that helps protect applications from security vulnerabilities, maintain data integrity, enhance user experience, and ensure system stability.

***Keywords*—Validation, Sanitization, Regex, Ruby Libraries, Cybersecurity.**

I. INTRODUCTION

In 2020, Cybercrime Magazine reported that by 2025, cybercrime would cost the global economy an estimated $10.5 trillion annually [3]. Furthermore, global spending on combating cybercrime is projected to grow by nearly 15% per year over the next four years. This forecast was echoed by Cybersecurity Ventures, a leading research and media organization known for its authoritative insights, market analysis, and forecasts on cybersecurity trends and the economic impact of cybercrime. According to its “Official Cybercrime Report 2025”, cybercrime is expected to cost the world $12.2 trillion annually by 2031 [4]. In the United States alone, cybercrime-related costs are rising sharply and are forecasted to reach approximately $1.82 trillion by 2028 [5].

Cybersecurity is a broad field, typically divided into several key areas [11]:

- Application Security;
- Cloud Security;
- Identity Management and Data Security;
- Mobile Security (tablets, cell phones, and laptops);
- Network Security.

As is known, the term “hacking”, which originated in the 1960s, refers to unauthorized access to personal information [8]. Today, with the advent of artificial intelligence (AI) and process automation [1], cybercriminals are able to conduct attacks with unprecedented speed and sophistication. Attacks on IT infrastructure, government and military systems, and on personal devices are increasing rapidly. Many organizations remain ill-prepared to defend against such attacks, prompting a global push to invest in advanced cybersecurity tools and strategies to safeguard digital systems for individuals, businesses, and governments alike.

Meanwhile, hacking itself has evolved—often progressing as fast, or even faster, than defensive measures. Two main categories of hackers have emerged [7]:

Malicious hackers, who exploit systems with the intent to cause harm, steal data, or disrupt operations.

Ethical hackers, who use the same techniques to identify vulnerabilities but report them to the appropriate security teams to prevent exploitation.

Ethical hackers play a critical role in modern cybersecurity. Their responsibilities typically include:

- Creating vulnerability testing scripts;
- Developing security tools;
- Conducting risk assessments;
- Reviewing security policies;
- Training network security personnel.

This practice, known as ethical hacking, serves as a proactive defense strategy—helping organizations discover

[1] Artificial intelligence (AI) and automation are related but distinct concepts. While automation focuses on performing tasks according to fixed rules, AI embeds intelligence by allowing systems to learn, adapt, and make decisions autonomously. AI-powered automation combines the two, enabling more flexible and intelligent process automation.

and fix weaknesses before they can be exploited by malicious actors.

In today's increasingly digital world, cyber threats are becoming more advanced and widespread. As a result, cybersecurity has become a fundamental component of software development, especially since applications often handle sensitive data and perform critical functions that require robust protection.

Although Ruby is not the only language used in cybersecurity, it remains a valuable tool due to its simplicity, readability, and rich set of libraries. Ruby is a dynamic, open-source programming language designed for productivity and ease of use [6]. Its clean and expressive syntax makes it ideal for writing custom scripts and building specialized tools for security analysis and penetration testing. For cybersecurity professionals, Ruby’s flexibility makes it a practical and efficient language for developing solutions aimed at protecting applications and digital infrastructure.

II. Threat Modeling in Application Development

Threat modeling involves identifying potential threats to a system and understanding how, why, and where an attacker might exploit it. In web applications, user input is often the most vulnerable entry point. Understanding the attack surface helps in applying the right validation and sanitization techniques. Models such as STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) are commonly used to guide security decisions.

III. User Input Validation and Sanitization

Input validation and sanitization are essential components of web application security, ensuring that user input follows the expected format and is free from malicious content. These practices help prevent common security vulnerabilities such as SQL injection, cross-site scripting (XSS), and other forms of data manipulation. In Ruby, developers have access to a variety of built-in methods and external libraries to implement robust validation and sanitization mechanisms [1, 12].

This article demonstrates the use of several such techniques by presenting Ruby code that performs both input validation and, when necessary, sanitization—ensuring data is clean and secure before further processing [2].

The article includes a script that checks the validity of an email address, illustrating how to validate and sanitize user input effectively within a Ruby application.

IV. Complex Input Scenarios and Real-World Attack Vectors

As stated, the report used a programming language to validate and sanitize input data using both built-in Ruby methods and external libraries. It should be noted that Sanitization is the process of removing or modifying potentially dangerous data from user input to prevent security vulnerabilities. It protects systems from malicious input that can be used to execute malicious commands, leading to data leaks, unauthorized access, and other security issues.

Validation in Ruby, particularly within the context of cybersecurity, primarily refers to input validation (to prevent attacks like SQL injection, Cross-Site Scripting (XSS), and other injection vulnerabilities) and data validation (to maintain data integrity and consistency within the application and database). This is a critical security measure to prevent various attacks by ensuring that data received by an application is in the expected format and free from malicious content.

The article includes a script that checks the validity of an email address, illustrating how to validate and sanitize user input effectively within a Ruby application.

Validation ensures that input conforms to expected formats. For example:

- Emails must follow a standard pattern (e.g., user@example.com)
- Phone numbers must contain only digits and specific characters
- Form fields must not exceed a reasonable length

Ruby makes this easy with Regular Expressions (Regex) and libraries like ActiveModel::Validations that enforce rules seamlessly. These functions verify that the email address string conforms to a valid pattern. This check can prevent incorrect or malicious data from entering the system.

Even validated input can carry hidden threats. Sanitization involves stripping or escaping harmful characters or scripts that could compromise the application. For instance, HTML tags in user comments can be sanitized to prevent XSS attacks.

Ruby offers libraries like sanitize or Loofah. By removing dangerous tags or attributes, sanitization helps neutralize potentially harmful input before it interacts with the application or database.

In addition to guarding against XSS and SQL injection, developers must also consider other web vulnerabilities that exploit input or session context. One such threat is Cross-Site Request Forgery (CSRF), where a malicious site tricks a user into submitting unintended requests to a web application where they’re authenticated.

Mitigating CSRF involves proper session management, the use of anti-CSRF tokens, and the enforcement of same-origin policies. These practices are essential complements to input validation and sanitization, ensuring a well-rounded defense against common web attacks.

V. Ruby Code

```
require 'uri'
require 'cgi'
    user_input="<script>alert('Hacked!');</script>
     user@example.com"
     def sanitize_input(input)
CGI.escapeHTML(input)
end
     def valid_email?(email)
          # email regex validation
  !!(/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i.match?(email))
end
          # Applying sanitization
safe_input = sanitize_input(user_input)
puts "Sanitized input: #{safe_input}"
          # Extracting possible email from input (e.g., from a
     string)
extracted_email    =    user_input[/[\w+\-.]+@[a-z\d\-.]+\.[a-
     z]+/i]
```

```
if extracted_email && valid_email?(extracted_email)
  puts "Valid email detected: #{extracted_email}"
else
  puts "No valid email found."
end
```

After running the code, the result of Sanitize and confirmation of the existence of the specified email address was obtained with the following output:

```
Sanitized input:
&lt;script&gt;alert(&#39;Hacked!&#39;); &lt;/script&gt;
user@example.com
Valid email detected: user@example.com
```

Sanitization involves removing potentially malicious HTML/JS files and cleaning them using CGI. The Common Gateway Interface (CGI) is a simple protocol that allows web servers to pass HTTP requests to standalone programs and return the output to a web browser, i.e, it is a set of standards that define how information is exchanged between a web server and a user script.

The address user@example.com used is not a real email address; it is a placeholder address used in documentation and examples to indicate a generic email address. The domain example.com is reserved specifically for this purpose by the Internet Assigned Numbers Authority (IANA). It is used in examples to avoid using real email addresses that could cause confusion or unintended consequences, although the code in question actually works to validate any address provided.

In the Ruby code, the validation and sanitization of the input data where performed as follows. At the beginning of the code, the following lines were used:

```
require 'uri'
require 'cgi'
user_input="<script>alert('Hacked!');</script>user@example.com"
```

where require 'uri' and require 'cgi' are the standard Ruby libraries [9].

URI can be used to parse and validate URLs, while CGI is used here to sanitize HTML input by escaping unsafe characters such as „<“, „>“ and „&“.

In this code, the sanitization function is defined by the following lines:

```
def sanitize_input(input)
  CGI.escapeHTML(input)
end
```

where CGI.escapeHTML(input) replaces HTML characters: „<“ becomes &lt; „>“ becomes &gt; Prevents **Cross-Site Scripting (XSS)** attacks by neutralizing HTML/JavaScript in input.

```
def valid_email?(email)
    !!(/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i.match?(email))
end
```

In code uses a regular expression (regex) in code to check if the string looks like an email. A regular expression, or regex for short, is a sequence of characters that defines a search pattern. This powerful tool is used by many programming languages and text editors to search, match, and process text strings. Regex is useful for tasks such as data validation, parsing, and text processing.

The regular expression /\A...\z/ anchors the match to the start and end of the string.
While the record !!(...) turns the result into a boolean (true or false).
It is worth mentioning here in a few words that the regular expression:

[\w+\-.]+ is a Local part (e.g., First name.Last name);
@ ⇒ The @ symbol;
[a-z\d\-.]+ ⇒ Domain (e.g., example); and
\.[a-z]+ ⇒ TLD (e.g., .com).

The next part of the code is:

```
safe_input = sanitize_input(user_input)
 puts "Sanitized input: #{safe_input}"
```

which takes potentially unsafe input (with HTML/JS) and escapes it for safe display or storage.

String:

```
extracted_email=user_input[/[\w+\-.]+@[a-z\d\-.]+\.[a-z]+/i]
```

uses regex to find an email-like string in the input (even if it is mixed with other data).

with the help of a code snippet

```
if extracted_email && valid_email?(extracted_email)
  puts "Valid email detected: #{extracted_email}"
else
  puts "No valid email found."
end
```

if an email is found and it passes validation, it's reported.

So the given code mainly uses:

- a regular expression (regex) to check if the string looks like an email;
- /\A...\z/ anchors the match to the start and end of the string. In regular expressions, "anchors" are special characters that don’t match actual characters in the input — instead, they match positions in the string. Anchors are used to "*anchor*" the match to certain parts of the string, like the beginning or the end;
- !!(...) turns the result into a boolean (true or false).

The article also develops a Ruby code that can be used in cases where there is a need to check the validity of phone numbers. Depending on what kind of output we want to get, that is, whether we want to output the values as text or as a logical variable, the code will have a) in the first case:

```
def valid_phone?(phone)
  # Match format: +995 599 123 456 or
    +995599123456
    regex = /\A\+?995[\s-]?5\d{2}[\s-]?\d{3}[\s-]?\d{3}\z/
      !!(regex.match?(phone))
    end
        phone_number = "+995 599 125 124"
         phone_number = "+95 599 125 124"
    if valid_phone?(phone_number)
      puts "Valid phone number"
    else
      puts "Invalid phone number"
    end
```

As a result we get:

```
    Valid phone number
  or
    Invalid phone number
```

b) In the second case we will have:

```
    def valid_phone?(phone)
      !!(/\A(\+?\d{1,2}[\s-]?)?(\(?\d{3}\)?[\s-]?)?\d{3}[\s-
    ]?\d{4}\z/.match?(phone))
    end
    puts valid_phone?("123-456-7890")
    puts valid_phone?("abc123-456-7890xyz")
    puts valid_phone?("+995-599-456-789")
    puts valid_phone?("995599456789")
    puts valid_phone?("(599) 456-789")
```

we will receive:

```
  # => true
  # => false
  # => false
  # => true
  # => false
```

In this regular expression:
/\A(\+?\d{1,2}[\s-]?)?(\(?\d{3}\)?[\s-]?)?\d{3}[\s-]?\d{4}\z/

At the beginning and end of the line, as in the previous case, anchors are used:

```
  \A = Match start of string (anchor)
  \z = Match end of string (anchor)
```

which ensures only the phone number is entered, nothing before or after.

Then comes the fragment (\+?\d{1,2}[\s-]?)?, where the country code is and the fragment (\(?\d{3}\)?[\s-]?)? where the city code is.

In the next section, the phone number \d{3}[\s-]?\d{3} will be placed as follows:

```
  \d{3} ⇒ 3 digits (prefix)
  [\s-]? ⇒ Optional space or hyphen
  \d{3} ⇒ 3 digits (line number)
```

VI. Conclusion

In the ever-evolving cybersecurity landscape, Ruby is emerging as a powerful tool for securing applications and data [10]. Its flexibility, strong community support, and extensive library ecosystem make it an effective language for addressing a wide range of security challenges. By following best practices—such as input validation, secure authentication, encryption, and safe coding—developers can harness Ruby's capabilities to build robust and secure software solutions.

As cyber threats become more sophisticated, leveraging Ruby's strengths helps developers stay ahead in the ongoing effort to protect digital systems. Ruby is a feature-rich and versatile programming language with strong support for metaprogramming, cross-platform development, and library integration. Its elegant syntax and readability promote clean, concise code—an essential factor in building and maintaining secure systems. The language's dynamic nature also facilitates rapid development and quick iteration, which is particularly valuable in cybersecurity, where timely responses to new vulnerabilities are critical.

In a world increasingly dependent on digital technologies, the importance of cybersecurity continues to grow. Threats are becoming more frequent and complex, making it vital to employ a broad range of tools and practices to safeguard data, privacy, and digital infrastructure. Ruby's adaptability enables teams to respond swiftly to emerging threats by allowing rapid modifications and updates to codebases.

Validation and sanitization, as demonstrated in this article, play a foundational role in creating secure Ruby applications. Validation ensures that user input adheres to expected formats—such as proper email addresses or phone numbers—thereby preventing malformed or malicious data from entering the system. Sanitization complements this by cleaning input to remove or neutralize harmful characters, protecting against common attacks like cross-site scripting (XSS) and SQL injection.

Together, these practices significantly enhance application security. They help maintain data integrity, reduce vulnerabilities, improve user experience, and ensure overall application stability. When combined with Ruby's strengths as a development language, these techniques empower developers to build fast, secure, and reliable systems that can adapt to the ever-changing cybersecurity landscape.

References

[1] J. Evans, *Polished Ruby Programming,* Packt Publishing, 2021.
[2] N. Rappin and D. Thomas, *Programming Ruby 3.3*, Pragmatic Bookshelf, 2024.
[3] https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/
[4] https://cybersecurityventures.com/official-cybercrime-report-2025/
[5] https://www.statista.com/forecasts/1399040/us-cybercrime-cost-annual
[6] N. Metzler, *Ruby programming,* Independently published, 2020.
[7] M. Kofler, & 10 more, *Hacking and Security: The Comprehensive Guide to Ethical Hacking, Penetration Testing, and Cybersecurity,* Rheinwerk Computing, 2023.
[8] G. Weidman, *Penetration Testing: A Hands-On Introduction to Hacking,* San Francisco, 2014.
[9] E. John, *Intro to ruby programming,* Publisher:Codemy.com, 2016.
[10] S. Metz, *Practical Object-Oriented Design: An Agile Primer Using Ruby,* Addison-Wesley Professional, 2018.
[11] https://github.com/Raunaksplanet/My-CyberSecurity-Store/blob/main/Books/Ruby%20by%20Example.pdf;
[12] E. Asabashvili, *Comparative Analysis of Ruby Language Libraries in the Field of Data Science*, The University of Georgia, 2024. https://doi.org/10.62343/csit.2024.1